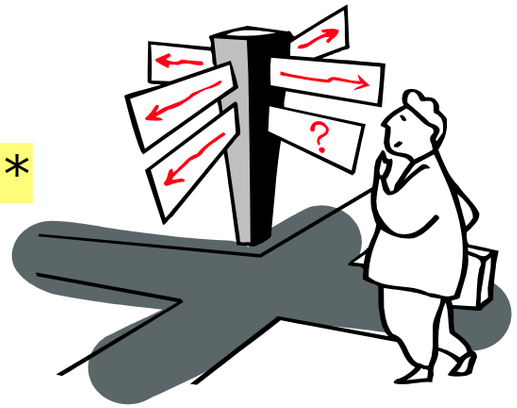
CHAPTER 7 – Formal Specification



Chapter completely revised

- Introduction
  + When, Why and What?
    ⇒ Design by contract & Testing
- Input/Output Specifications
  + Pre- and postconditions + invariants
  + Theorem proving
  + Weakest Possible Precondition
    - Statements, if-statements, loops, function calls
  + Experience report: JDK sort method
- State-Based Specifications
  + Statecharts
  + Guards, Nested states
  + Complete, Consistent, Unambiguous
  + Deduce test cases
- Formal Verification in Practice
- Conclusion
  + Correctness & Traceability

Literature

Books

+ [Ghez02] In particular, chapters “Specification” and “Verification - Analysis”
+ [Somm05] In particular, chapters “Formal Specification” & “Verification and Validation”
+ [Pres00] In particular chapters “Formal Methods” & “Cleanroom Software Engineering“

Articles

+ D. Cofer et al., "A Formal Approach to Constructing Secure Air Vehicle Software," in Computer, vol. 51, no. 11, pp. 14-23, Nov. 2018, doi: 10.1109/MC.2018.2876051.
+ de Gouw, S., de Boer, F.S., Bubel, R. et al. “Verifying OpenJDK’s Sort Method for Generic Collections.” Journal of Automated Reasoning 62, 93–126 (2019). doi.org/10.1007/s10817-017-9426-4

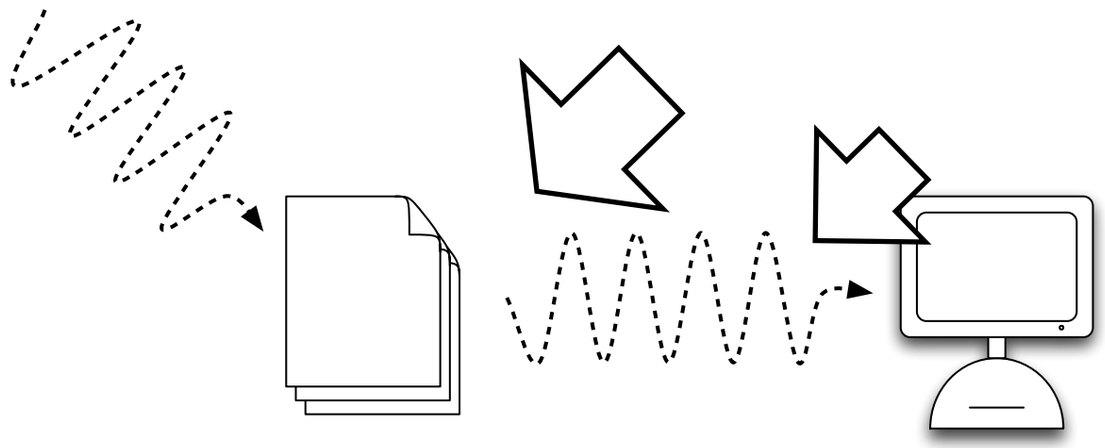
Your Opinion?

What was the most effective means to achieve “provably secure against cyberattacks”?

- ✓ 1. Modeling the system architecture and formal verification of its key security and safety properties.
- ✓ 2. synthesis of software components using languages that guarantee important security properties.
- ✓ 3. use of a formally verified micro-kernel to guarantee enforcement of communication and separation constraints specified in the architecture.
- ✓ 4. automatically building the final system from the verified architecture model and component specifications.
- ✓ 5. To assess the security of the software produced, we worked with a Red Team of professional penetration testers who evaluated our software and attempted to identify vulnerabilities.

When Formal Specification?

Correctness
- ~~Are we building the right product? = VALIDATION~~
- Are we building the product right? = VERIFICATION



Mistakes are possible (likely!?)
- while transforming requirements into a system

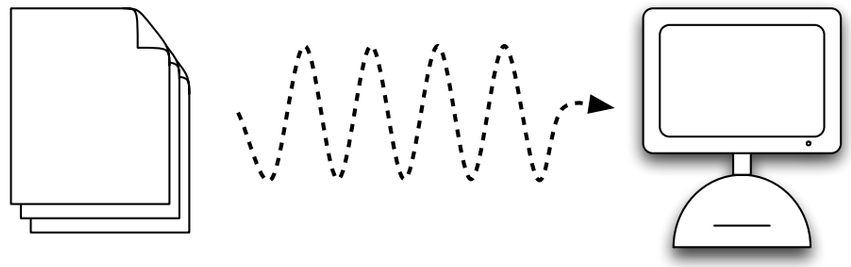
Formal Specification is used for
- (detailed) design
  + specify and verify key properties of *system under design*
    - e.g. State-based Specifications
- formal verification
  + mathematical proof: code is “correct”
    - e.g. Input/output specifications

Why Formal Specification?

- Software projects rely more and more on “Buy” than on “Build”
  + Cheaper, more reliable, ...
  + Companies focus in-house development on core business
    - Buy 3rd party components for functionality outside the core
  + 3rd party components evolve
    - require well-specified interface

- Buy vs. Build
  + But if we buy we need to specify
    - clearly
    - unambiguously
    - completely

⇒ Formally



Why do we Care?

It is possible to build high-quality products without formal specifications!
(And it is possible to build low-quality products with formal specifications)

- **Fact**
  + For most systems it is more cost effective to apply other techniques (reviews, tests, ...)
    - business systems, information systems, ...
    - Most software development is in that area

- **Fact**
  + For some areas, the benefits more than outweigh the costs
    - high-risk systems: human lives depend on the software
      > (because reliability is such a big issue)
    - embedded systems: software controlling hardware
      > (because components evolve at different rates)

    - standards: defining information exchange protocols
      > (because the same specification is reused by a lot of implementations)

} **Cyber-Physical Systems**

Sooner or later you will be confronted with one of these!

What are Formal Specifications?

- What is a ...

  + **Specification**. A description of desired system properties.
    - Preferably the “what” and not the “how”

  + **Informal specification**. Specification in natural language.
    - augmented with figures, tables, examples, scenarios

  + **Semi-formal specification**. Specification based on a notation with precise syntax but loose semantics.
    - e.g. UML class & sequence diagrams

  + **Formal Specification**. Specification based on a formal model with precise syntax & semantics.

Testing and Design by Contract

Formal foundation: formal syntax + formal semantics

- Possible to mathematically prove that a given system satisfies the specification

A system is correct with respect to its specification !
Note: faults (omissions!) in the specification are still possible

Testing

- Formal specifications ⇒ black-box testing

  test-cases: complete coverage, thus highest probability of finding mistakes

Design by Contract

- Formal specifications ⇒ natural pre- and post conditions

Is this Valid? Output?

```
<p>
  <strong>
    <em>
      Hello World
    </em>
  </strong>
</p>
```

```
<p>
  <strong>
    <em>
      Hello World
    </strong>
  </em>
</p>
```

A) Input/Output Specifications

- include logic assertions (pre & post-conditions + invariants) inside an algorithm
- verify termination and correctness via stepwise formal reasoning

Example: Input/Output Specification for a binary search procedure

```
procedure Binary_search (Key : ELEM ; T: ELEM_ARRAY;
    Found : out BOOLEAN; L: out ELEM_INDEX) ;

Pre-condition
    T'LAST - T'FIRST ≥ 0 and   -- not empty
       for_all i, -- universal qualifier
           T'FIRST ≤ i ≤ T'LAST-1, T (i) ≤ T (i + 1) --sorted

Post-condition
    (Found and T (L) = Key)  or
        (not Found and
        not (exists i, -- existential qualifier
            T'FIRST ≤ i ≤ T'LAST, T (i) = Key))
```

Proving Correctness

Goal:

- mathematically prove that post-condition is always satisfied when pre-condition is true

Termination?

- While loop terminates if Found or Bott > Top
- If an element = key exists, Found is set true
- In a loop execution either Found := true, Bott >> or Top <<
- Initially, Top > Bott thus (if Found remains false) eventually Bott > Top

Correctness?

- Loop invariant is “true” on entry to the loop.
- Assertion 2 follows because of the successful test Key = Mid
- Assertion 3 follows because the array is ordered. If T (Mid) < Key all values up to T (Mid) must also be less than the key
- Assertion 4 follows by substituting Bott-1 for Mid (if T(mid) != Key)
- Assertions 5 and 6. Similar argument to 3 and 4
- After loop execution, either the key has been found or there is no value in the array which has been searched which matches the key. However, Bott > Top so all the array has been searched

⇒ Therefore, the binary search routine code conforms to its specification

Intermediate Assertions

```
01.     Bott := T'FIRST; Top := T'LAST ;
02.     L := ( T'FIRST + T'LAST ) mod 2; Found := T( L ) = Key;
03.     -- 1 . ASSERT (Found and T(L) = Key) or ((not Found)
04.     -- and (not Key in T(T'FIRST..Bott-1, Top+1..T'LAST)));
05.     while Bott <= Top and not Found loop
06.             Mid := (Top + Bott) / 2;
07.             if T( Mid ) = Key then
08.                 Found := true; L := Mid;
09.             --  2. ASSERT Key = T(Mid) and Found;
10.             elsif T( Mid ) < Key then
11.                 --  3. ASSERT not Key in T(T'FIRST..Mid);
12.                 Bott := Mid + 1;
13.             --  4. ASSERT not Key in T(T'FIRST..Bott-1);
14.             else
15.                 --  5. ASSERT not Key in T( Mid..T'LAST );
16.                 Top := Mid - 1;
17.             --  6. ASSERT not Key in T(Top+1..T'LAST);
18.     end if;
19.     end loop;
```

Automated Theorem Provers

Compiler support to prove that
when pre-condition is true post-condition will (should) always be satisfied.

(Sometimes counter examples when proof does not hold.)

The KeY Project

Java

ethereum

solidity

VeriFast

THEOREM PROVER

TLA+

intel

aws

Microsoft Azure

elastic

Dafny

https://dafny.org/

Hoare Logic (revisited)

Let:

S series of statements

{P} and {Q} are *properties*

{P} is the precondition

{Q} is the postcondition

Then:

{P} S {Q}

is a *Hoare Triple* meaning

"Any execution of A starting in a state where P holds,

~~must~~ *should* terminate in a state where Q holds"

Example: ∀ x positive Integer

{x = 5} x := x * 2 {x > 0}

Adapted from Chapter 5. Design by Contract

Partially Correct / Totally Correct

Then:

{P} S {Q}

is a Hoare Triple meaning

"Any execution of A starting in a state where P holds, *should* terminate in a state where Q holds"

The implementation of S with respect to its specification is …

- **Partially correct.**
  + Assuming the precondition is true just before the function executes, then *if the function terminates*, the postcondition is true.
    - Infinite loops, raising exceptions, … is allowed

- **Totally correct.**
  + Again assuming the precondition is true before function executes, the function *is guaranteed to terminate* and when it does, the postcondition is true.

Stronger (and Weaker)

Let {P1} and {P2} be conditions expressed via predicates

- {P1} is *stronger* then {P2} iff
  + {P1} ≠ {P2}
  + {P1} ⇒ {P2}
- {P1} is *weaker* then {P2} iff
  + {P1} ≠ {P2}
  + {P2} ⇒ {P1}

- example
  + {x = 5} x := x * 2 {x > 0}
  + {x = 5} x := x * 2 {x > 5 and X < 20}

    - {x > 5 and X < 20} is stronger than {x > 0}
      > stronger is better for a post-condition
        (it is more precise about the outcome)

Adapted from Chapter 5. Design by Contract

Strongest Postcondition

Consider the Hoare triple
{P} S {Q}

if ∀ Q’ such that {P} S {Q’}, Q ⇒ Q’

then Q is the *strongest postcondition* of S with respect to P

Denoted with sp(S, Q)

Quizz

- example
  + {x = 5} x := x * 2 {x > 0}
  + {x = 5} x := x * 2 {x > 5 and X < 20}

What is the *strongest postcondition* for this Hoare triple?

+ {x = 5} x := x * 2 {……}

Deducing the Strongest Postcondition

Consider the Hoare triple
{P} S {Q}

When we know {P} and S we can deduce sp(S,P)

For assignment
{P} x:= E {x = E}

Assignment with operation
{x+y = 5} x := x + z {x’ + y = 5 and x = x’ + z}

x’ represents the “old” value of x, thus before S executes

Weakest Precondition

For proving correctness it makes more sense(*) to calculate the inverse:

- Given a statement S and a postcondition Q,
  + what is the weakest possible precondition?

Consider the Hoare triple

{P} S {Q}

If ∀ P’ such that {P’} S {Q}, P’ ⇒ P, then

P is the *weakest precondition* of S with respect to Q.

Denoted with wp(S, Q)

(*) Why does it make more sense to find the weakest possible precondition?

- It represents the least amount of work to prove correctness
- Too strong a pre-condition may imply that we cannot prove correctness

Weakest Precondition (assignment)

Consider the Hoare triple
{P} x:= E {Q}

Then the weakest precondition means that we should

- replace each occurrence of x in Q with E
  - denoted with [E/x] Q
- and then substitute in the precondition.

Thus {[E/x] Q} x := E; {Q}

{……} x := x - 2; {x > 0}

Fill in the weakest precondition for {……}

Weakest Precondition (multiple statements)

Consider the Hoare triple

$\{P\}\ S_1; S_2; \ldots S_n\ \{Q\}$

Then the weakest precondition is deduced backwards.

$\{P\} = wp(S_1; S_2; \ldots S_n, Q)$

$= wp(S_1; S_2; \ldots S_{n-1}, wp(S_n, Q))$

$= wp(S_1, wp(S_2, wp(\ldots wp(S_n, Q)\ldots)))$

$\{P\}$ x:= z + 1; y:= x + y $\{y > 5\}$

step 1: wp(y := x + y, y > 5)

[x + y / y] y > 5

(x + y > 5)

step 2: wp(x := z + 1, x + y > 5)

[..... / .....] x + y > 5

(..... > 5)

Fill in the weakest precondition for $\{P\}$

Weakest Precondition (if statement)

Consider the Hoare triple
{P} if C then S else T; {Q}

Then the weakest precondition means that we should
- calculate the result depending on C being true or false
- and then substitute in S or T branch.

Thus wp (if C then S else T, Q)

= (c ⇒ wp(S, Q)) ∧ (¬c ⇒ wp(T, Q))

= (c ∧ wp(S, Q)) ∨ (¬c ∧ wp(T, Q))

Legend

∧	logical and
∨	logical or
¬	negation (not)

{……} if (x> y) then z := x else z := y; {z = max(x,y)}

Fill in the weakest precondition for {……}

Loops

Consider the Hoare triple
{P} while C do S; {Q}

Proof by induction - introduce (i) loop invariant and (ii) loop variants

- (i) loop invariant I — what will ensure the postcondition?
  + The invariant is initially true (base case): $P \Rightarrow I$
  + Each loop step preserves the invariant (inductive step): $\{I \wedge C\}\ S\ \{I\}$
  + After the loop terminates the postcondition is true: $\{\neg C \wedge I) \Rightarrow Q$
- (ii) loop variant — what guarantees that the loop terminates?
  = a monotonically decreasing function integer-value function v
  + a strictly decreasing with every step: $\{I \wedge C \wedge v = V\}\ S\ \{I \wedge v < V\}$
  + when v reaches zero the loop terminates: $\{I \wedge v \leq 0\} \Rightarrow \neg C$

Quizz

What is the *weakest precondition* for the following loop?

+ {……} while (x > 0) do x := x-1; {x = 0}

in pseudo code

```
function int useless_loop(x int) {
    require (……);
    while (x > 0)do x := x - 1;
    ensure (x == 0);
}
```

Quizz — step 1 - loop *in*variant

What is the *weakest precondition* for the following loop?

+ {……} while (x > 0) do x := x-1; {x = 0}

- establish loop invariant I where {I ∧ C} S {I}
  + {I ∧ (x > 0)} x := x -1 {I}
- look for the weakest pre-condition
  + I ∧ (x > 0) = wp(x := x -1, I)
  + I ∧ (x > 0) = [x - 1 / x] I
- Which I would resolve the above?
  + I = ……
  + ……
- Does it terminate the loop? {¬C ∧ I) ⇒ Q
  + {¬C ∧ ……) ⇒ x = 0

Quizz — step 2 - loop variant

What is the *weakest precondition* for the following loop?

+ {……} while (x > 0) do x := x-1; {x = 0}

- establish loop variant so that {I ∧ C ∧ v = V} S {I ∧ v < V}
  + {(x ≥ 0) ∧ (x > 0) ∧ v = V} x := x-1; {x ≥ 0 ∧ v < V}
- choose x for v
  + {(x ≥ 0) ∧ (x > 0) ∧ x = V} x := x-1; {x ≥ 0 ∧ x < V}
- substitute x-1 for x in the postcondition; always true?
  + ……
  + ……
- when v reaches zero the loop terminates: {I ∧ v ≤ 0} ⇒ ¬C
  + {(x ≥ 0) ∧ x ≤ 0} ⇒ ¬(x > 0)
  + ……
  + ……

Function (Procedure) Calls

Consider the Hoare triple

$\{P\}\ S_{prev};\ F(\ldots);\ S_{next};\ \{Q\}$

where F is a function call / procedure call / method invocation / …

- > F is considered a black box
- > We need a pre- and postcondition for F
  - * Must be provided by the developer of F

Postcondition for F? = weakest precondition for S_{next}.

Postcondition for S_{prev}? = strongest precondition for F.

Example

Open Access | Published: 31 August 2017

Verifying OpenJDK's Sort Method for Generic Collections

Stijn de Gouw ✉, Frank S. de Boer, Richard Bubel, Reiner Hähnle, Jurriaan Rot & Dominic Steinhöfel

Journal of Automated Reasoning **62**, 93–126 (2019) | Cite this article

2991 Accesses | **15** Citations | **1** Altmetric | Metrics

Abstract

TimSort is the main sorting algorithm provided by the Java standard library and many other programming frameworks. Our original goal was functional verification of TimSort with mechanical proofs. However, during our verification attempt we discovered a bug which causes the implementation to crash by an uncaught exception. In this paper, we identify conditions under which the bug occurs, and from this we derive a bug-free version that does not compromise performance. We formally specify the new version and verify termination and the absence of exceptions including the bug. This verification is carried out mechanically with KeY, a state-of-the-art interactive verification tool for Java. We provide a detailed description and analysis of the proofs. The complexity of the proofs required extensions and new capabilities in KeY, including symbolic state merging.

Your Opinion?

What do you think happened with the bug report on the broken Java.utils.Collection.sort ()?

- ☐ The suggested fix was correctly incorporated; the sort method now is provably correct.
- ☐ They fixed the symptom and not the root cause; the risk is reduced but still not correct.
- ☐ The bug report was ignored because the fault could not be reproduced (i.e. "Works for me”).
- ☐ The bug report was closed without fix, because it was a low risk bug.

What actually happened …

We favored the second suggestion which is to formalize the invariant as originally intended and to fix the code of the method mergeCollapse that is responsible for reestablishing the invariant. We were able to formally and mechanically prove that this fixed version of the algorithm is correct in the sense that the stack lengths are sufficient and no ArrayIndexOutOfBoundsException is thrown. We describe this fix and its verification in Sect. 4.3 below.

In the aftermath of our discovery, it turned out that the bug was present in several implementations of TimSort. Besides in (Open)JDK, the bug was present in
(1) its original Python implementation,
(2) Android,
(3) an independent Java implementation used by Apache Lucene, as well as
(4) a Haskell implementation.

All of these projects fixed the bug within a short time frame. The OpenJDK project was the only one where the bug was fixed by just increasing the allocated array lengths, which is in our opinion sub-optimal, and there is no machine checked proof of that fix. All other projects implemented our second suggestion and fixed the underlying problem.

Cited from … (with slight lay-out changes)
- de Gouw, S., de Boer, F.S., Bubel, R. et al. "Verifying OpenJDK's Sort Method for Generic Collections." Journal of Automated Reasoning 62, 93–126 (2019). doi.org/10.1007/s10817-017-9426-4

(a) suggested fix was correctly incorporated

(b) fixed the symptom not the root cause

B) State-Based Specifications

Typically based on the notion of finite state machines

- describes the sequence of states a system is supposed to go through … in response to external stimuli (a.k.a. events)

StateCharts

- widely used: present in UML
- commonly used in real-time systems

- Definitions

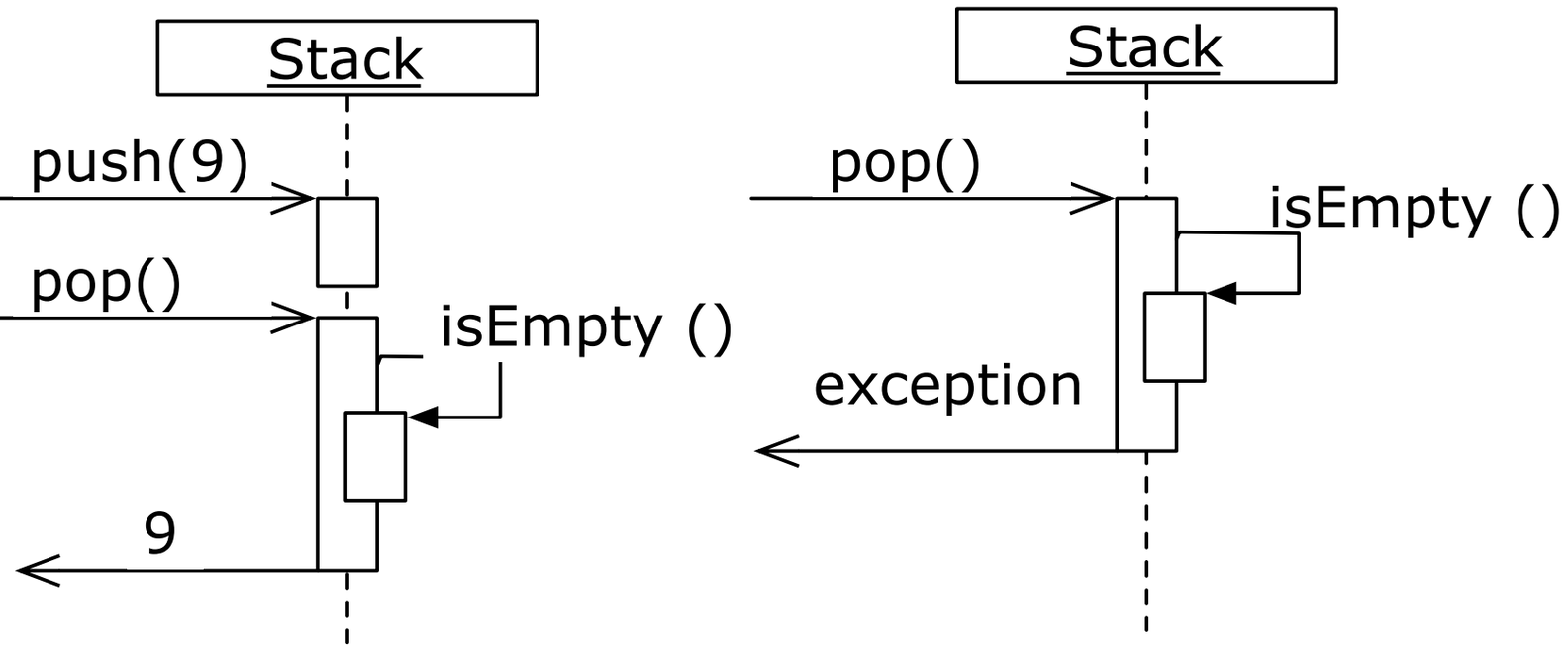
  + state = a condition an object satisfies
    (i.e. a predicate computing its result with the
    attribute values of the object)
      > The state can be observed from the outside !

  + transition = a change of state triggered by an event, condition or time

Sequence Diagrams

a) happy day scenario: pop of a non-empty stack

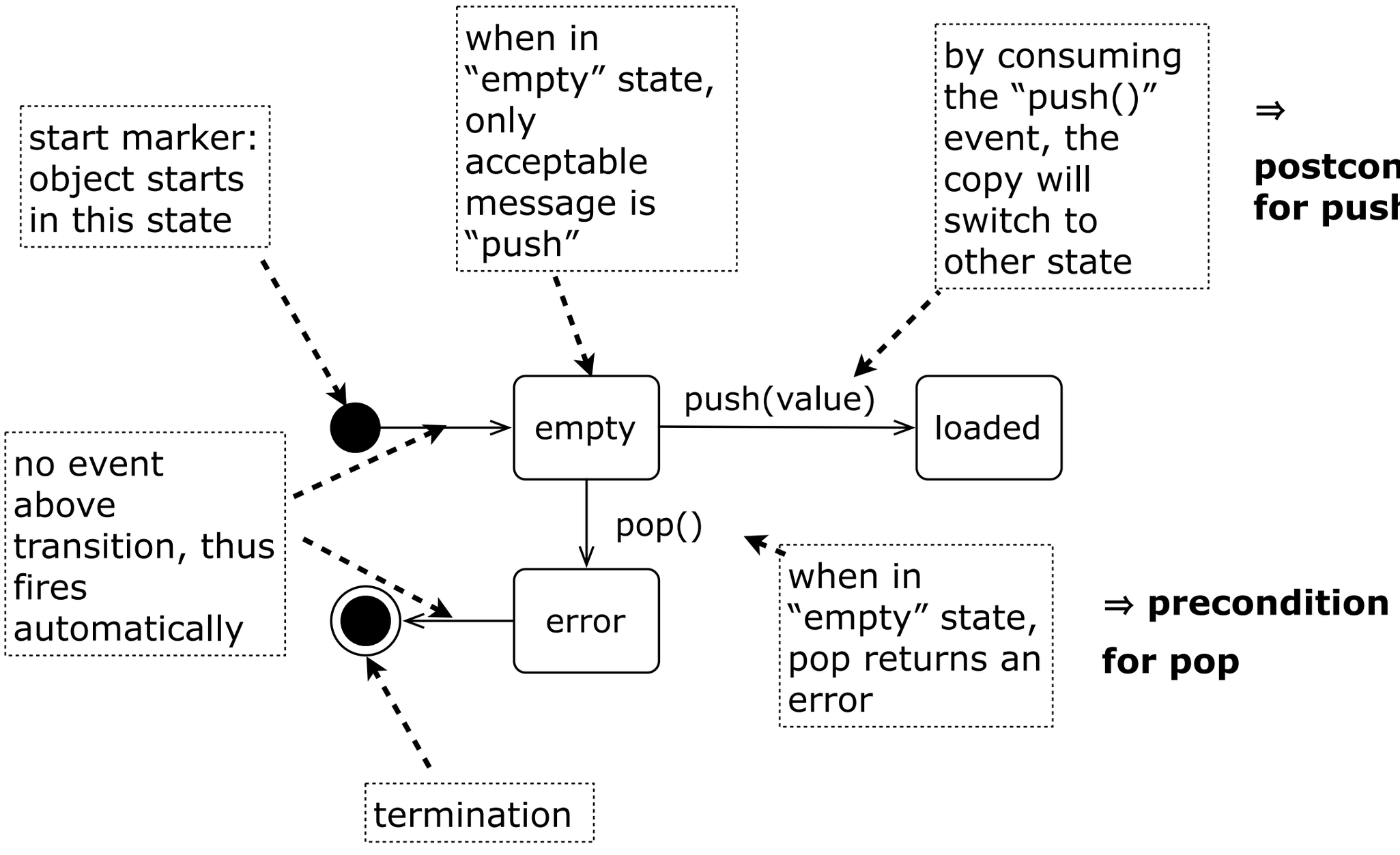
b) secondary scenario: pop of an empty stack



- What are acceptable message sequences for a stack?
- What is the union of all possible scenarios?

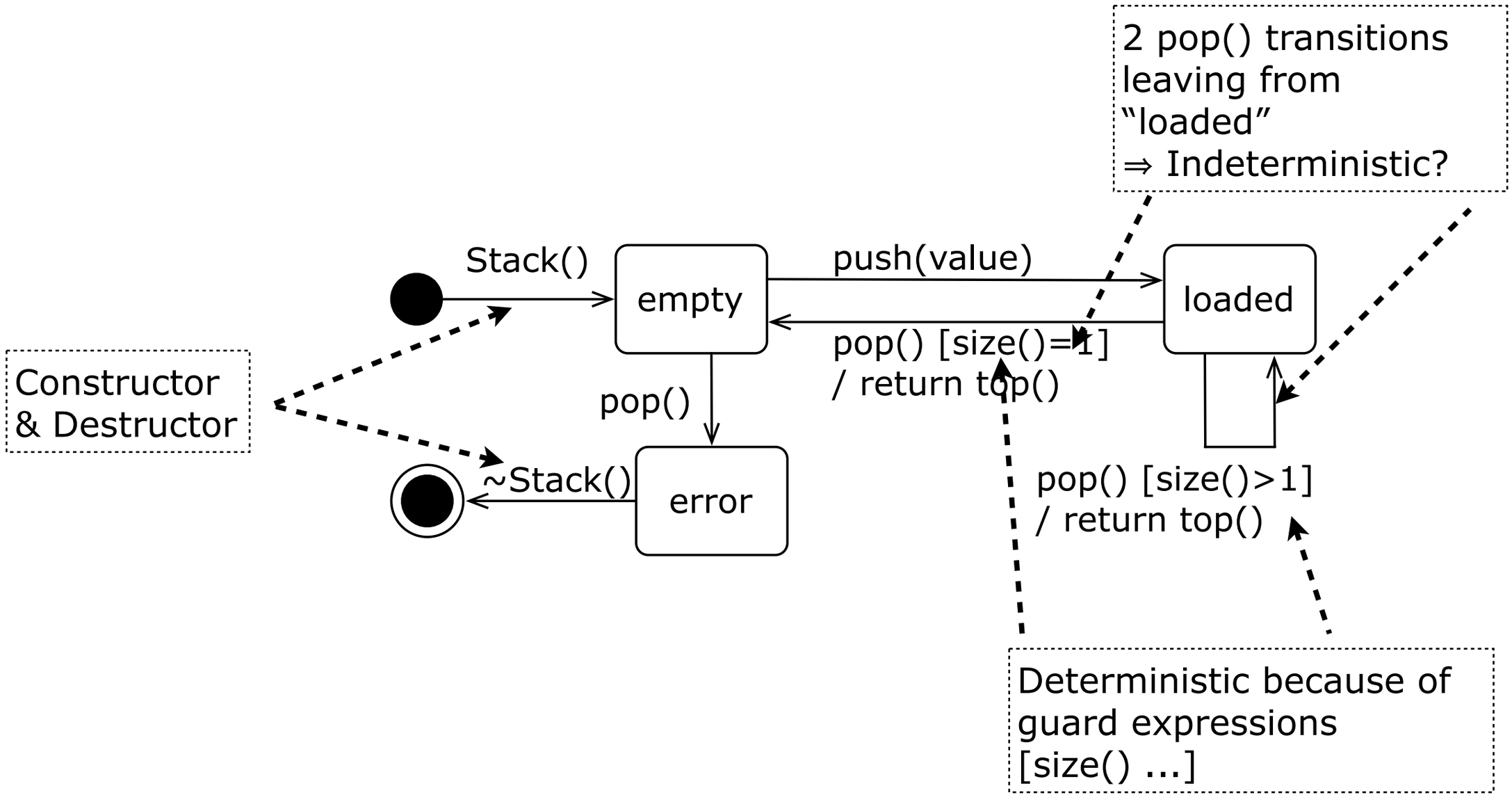
⇒ A statechart allows to specify all valid (also all invalid) scenarios

Statechart for a Stack



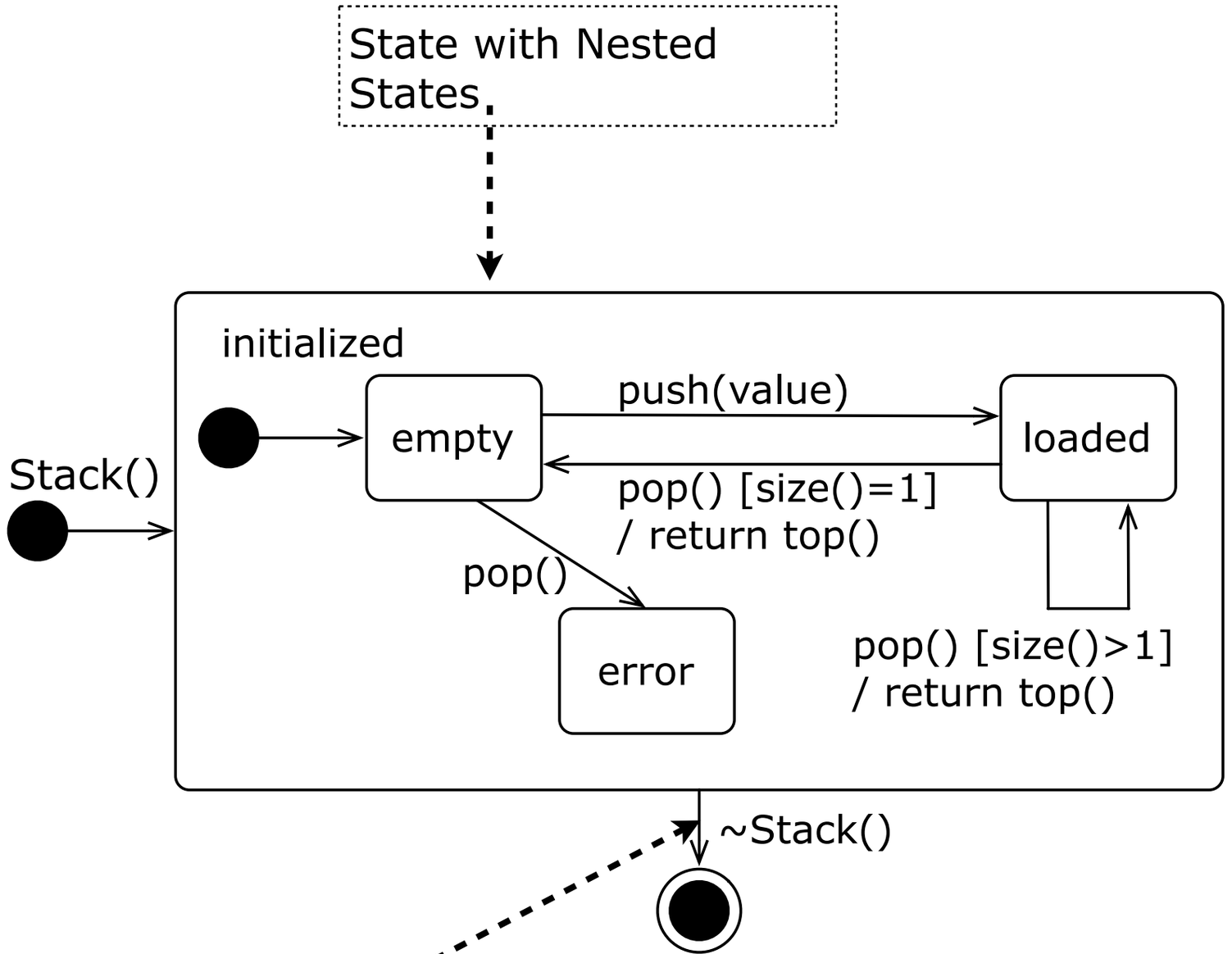
What about a pop() on a loaded stack?

Guarded Transition



What about destructors on empty/loaded state?

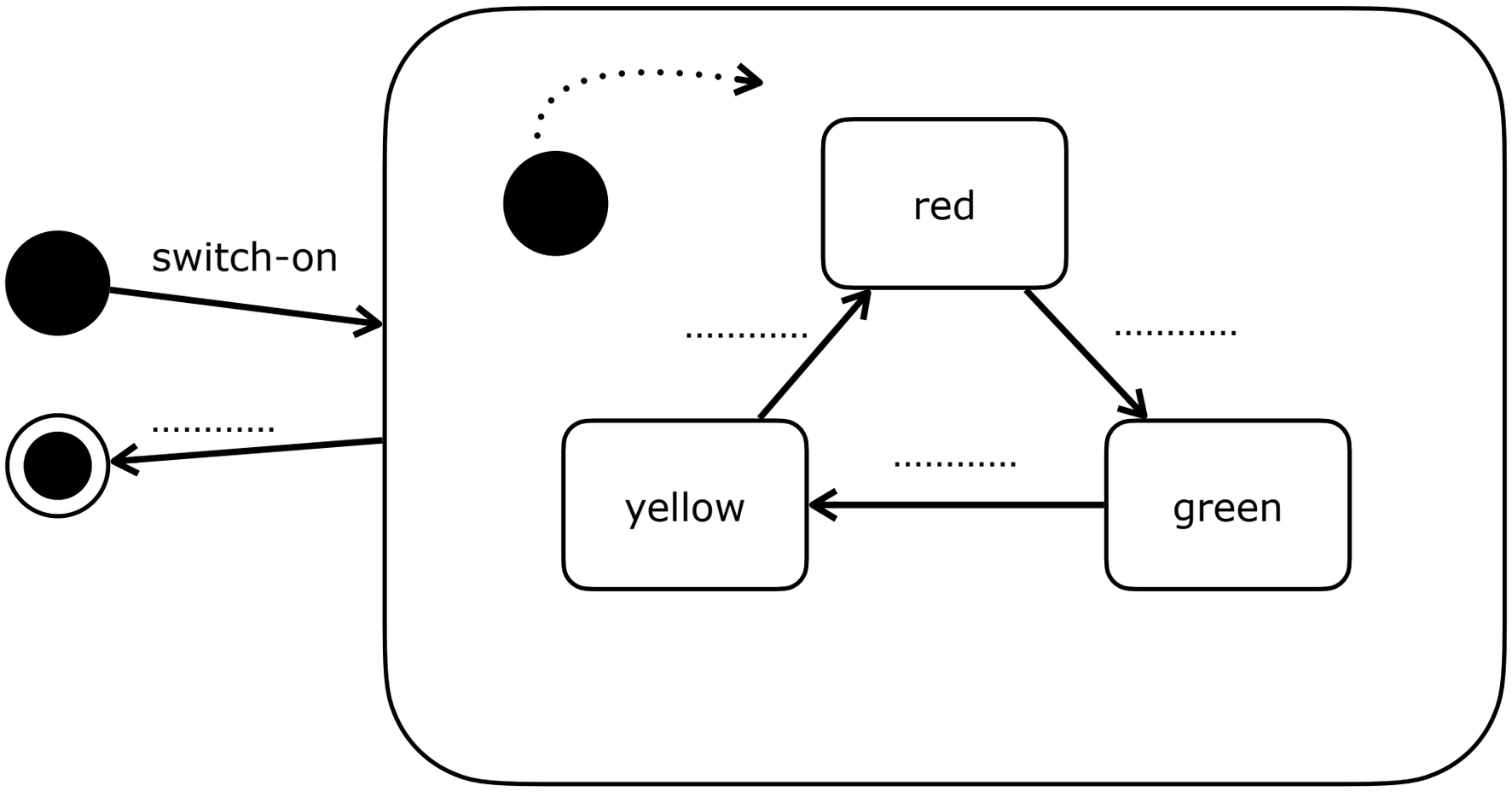
Nested State



shorthand for multiple transitions with same events and target states

Quiz

Complete the Traffic Light Statechart



Consistent / Complete / Unambiguous

When is a state-based specification …

- Complete
  + every event/state pair has a transition
    - Create table: events (incl. guards) x state
      one cell contains target state
    - all cells should have a target state

- Consistent
  + every state is reachable from initial state
    & final state is reachable from every other state
    - Breadth-first spanning tree; root is initial state
    - all leaf nodes of the graph should be terminal state

- Unambiguous (= deterministic)
  + same event (incl. guard) does not appear on more than one transition leaving any given state
    - Verify using table created in completeness

Deducing Test Cases

Test cases

+ cover all state transitions at least once (*)

- define a predicate for each state,
  + which answers whether object is in that state
    - (Thus `initialized() — empty() — loaded() — error()`)

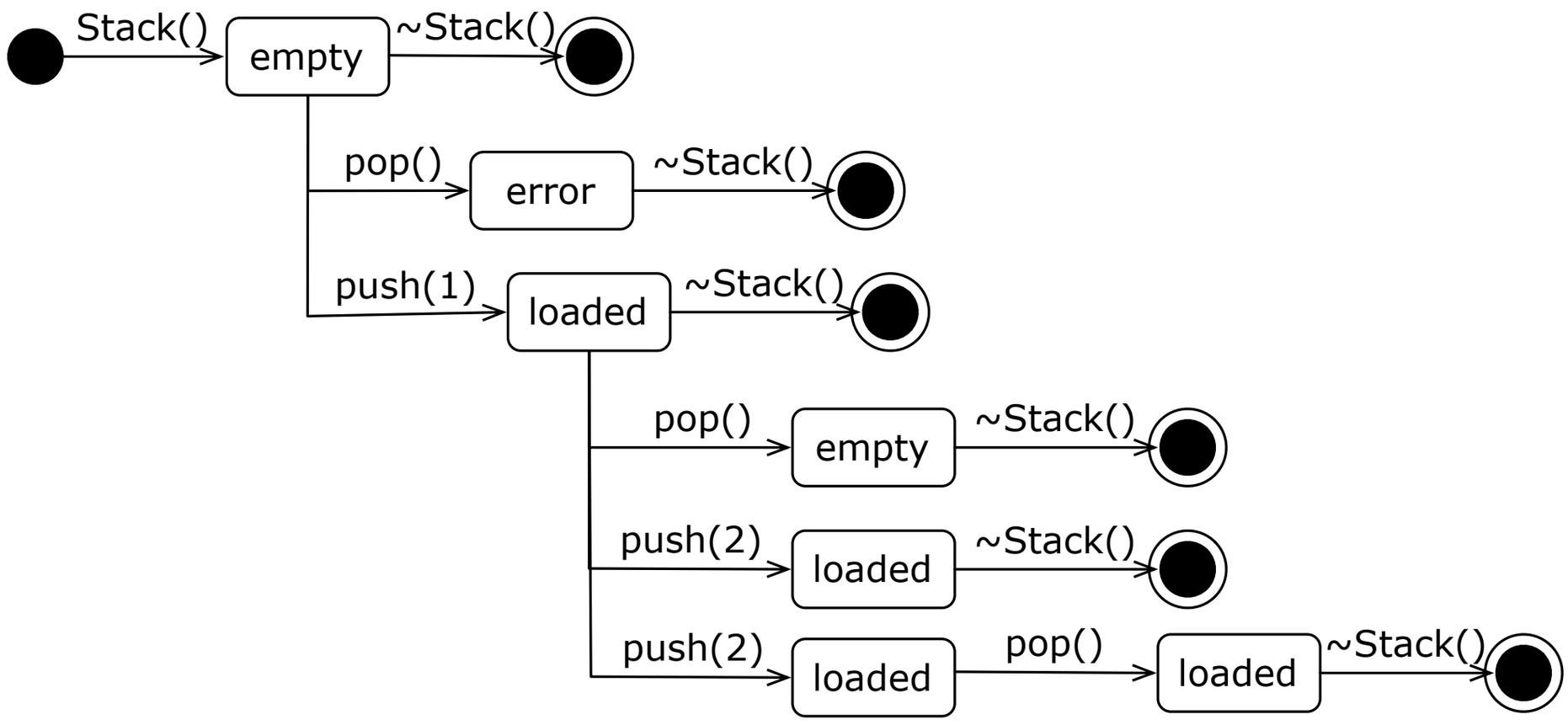
- test-cases must cover the breadth-first spanning tree
  + construct with same table used to verify completeness
    - rows and columns = events (incl. guards) x state
      cell contains target state

(*) Stronger coverage is possible:

- cover all sequences of state transitions of length n
- force all guards
- force all guards with boundary values
- …

Test cases for Statechart “Stack” (1/2)

	empty	loaded	error	initial
Stack()	empty (??)	loaded (??)	error	empty
push	loaded	loaded	error (??)	error
pop() [size()=1]	error	empty	error (??)	error
pop() [size()>1]	error	loaded	error (??)	error
~Stack	terminated	terminated	terminated	error



Test cases for Statechart “Stack” (2/2)

```
s := Stack();
assertTrue(initialised(s));
assertTrue(empty(s));
s.~Stack();
```

```
s := Stack();
assertTrue(initialised(s));
assertTrue(empty(s));
pop(s);
assertTrue(error(s));
s.~Stack();
```

```
s := Stack();
assertTrue(initialised(s));
assertTrue(empty(s));
push(s, 1);
assertTrue(loaded(s));
s.~Stack();
```

```
s := Stack();
assertTrue(initialised(s));
assertTrue(empty(s));
push(s, 1);
assertTrue(loaded(s));
pop(s);
assertTrue(empty(s));
s.~Stack();
```

```
s := Stack();
assertTrue(initialised(s));
assertTrue(empty(s));
push(s, 1);
assertTrue(loaded(s));
push(s, 2);
assertTrue(loaded(s));
s.~Stack();
```

```
s := Stack();
assertTrue(initialised(s));
assertTrue(empty(s));
push(s, 1);
assertTrue(loaded(s));
push(s, 2);
assertTrue(loaded(s));
pop(s);
assertTrue(loaded(s));
s.~Stack();
```

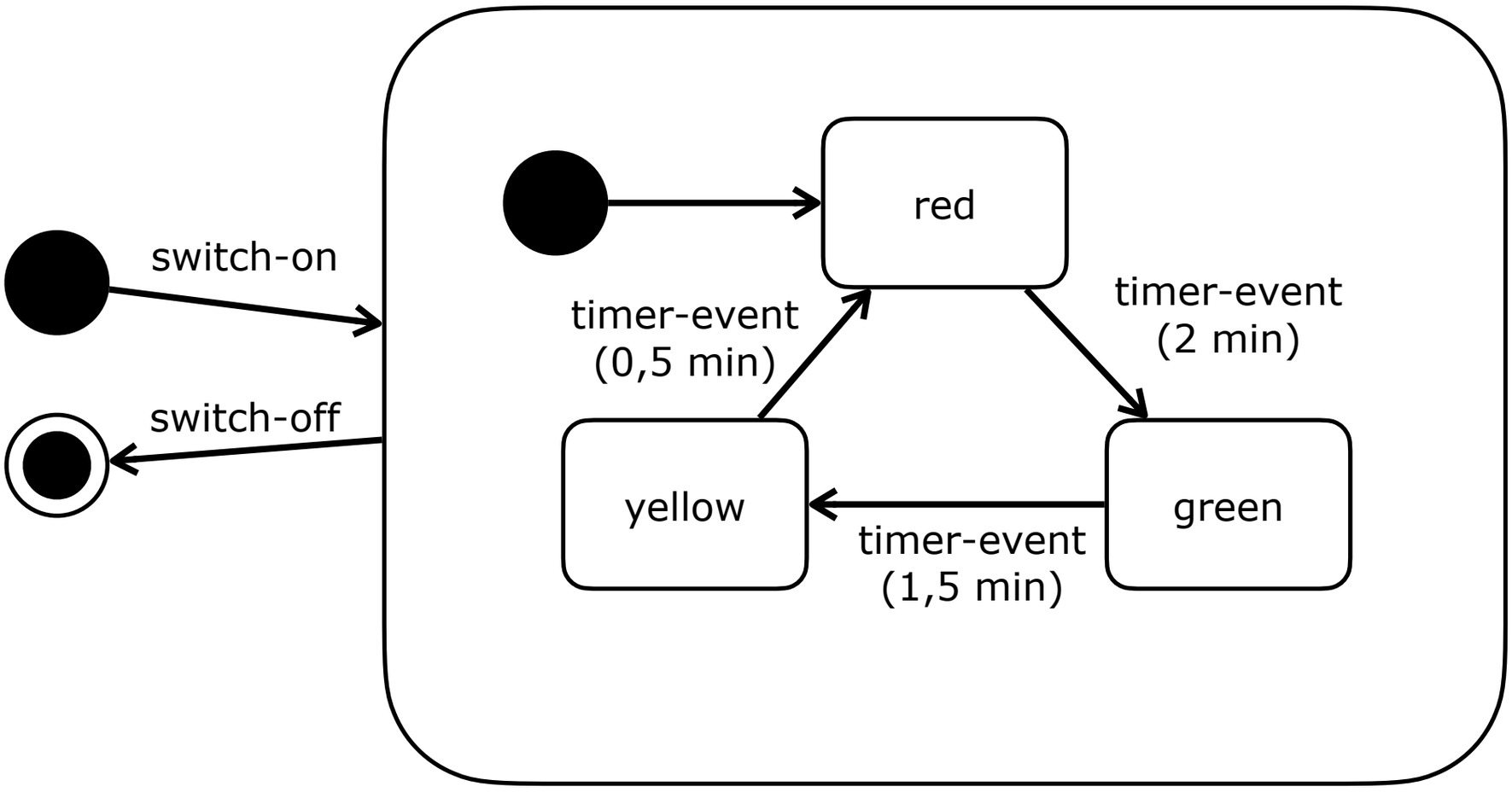
State-based Specifications Revisited

State-based Specifications

- Are particularly suitable for specifying “acceptable” message sequences
  + unify effect of all possible scenarios on one class in one statechart
  + “unacceptable” implies precondition
  + state change implies postconditions
    > Design by Contract

- Specify acceptable message sequences as paths through a graph
  > cover all paths
  > Path Testing

Example (advanced): Traffic Light (1/2)

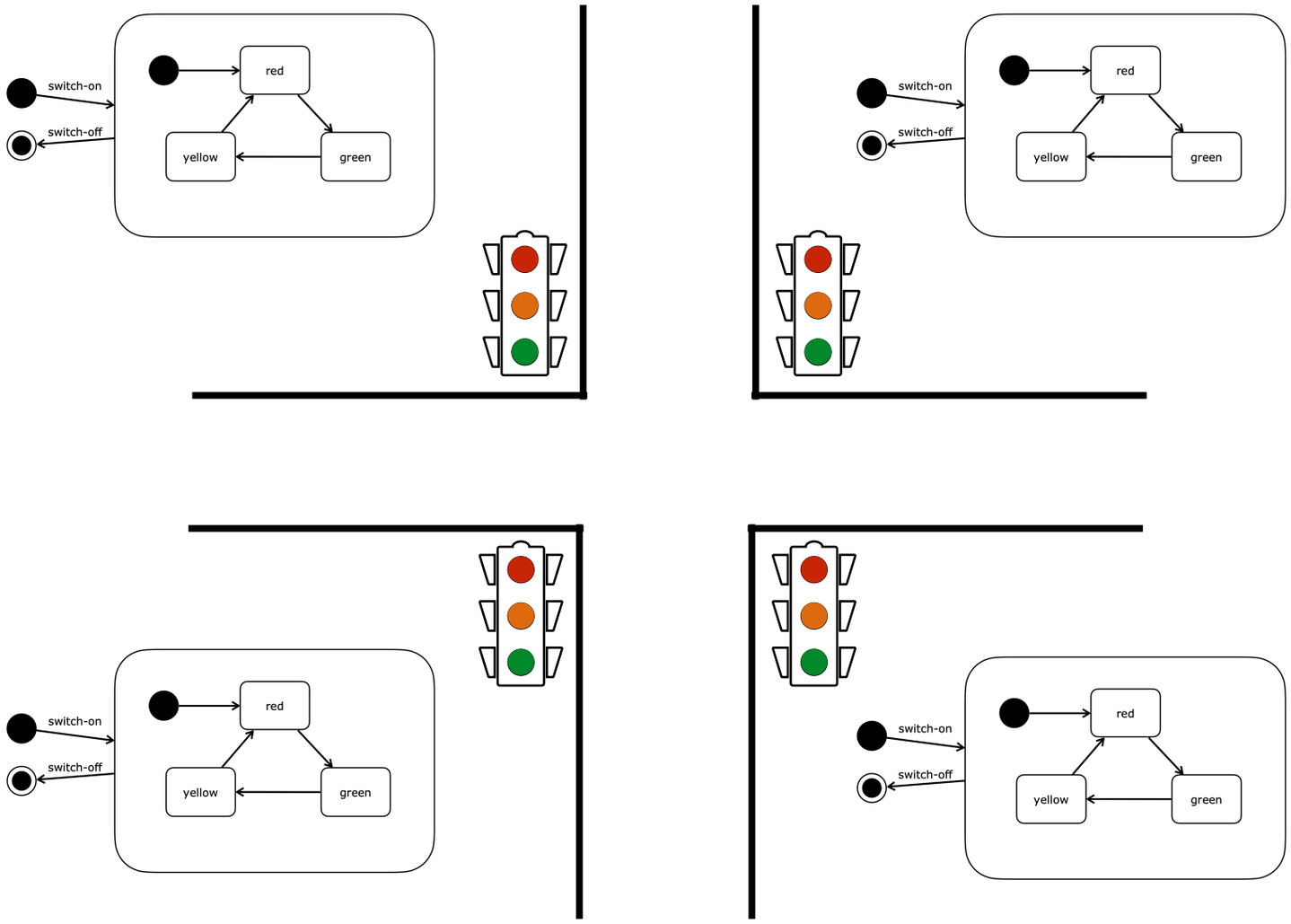


What is the starting state of this statechart?
Is this what you want?

Example (advanced): Traffic Light (2/2)

- safety: “something bad never happens”
- liveness: “something good eventually happens”
- fairness: “if something may happen frequently, it will happen”

} formal verification + simulation + testing



Formal Verification in Practice (1/2)

A Formal Verification Study on the Rotterdam Storm Surge Barrier

Ken Madlener, Sjaak Smetsers & Marko van Eekelen

Conference paper

1051 Accesses | 1 Citations

Part of the Lecture Notes in Computer Science book series (LNPSE,volume 6447)

A lightweight model of the C++ code and the Z specification of the component was manually developed in the theorem prover PVS. As a result, some essential mismatches between specification and code were identified.

Windows 7

DOI:10.1145/1866724.1866743

SLAM is a program-analysis engine used to check if clients of an API follow the API's stateful usage rules.

BY THOMAS BALL, VLADIMIR LEVIN, AND SRIRAM K. RAJAMANI

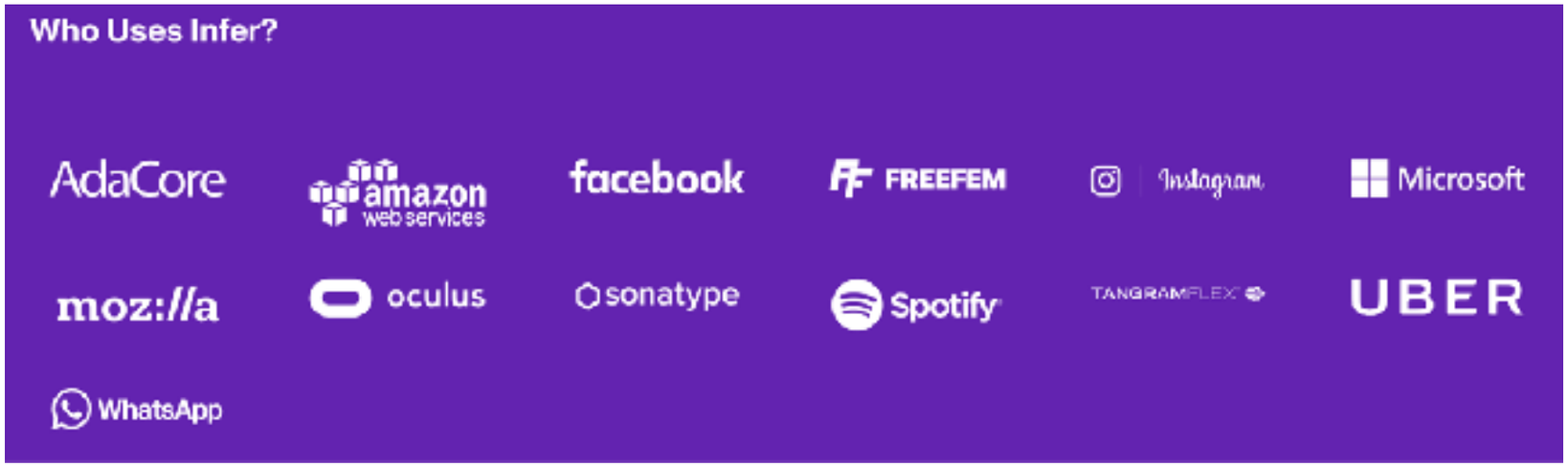
A Decade of Software Model Checking with SLAM

SDV was applied later in the cycle after all other tools, yet found 270 real bugs in 140 WDM and WDF drivers.

Formal Verification in Practice (2/2)

Infer

A tool to detect bugs in Java and C/C++/Objective-C code before it ships



aws

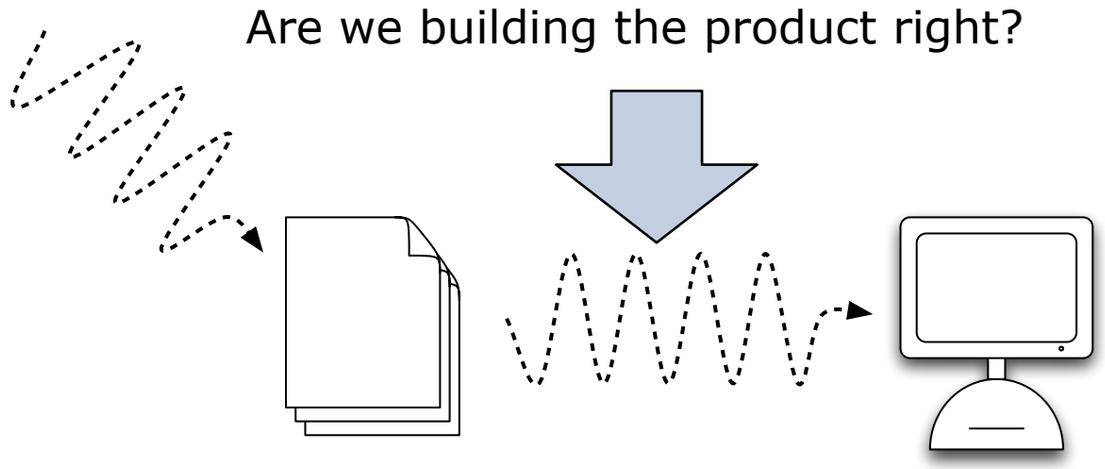
Provable Security

Security Assurance, Backed by Mathematical Proof

We are committed to helping you achieve the highest levels of security in the cloud. We've developed automated reasoning tools that use mathematical logic to answer critical questions about your infrastructure to detect misconfigurations that could potentially expose your data. We call this provable security because it provides higher assurance in the security of the cloud and in the cloud.

The Verification Landscape

Copied
06.Software Testing



Formal Specifications

Simulation

Testing

Specification and Verification
6 ECTS-credits 1E SEM
Lecturer(s): Guillermo Alberto Perez

Mathematical Foundations of Reinforcement Learning
6 ECTS-credits 1E SEM
Lecturer(s): Guillermo Alberto Perez Benny Van Houdt

Modelling of Software-intensive Systems
6 ECTS-credits 1E SEM
Lecturer(s): Hans Vangheluwe

Software Testing
6 ECTS-credits 2E SEM
Lecturer(s): Serge Demeyer

Correctness & Traceability

- Correctness
  + Are we building the system right?
    - Formal specifications allow to verify presence of desired properties
      * Mathematical proof
      * Semi-automatic generation of test-cases
    - Faults (omissions!) in the specification are still possible

  + Are we building the right system?
    - (Some) formal specifications can be simulated / animated
      * May play the role of a prototype
      * Counterexamples to illustrate corner case behaviour

- Traceability
  + Requirements ⇔ System?
    - Formal specification is an intermediate representation
      * Traceability depends on usage and discipline

Summary(i)

You should know the answers to these questions

- Why is an UML class diagram a semi-formal specification?
- What is an automated theorem prover?
- What is the distinction between “partially correct” and “totally correct”?
- Give the mathematical definition for the *weakest precondition* of Hoare triple {P} S {Q}
- Why is it necessary to complement sequence diagrams with statecharts?
- What is the notation for the start and termination state on a state-chart? What is the notation for a guard expression on an event?
- What does it mean for a statechart to be
  (a) consistent, (b) complete, and (c) unambiguous?
- How does a formal specification contribute to the correctness of a given system?

You should be able to complete the following tasks

- Use a theorem prover (Daphny) to prove that a given piece of code is correct.
- Create a statechart specification for a given problem.
- Given a statechart specification, derive a test model using path testing.

Summary(ii)

Can you answer the following questions?

- (Based on the article “A Formal Approach to Constructing Secure Air Vehicle Software”.)
  + What is according to you the most effective means to achieve “provably secure against cyberattacks”?
- Why is it likely that you will encounter formal specifications?
- Explain why we need both the loop variant and the loop invariant for proving total correctness of a loop?
- What do you think happened with the bug report on the broken Java.utils.Collection.sort ()? *Why* do you think this happened?
- Explain the relationship between “Design By Contract” on the one hand “State based specifications” on the other hand.
- Explain the relationship between “Testing” on the one hand and “State based specifications” on the other hand.
- You are part of a team build a fleet management system for drones transporting medical goods between hospitals. You must secure the system against cyber-attacks. Your boss asks you to look into formal specs; which ones would you advise and why?