Black Box Testing (A&O Ch. 4)

(2nd Ed. Ch. 6)

Course Software Testing & Verification

2024/25

Wishnu Prasetya & Gabriele Keller

Plan

- Partitioning based testing (A&O Ch 4/2nd Ed Ch. 6).
- Model based testing

Note: black box testing is an important concept. For example, system level testing is often done in a black box setup. In this lecture we will discuss two important techniques that are commonly used in such a setup. The first one, partition based testing, is discussed in length in A&O. However, the second one, model based testing (MBT), is only lightly touched in A&O. This lecture will introduce you to some basic concepts of MBT. For a more practical exposition on MBT you can look at e.g. Practical Model-Based Testing by Utting - Lageard (UU students can access it for free from Scienc Direct, https://www.sciencedirect.com/science/book/9780123725011)

White box testing

Testing is "**white box testing**" if you have knowledge of the source code of the target program to help you designing the tests, and your tests have in principle access to all the program's variables so that they can inspect them if they are in the correct state.

- Compare this with Def 1.26 A&O (2nd Ed. See p 26).
- White box setup is common at the unit-testing level.

Black box testing

Testing is ”**black box testing**” if you **do not** assume full knowledge of the inner working of the target program. Usually this also entails that your tests have only limited access to the program state to inspect its correctness.

- Compare with Def 1.25 in A&O (2nd Ed. See p26).
- Note that although we may have full access to the source code, at the system level we may choose not to use this knowledge because it becomes to complicated to comprehend.

Black box testing is common at the system-testing level. We will discuss two approaches : (1) **Partition-based testing**, Ch 4 (2nd ed. Ch. 6) and (2) **Model-based testing**.

Partition-based testing

- Premise:

The input space of a program can be partitioned into "equivalence classes", such that inputs from the same partition/equivalence-class lead to the "same kind" of behavior.

- It then makes sense to require that every partition should be tested at least once.
- Even without source code, we can often **propose** a reasonable partitioning.

Recall this example...

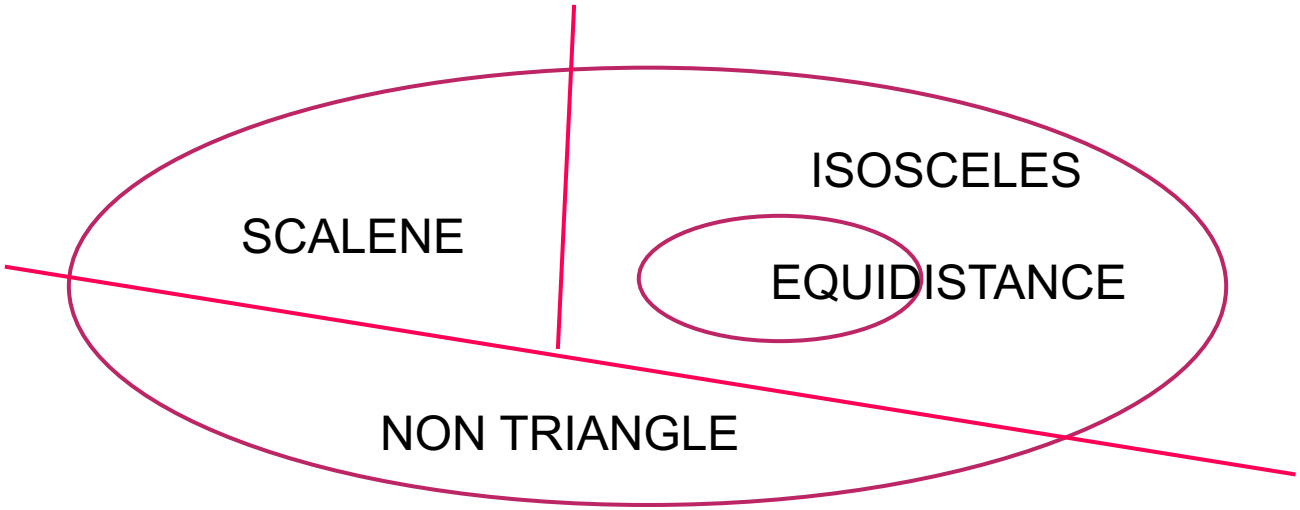
Enum TriType { Isosceles, Equilateral, Scalene }

TryType triangle(Float a, Float b, Float c) { ... }

If a, b, c represent the sides of a triangle, this methods determines the type of the triangle.

Let's now try to come up with test cases for this function from the black box perspective.

Partitioning triangle()'s domain



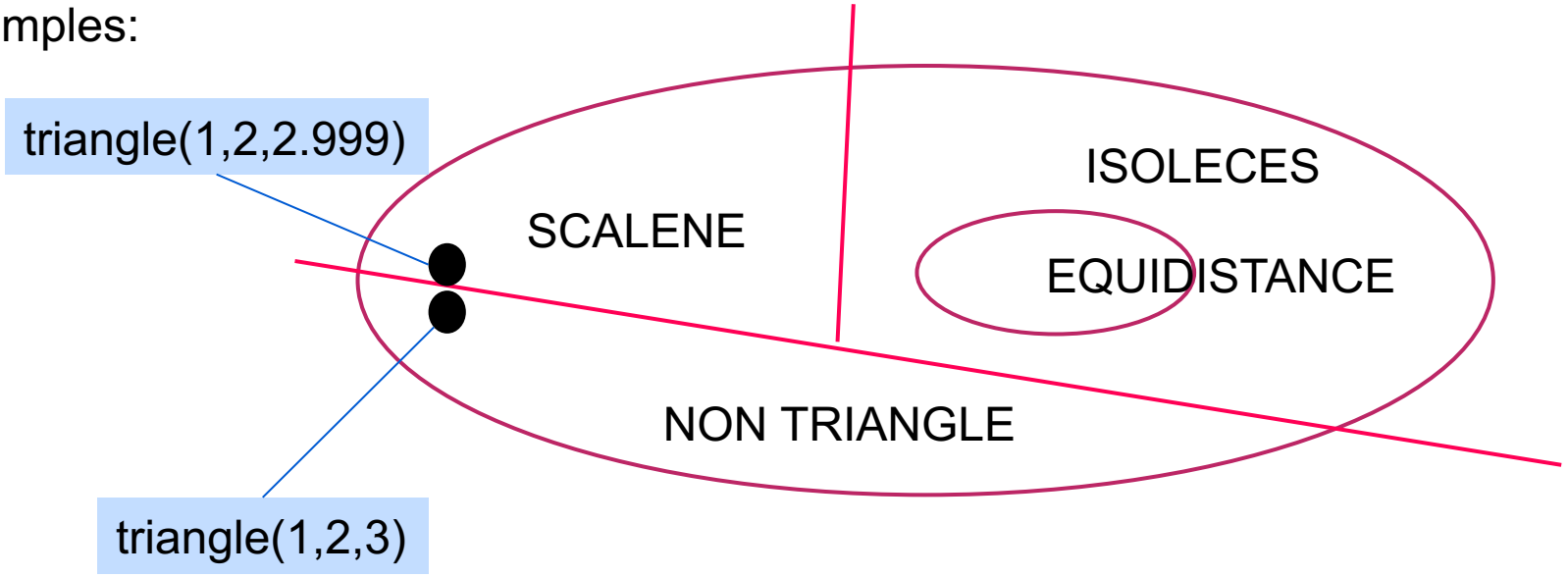
input domain of triangle

This suggests 4 test-cases, one for every partition.

Boundary value test

Errors often lurk in the "boundaries" between partitions → test values on and around the boundaries.

Examples:



Example 2

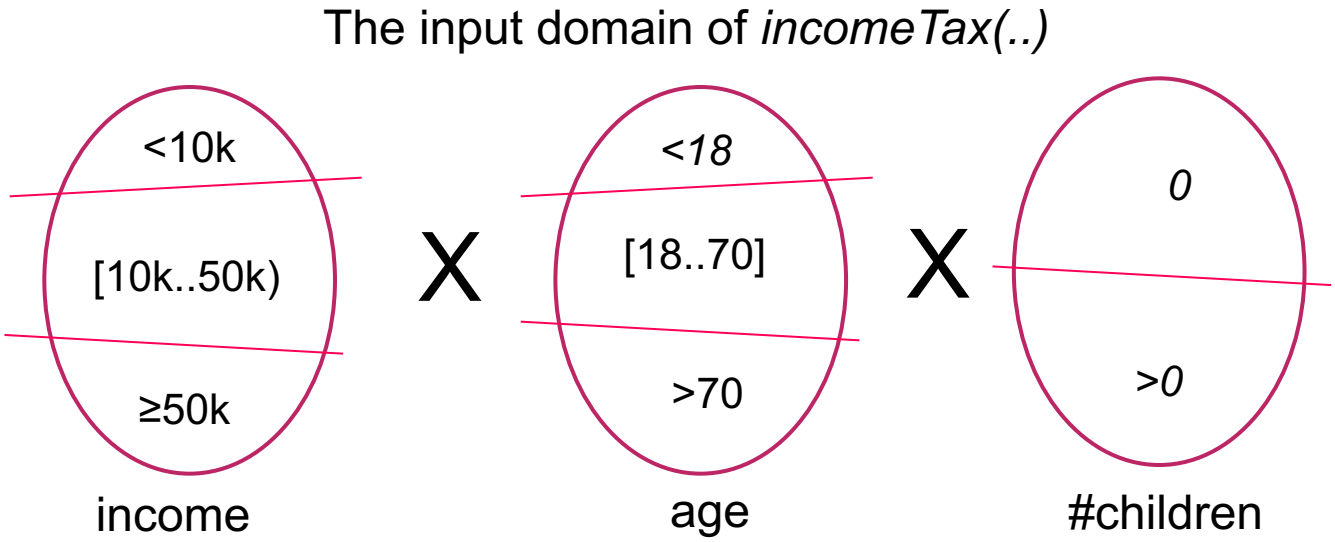
int **incomeTax**(int income, int age, int children)

Fragments from its informal specification: *the method calculates income tax:*

- *Income below 10K is not taxed.*
- *A person under 18y or above 70y is not taxed, as long as the income is below 50K.*
- *Tax reduction applies, linear to the number of children the person has.*

This method has a more complex input space than the "triangle" example.

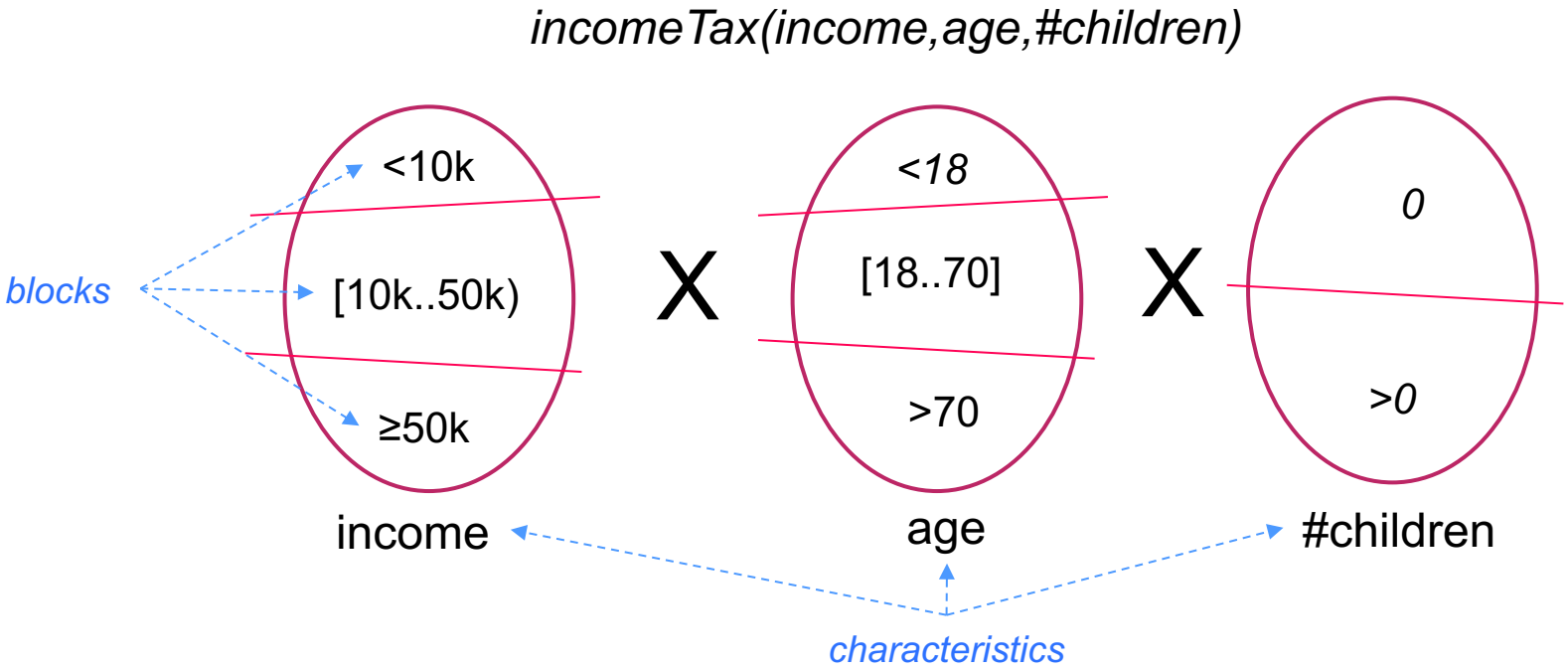
Consider the following partitioning



- As before, we can try to come up with test cases that would cover every partition.
- Note that just 3 test cases can cover all partitions!

→ does not feel very strong. This is because the approach ignores that different parameters may ”interact”. E.g. normally when the age<18 the person will not be taxed. However, the combination of income≥50k and age<18 will trigger its own behavior, namely that the person will be taxed anyway.

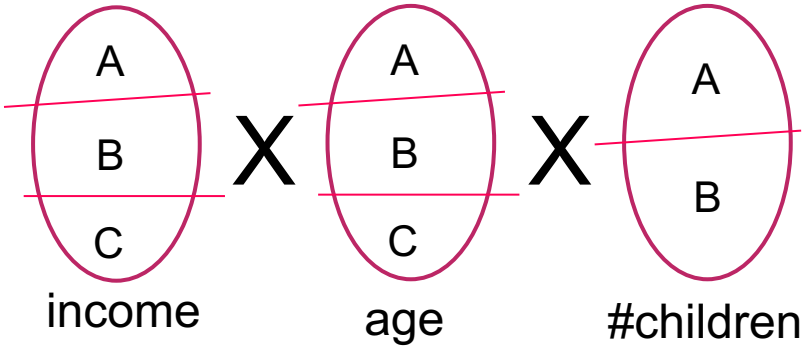
Combinatoric testing



- In this example the input domain is spanned by 3 characteristics, and there are in total 8 blocks.
- A block combination over the characteristics, e.g. (<10k,<18,0) abstractly specifies a test case.
- There are in total 3x3x2 = 18 such combinations of blocks.

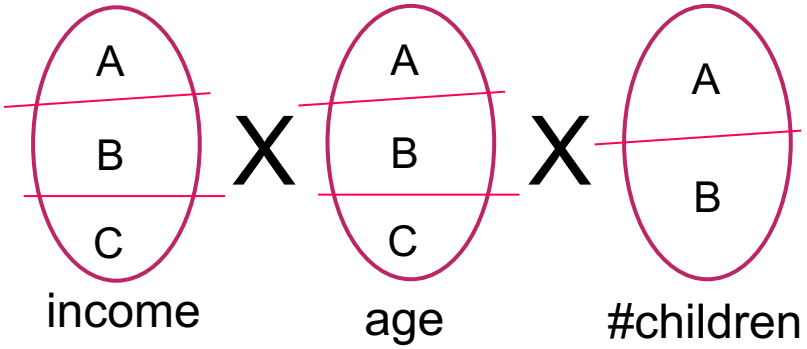
Combinatoric testing

Let's name the blocks for convenience:



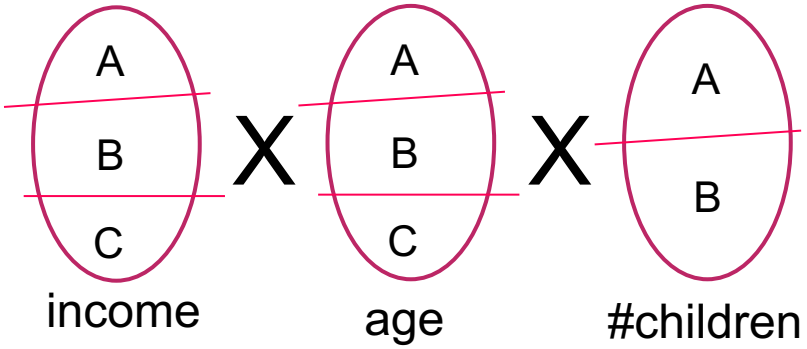
- Stats: 8 blocks, 18 combinations.
- (C4.24/2nd Ed. C6.2, **EACH CHOICE** coverage) Each block must be tested.
  $|T| = (\mathbf{max}\ i: 0 \le i < k: B_i)$; usually too weak.
- (C4.23/2nd Ed. C6.1, **ALL** coverage) All combinations must be tested.
  $|T| = (\Pi\ i: 0 \le i < k: B_i)$; does not scale up.

How about to just cover all pairs?



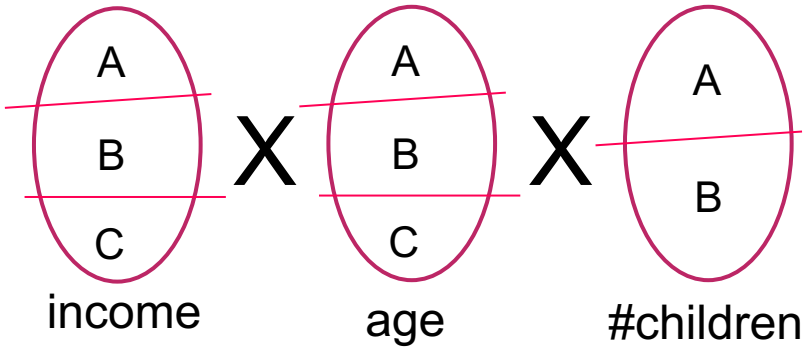
- Premise: most of the times, only *t* out of *N* parameters actually influence the behavior. E.g:
  - when the income is less than 10k, there will be no tax, regardless the age and #children.
  - When the income is ≥50k, there will be tax, regardless the age.
- If we speculate on *t*=2, this leads to "pair-wise" testing: to cover all block-pairs over different characteristics.

Pair-wise and t-wise testing



- (C4.25/2nd Ed. C6.3, **pair-wise coverage**). Each pair of blocks (from different characteristics) must be tested.
- There are 9+6+6 = 21 pairs to cover. But we can cover them with just 9 test cases.
- Pair-wise coverage is stronger than EACH CHOICE, and still scalable.
- (C4.26/2nd Ed. C6.4, **t-wise coverage**). Generalization of pair-wise.

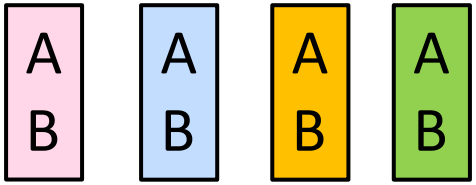
Example: a test set with full pair-wise coverage



	income	age	#children
Test-case 1	A	A	A
Test-case 2	A	B	B
Test-case 3	A	C	A
Test-case 4	B	A	B
Test-case 5	B	B	A
Test-case 6	B	C	B
Test-case 7	C	A	A
Test-case 8	C	B	B
Test-case 9	C	C	Does't matter

PWC, example 2

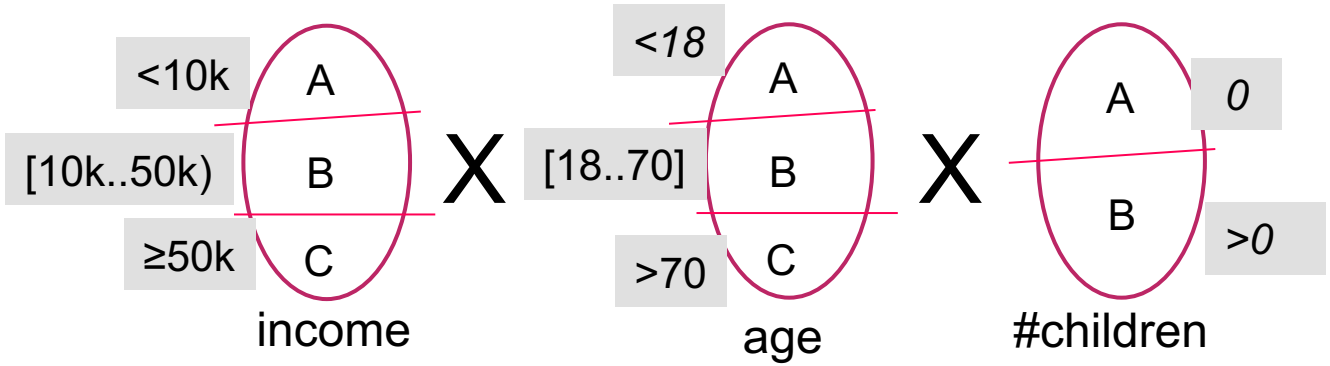
- Four characteristics, each with two blocks:



- Minimal test set that gives PWC:

	cha1	cha2	cha3	cha4
tc1	A	A	A	B
tc2	B	B	A	A
tc3	A	B	B	B
tc4	B	A	B	B
tc5	A	A	B	A

Consider again this test set



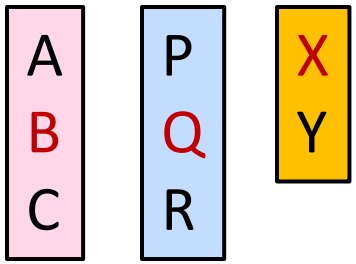
	income	age	#children
tc1	A	A	A
tc2	A	B	B
tc3	A	C	A
tc4	B	A	B
tc5	B	B	A
tc6	B	C	B
tc7	C	A	A
tc8	C	B	B
tc9	C	C	A

- Despite giving full pwc these tests arguably miss some important cases:
  - tax reduction on mid ages: (B,B,B)
  - tax reduction on young parent: (C,A,B)
- Solution: add constraints, but... (see next slide).

Pair-wise and t-wise testing

- Consider a program P with N characteristics [1..N]. For simplicity, suppose **each** characteristic is split into B number of blocks.
- Total number of pairs = $B^2 * V_2$ where V_2 is the number of subsets of size 2 out of N characteristics. So, $V_2 = \binom{N}{2} = \frac{N!}{2!*(N-2)!}$
- To cover all pairs you need at least B^2 number of test cases. Expect that it can be more (as in Example 2).
- In general, finding a minimum size test set that gives full t-wise coverage is **not** trivial.
- However, as pointed out before, k-wise testing ignores the "semantic" (that some combinations should be included or excluded (because they are not sensical)). We can add constraints, though this makes the problem of calculating the minimum test set even harder.

Adding a bit of semantic



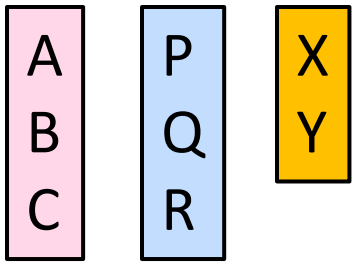
Example: t_0 = **(B,Q,X)**, generates these additional test requirements :

(B,Q,Y)
(A,Q,X) (C,Q,X)
(B,P,X) (B,R,X)

(C4.27/2nd Ed. C6.5, **Base Choice Coverage**, BCC) Decide a single *base test* t_0. Make more tests by each time removing one block from t0, and forming combinations with all remaining blocks (of the same characteristics).

$$|T| = 1 + (\Sigma i : 0 \leq i < k : B_i - 1)$$

Consider again



Base test t_0 = (**B,Q,X**), and these additional test, giving full BCC:

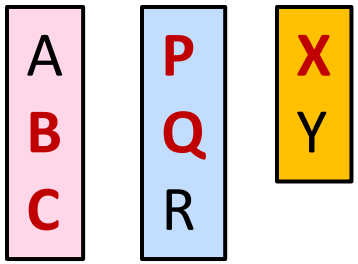
(B,Q,Y)

(A,Q,X) (C,Q,X)

(B,P,X) (B,R,X)

- What if we also need to insist on testing out all combinations of (C,P,-) ?
- Proposal: use multiple “base tests”

MBCC



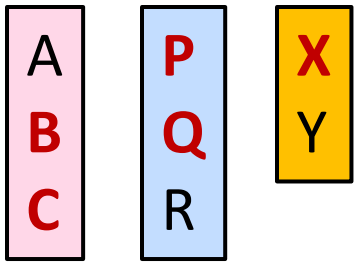
Decide the "base blocks": (red)
Choose one or more base tests. They can only use base blocks. E.g.:

base test t_0 = **(B,Q,X)**
base test t_1 = **(C,P,X)**

(C4.28/2nd Ed. C6.6, **Multiple Base Choices** coverage). For each characteristic we decide at least one *base block*. Then decide a *set* of base tests; each only include base blocks. For each base test, generate more tests by each time removing one base block, and forming combinations with *remaining non-base* blocks.

$$|T| \textit{ at most } \quad M + M^*(\Sigma i : 0 \leq i < k : B_i - m_i)$$

Example MBCC



The base blocks are marked red. The base tests:

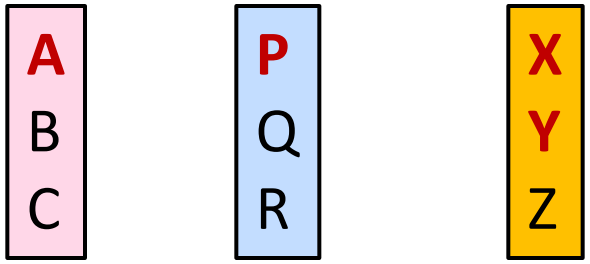
base test t_0 = **(B,Q,X)**

base test t_1 = **(C,P,X)**

We need to add these tests to get full MBCC:

- Varying t_0 over non-base blocks:
  (A,Q,X), (B,R,X), (B,Q,Y)
- Varying t_1 over non-base blocks:
  (A,P,X), (C,R,X), (C,P,Y)
- Remove duplicates (there are none above)

Example-2, MBCC



***Red** : base blocks*

Chosen base tests = **(A,P,X)**, **(A,P,Y)**

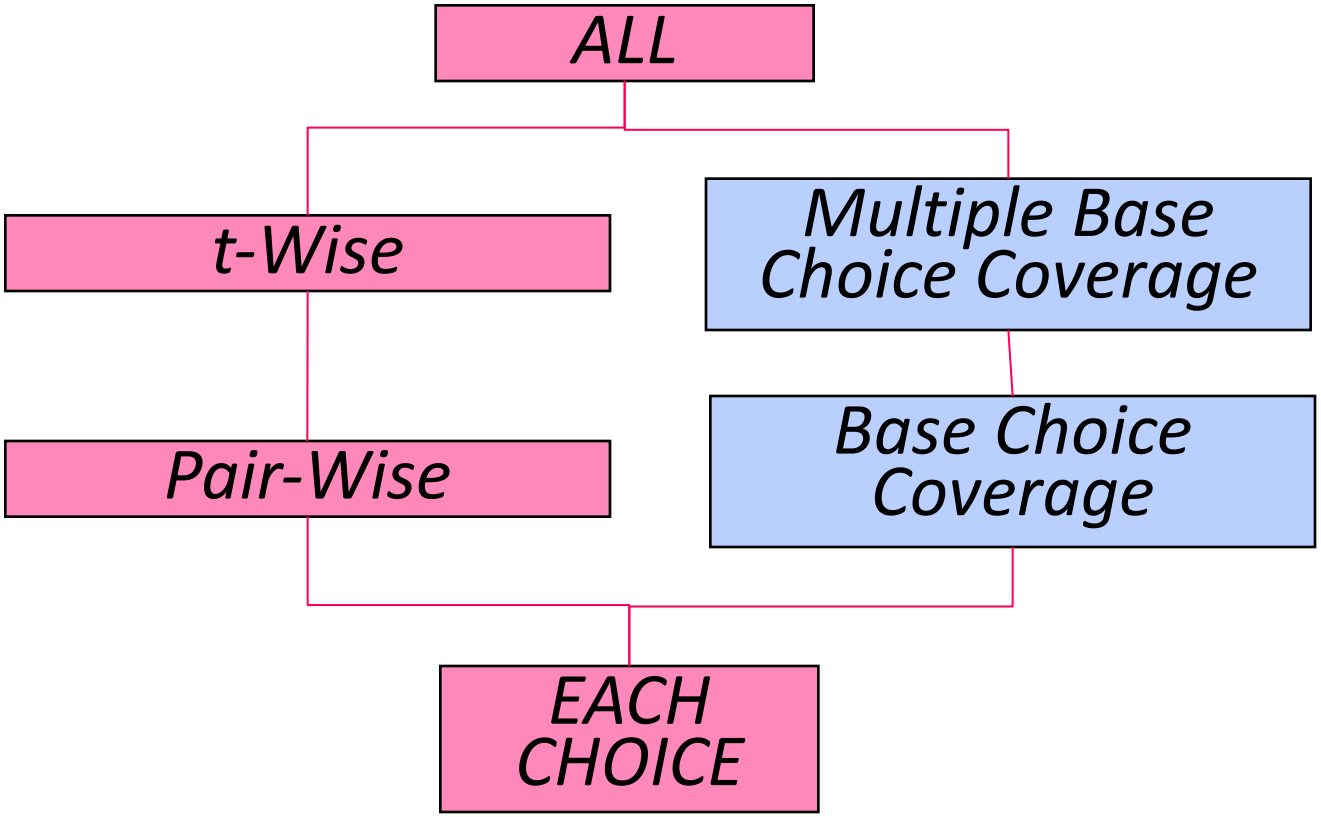
These produce these additional test requirements:

(B,**P,X**)	(**A**,Q,**X**)	(**A,P,**Z)
(C,**P,X**)	(**A**,R,**X**)	
		(**A,P,**Z) duplicate
(B,**P,Y**)	(**A**,Q,**Y**)	
(C,**P,Y**)	(**A**,R,**Y**)	

Some properties to note:

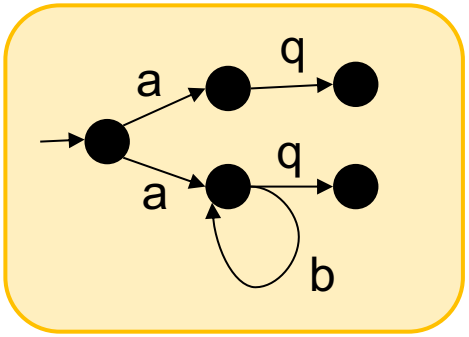
- base-blocks are not cross-combined except as in the base tests.
- non-base blocks are not cross-combined with each other.
- BCC and MBCC will also cover every pair of (base-block,non-base-block).

Overview of partition-based coverage



Models are useful

A simple FSM modelling some program:



- A model helps us in understanding the program it models.
- It can be used as a specification that defines the correctness of the program.
- It provides guidance on how to systematically test the program (e.g. if the model is an FSM, we can try to cover all its prime paths, rather than just randomly trying different actions).

Model based testing (MBT)

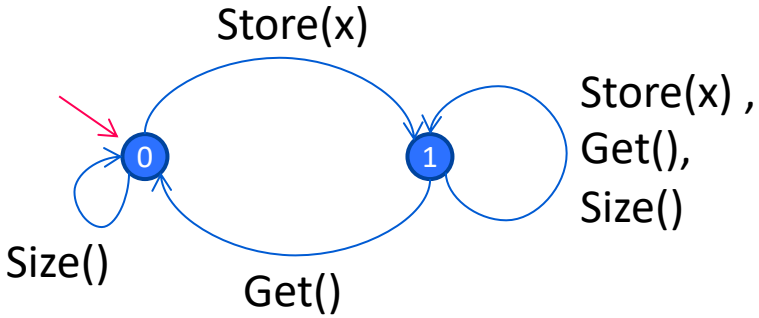
- Model-based Testing (MBT) is a way to test guided by a model:
  - allowing you to define test requirements and coverage measure in terms of the model (e.g cover all transitions in the model)
  - you can even use the model to automatically generate tests
- A popular testing technique. Applications:
  - Testing Web and Mobile applications
  - Testing communication protocols
  - Testing embedded systems
- Typically used in a blackbox setup.

Example: modelling the behavior of a program

```
class ItemStore<T> {
    Store(T  x)
    T   Get()
     int Size()
}
```

- The behavior of a program can be modelled with a finite state machine (FSM) (discussed in 2.5.2 A&O, 7.5.2 2nd ed.). Example in the next slide. Such a model is also called "behavioral model".
- We will also discuss "extended" FSM.

An FSM model of ItemStore

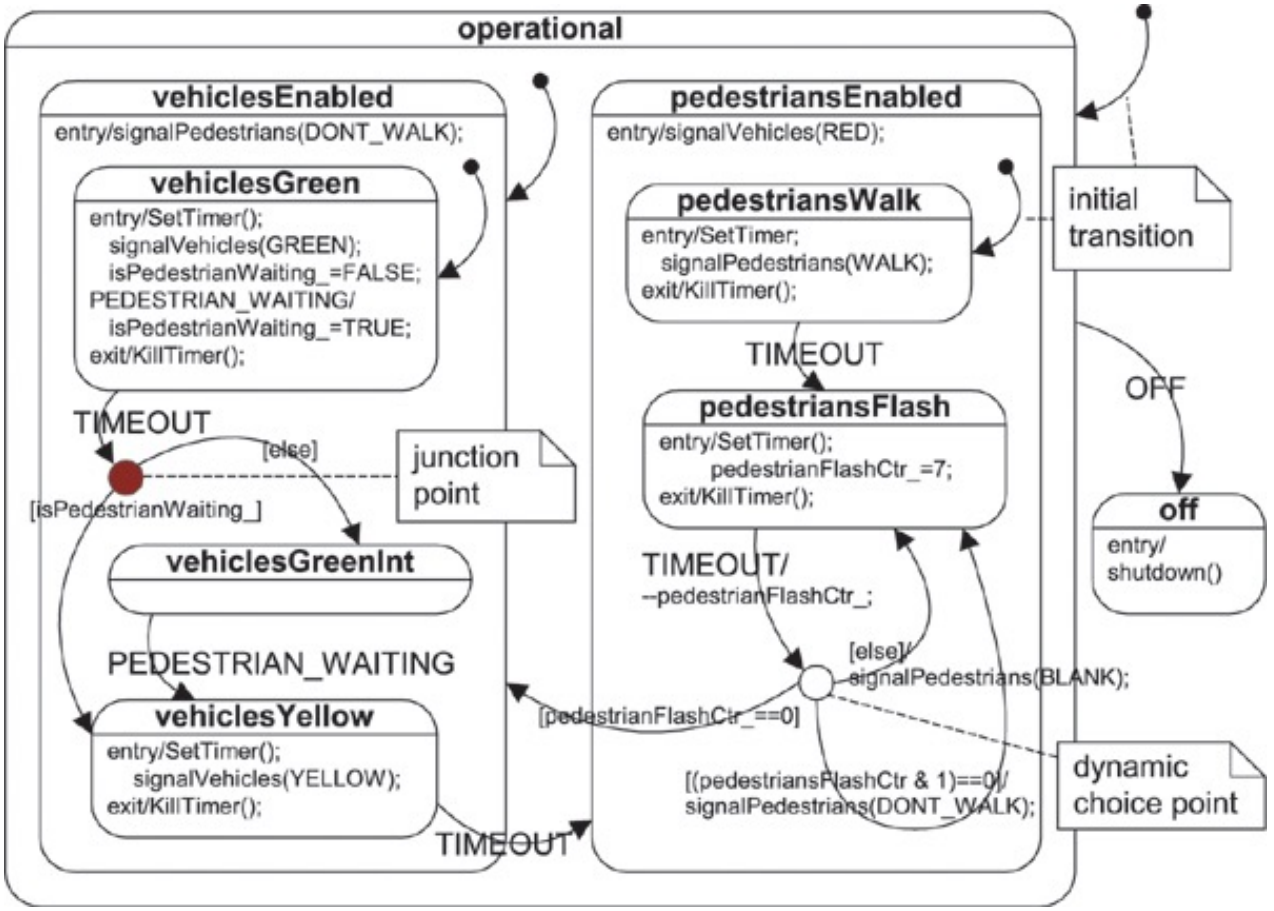


A finite state machine (FSM) is a graph (I,V,E,∑) that abstractly describes a program.

- V is a set of nodes representing the program's states.
- I⊆V is the set of its possible initial states.
- ∑ is a set of "actions" used to label the arrows.
- E is a set of of labelled arrows describing how the program transitions: $u \longrightarrow_a v$ means the FSM can go from the state u to the state v by executing the action a.

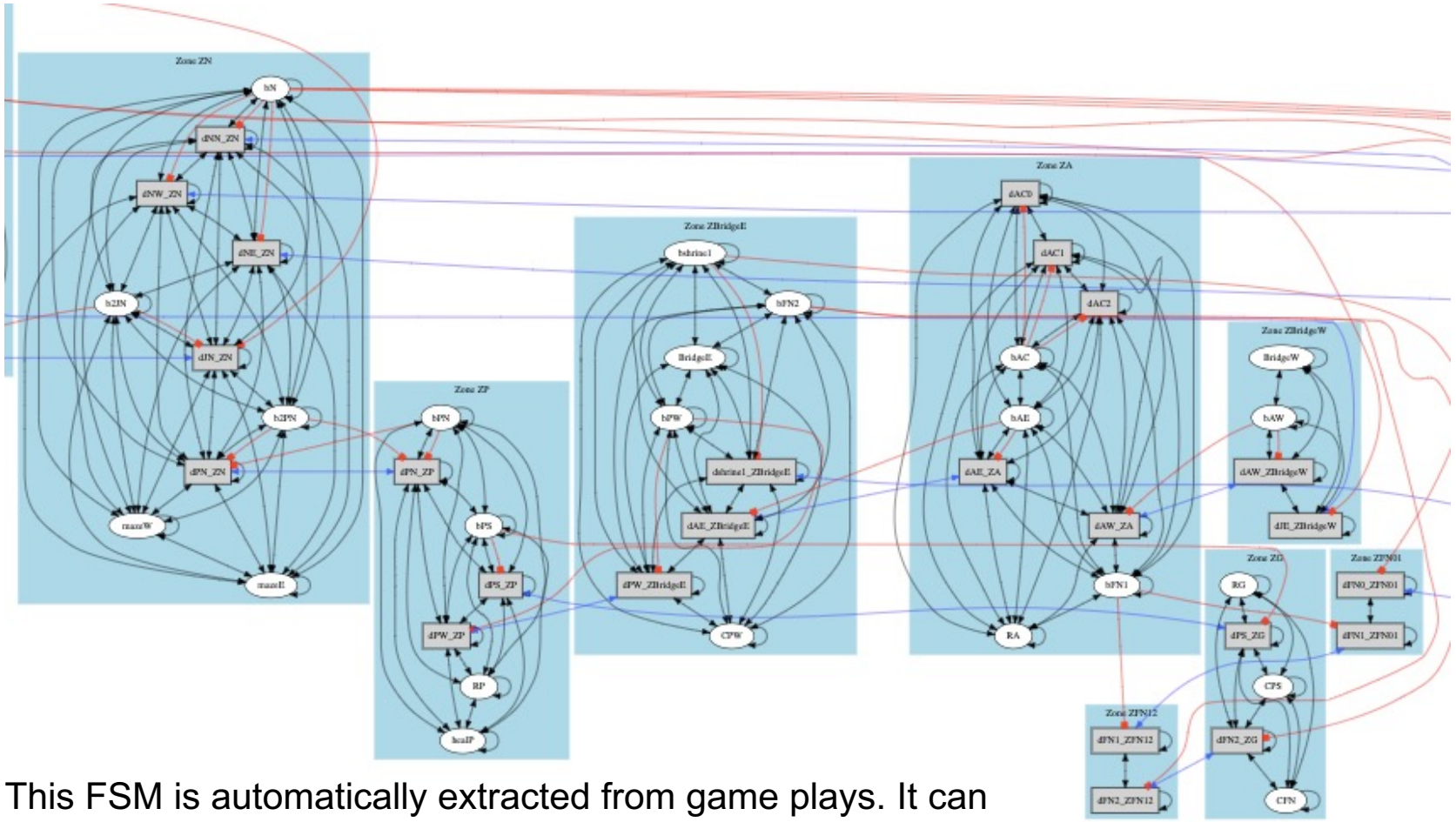
There are various variations of this; e.g. the actions can be parameterized as in Store(x), or we may want to have terminal/exit states.

Hierarchical FSM in UML



UML allows states and transitions to be labelled with additional information, and statues to be be hierarchically formed from another FSM.

FSM “extracted” from a game

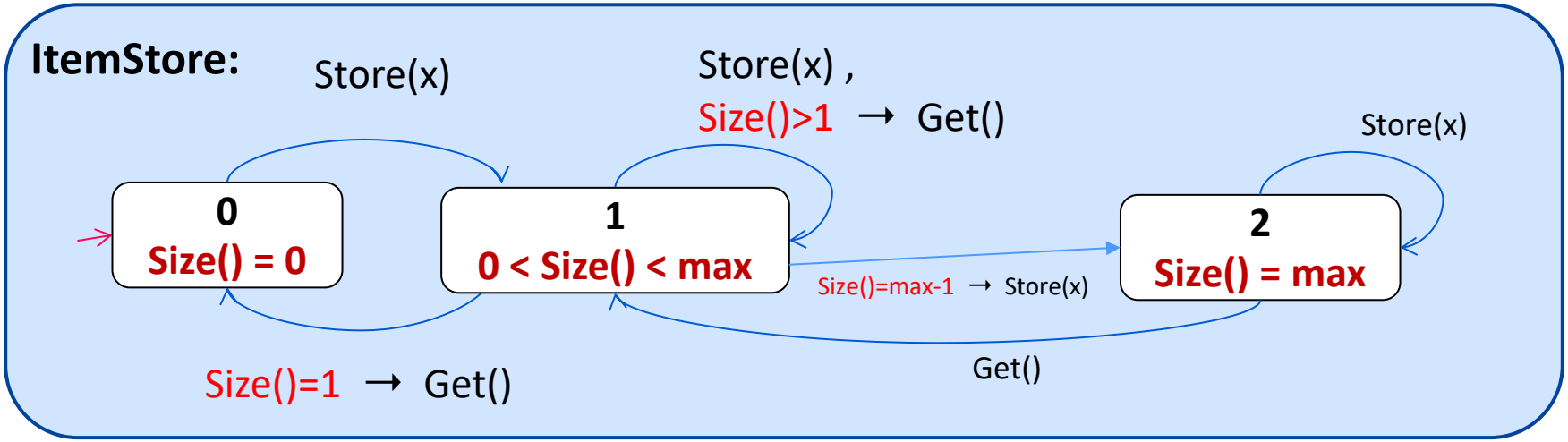


This FSM is automatically extracted from game plays. It can later be used for generating tests.

FSM: drawbacks and alternatives

1. We can't have a model with infinite states in an FSM.
2. FSM also blows up if we have many states, e.g. as in the previos game example. This makes a model hard to comprehend.
3. UML's uses hierarchical FSM to mitigate the problem in (2).
4. Alternatively, we can also use **Extended FSM** (EFSM).

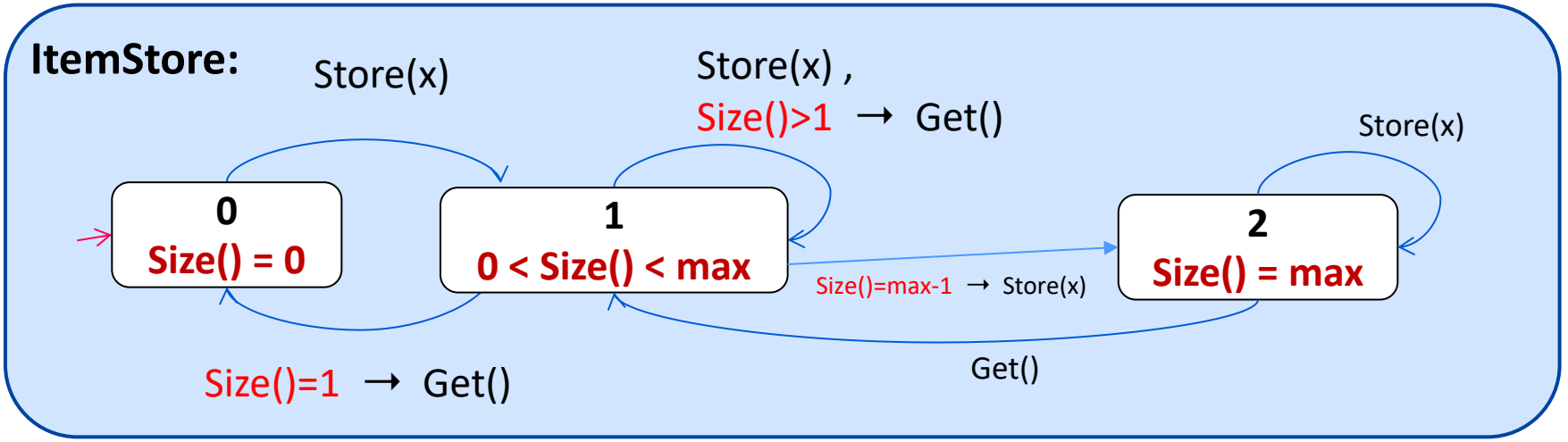
Extended FSM (EFSM)



To provide more a complete specification we use an extended FSM:

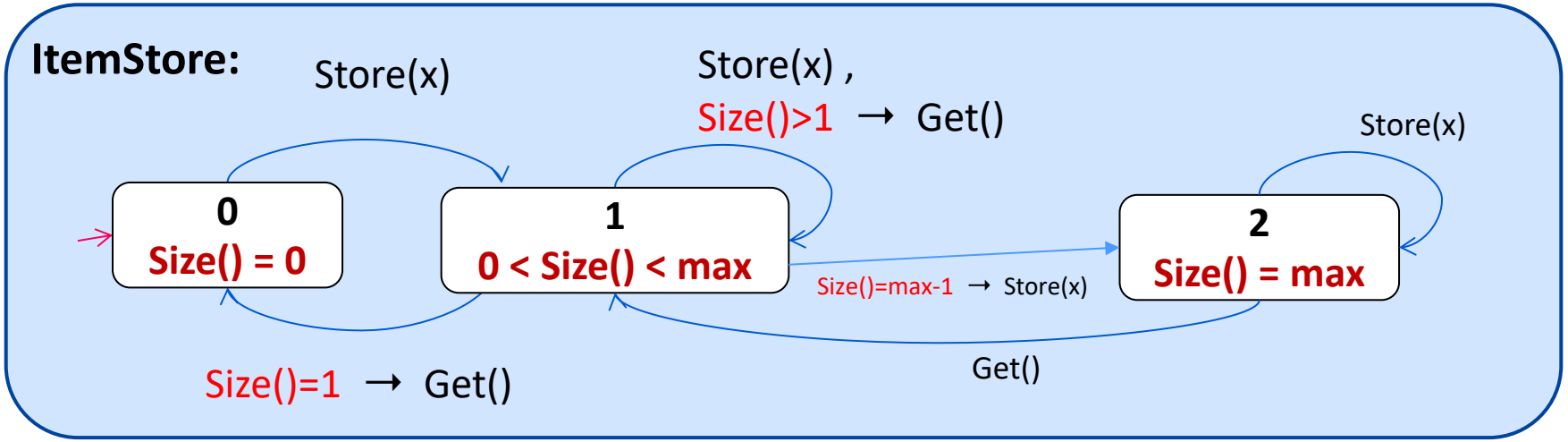
- The FSM has variables or getters V that can be inspected, e.g. Size().
- A transition can now be guarded. The **guard** specifies when the transition can be taken. A guard is a predicate over V e.g. "Size()>1".
- A transition may also specify a **post-condition**, which is a predicate over V that must hold after the transition is taken.
- Every state is labaled with relevant **predicates** over V that are expected to hold there, as in the above example.
- Note that the addition of V can make the FSM to actually have infinite states.

EFSM



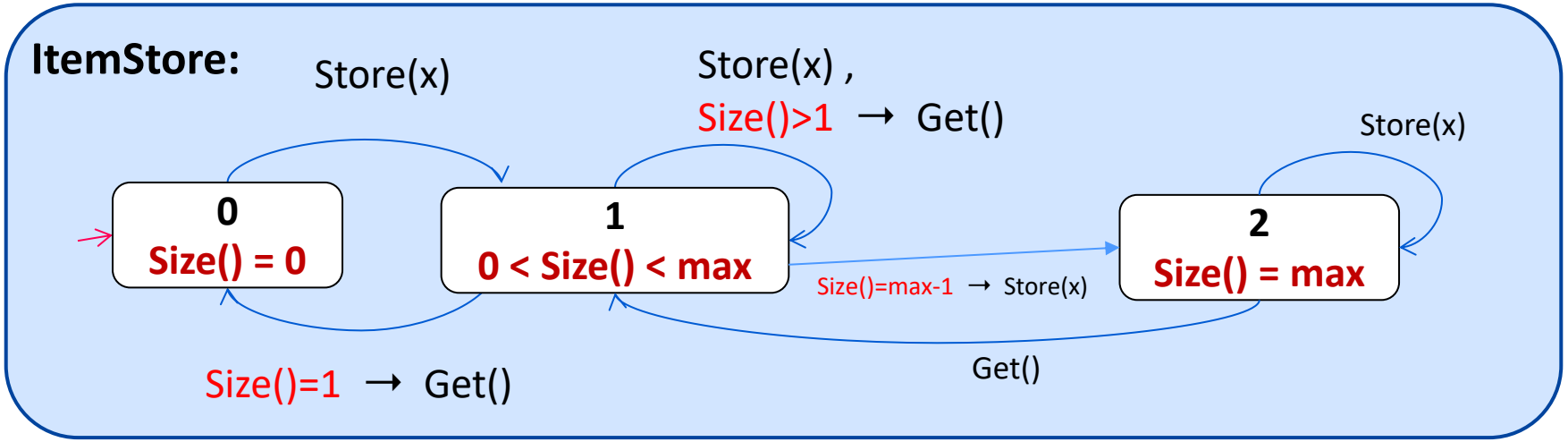
- A sequence of transitions is *admissible* if (1) it starts in the initial state, (2) forms a path in the FSM, and (3) all the guards along the path evaluate to true.
- **The EFSM is correctly implemented by a program F** if for every admissible sequence σ of transitions ending in a state s:
  - all predicates decorating *s* evaluate to true.
  - the post-condition of the last transition in σ is true.
- We should also provide a definition of the expected effect of executing a disallowed action. Here, we define the default effect is to remain in the same state.

Model Based Testing with EFSM



- We treat a model M as the specification of an actual program F.
- Black box: we can only observe the program's state through the vars or getters in V.
- A **test** is a sequence of admissible transitions. The correctness of the implementation F is checked by checking the transitions' post-conditions and the predicates that decorate the states along the sequence.
- You can design the tests manually, or generate them from the model.
- You can e.g. aim for full transition coverage, or even prime path coverage.

Challenge for automated test generation



- To generate a test, the sequence must be admissible. So, all the guards of the transitions along the way must be satisfies. This can be non-trivial.

  E.g. generating a test that reach state-2 is less trivial.

- If store(x) transitions above are also guarded e.g. requiring that x should be contain a valid item-id, generating a test can be hard without knowledge what "valid" id is, and how to generate one.