NO MORE GARBAGE IN:

VALIDATING FORMAL MODELS

HOW TO AVOID MISTAKES

EVEN THE EXPERTS MAKE!

25 May 2023

Pamela Zave

Princeton University

Princeton, New Jersey

Tim Nelson

Brown University

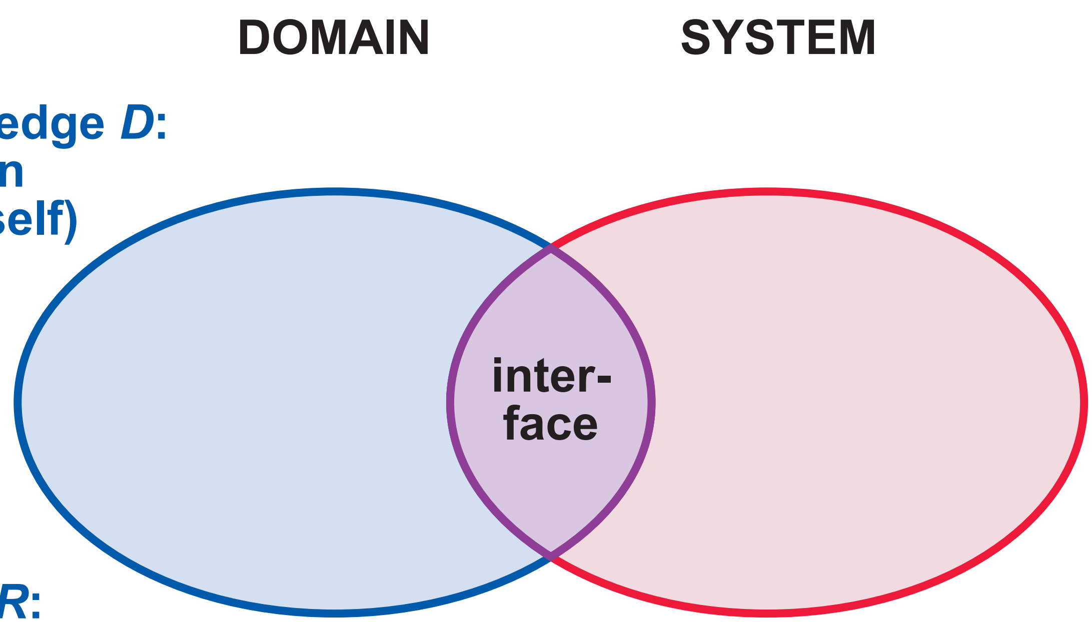
Providence, Rhode Island

A LITTLE HISTORY . . .

This is the Jackson-Zave model (for Michael Jackson and me).

developed in mid-1990s

It is considered to be the foundation of requirements engineering.



Domain Knowledge *D*: how the domain behaves (by itself)

Requirements *R*: how the domain should behave (with the system in place)

Specification *S*: how the system behaves (as observed at its interface)

The Primary Proof Obligations for Validation:

D, *S* are consistent; (*D* and *S*) implies *R*

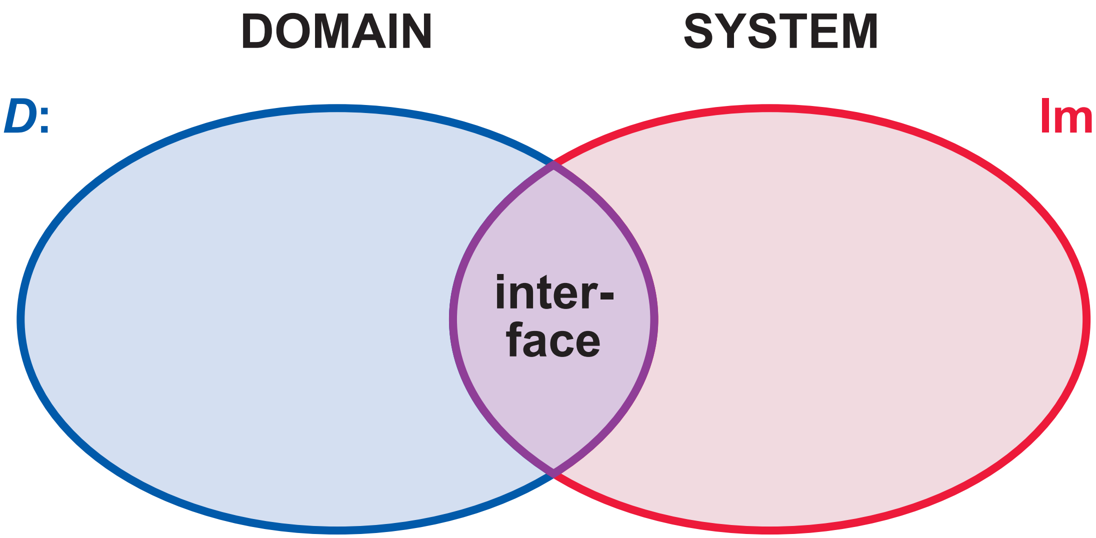
BUT . . .

. . . requirements engineering is part of software engineering, . . .

. . . which is now regarded as quite separate from "formal methods and verification," . . .

. . . so it seems to have been forgotten.

THE CONTENT OF A GOOD FORMAL MODEL



Domain Knowledge *D*: how the domain behaves (by itself)

Implementation *M*: how the system behaves

Requirements *R*: how the domain should behave (with the system in place)

Specification *S*: how the system behaves (as observed at its interface)

note that the phenomena of the interface are shared between the domain and the system!

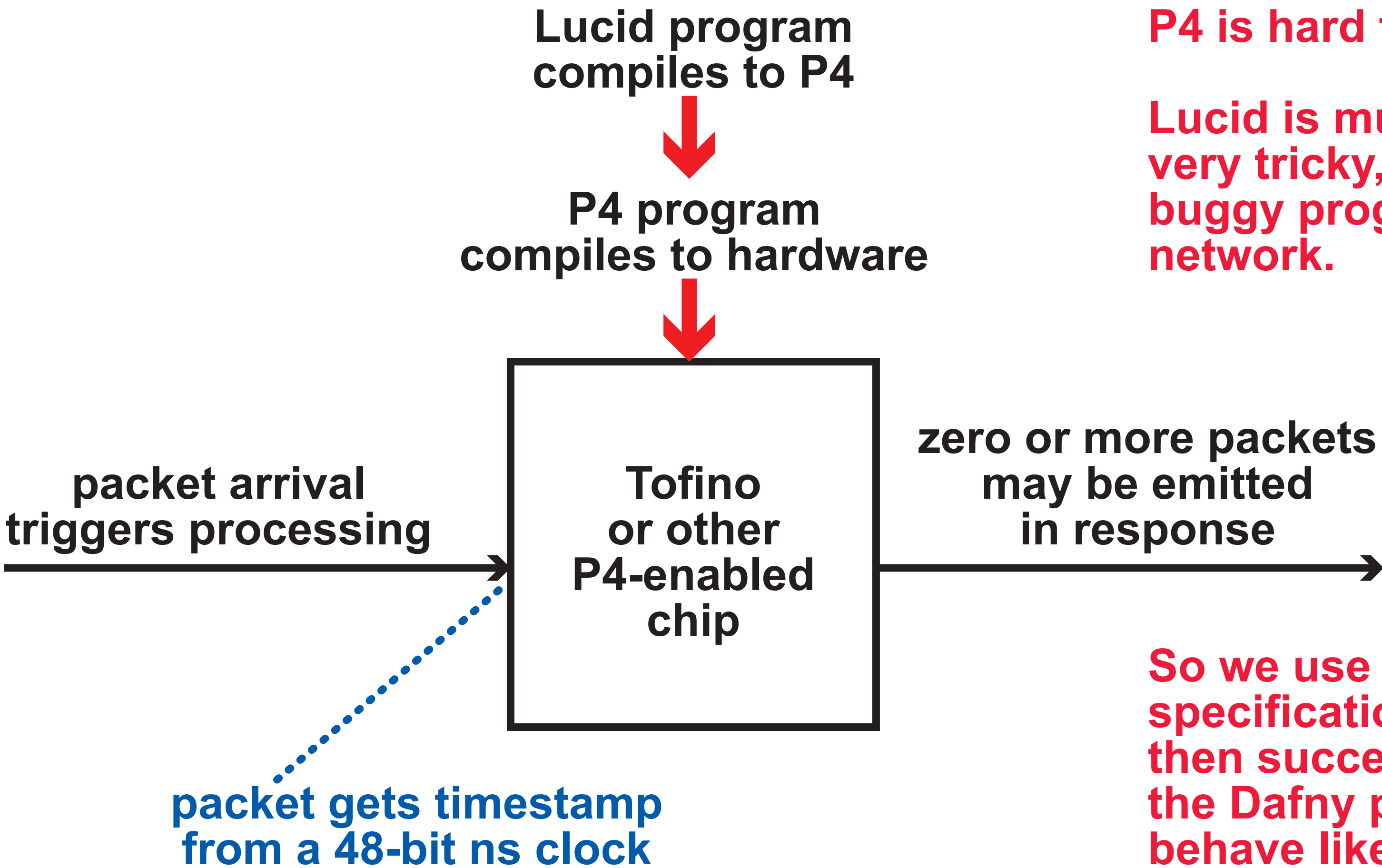
→ **this means that *S* is actually a subset of *R*, and may be all of it**

→ **if there is no *R* apart from *S*, validation is weaker**

THE PRIMARY PROOF OBLIGATIONS

M **implies** ***S***	**verification, where *S* consists of assertions**
D*, *S* consistent; (*D* and *S*) implies *R	**validation, where *S* consists of facts**

PROGRAMMING PACKET-PROCESSING HARDWARE



P4 is hard to use!

Lucid is much better, but still very tricky, and we cannot risk buggy programs deep in a network.

So we use Dafny for specification and validation, then successively transform the Dafny program to look and behave like Lucid . . .

. . . while re-proving each successive version.

Verification of the implementation is compiler verification.

MEASURING WAIT TIME

This is a partial specification in Dafny—just enough for our topic.

```
class PacketProcessing
{  // Parameters
   const Min : nat
   // State Variables
   var lastTime : nat
   var waiting : bool
   var waitBegan : nat


   method turnWaitingOn (time: nat)
      requires ! waiting
      requires stateInvariant (time)
      ensures stateInvariant (time)
   {  waiting := true;
      waitBegan := time;
      lastTime := time;              }


   method turnWaitingOff (time: nat)
      requires waiting
      requires stateInvariant (time)
      ensures stateInvariant (time)
   {  waiting := false;
      lastTime := time;              }


   predicate enoughWait (time: nat)
      requires waiting
   {  time - waitBegan >= Min      }
}
```

minimum waiting time

timestamp of last packet processed

timestamp of packet being processed

precondition of packet arrival requires that time > lastTime, because time must be moving forward

[Is this bit of code also specification?]

called by other methods when they need to know

COMPLICATION: TIMESTAMPS ARE BIT VECTORS

```
class PacketProcessing
{  // Parameters
   const T : nat := 256
   const Min : bits
   // State Variables
   var lastTime : nat
   var waiting : bool
   var waitBegan : nat
   var implWaitBegan : bits

   predicate stateInvariant (time: bits, natTime: nat)
   {  ( implWaitBegan == waitBegan % T )
   && ( waiting ==>
         (    enoughWait (natTime)
         <==> implEnoughWait (time) ) ) }

   method turnWaitingOn (time: bits, natTime: nat)
      requires ! waiting
      requires stateInvariant (time, natTime)
      ensures stateInvariant (time, natTime)
   {  waiting := true;
      waitBegan := natTime;
      implWaitBegan := time;
      lastTime := natTime;          }

   predicate enoughWait (natTime: nat)
      requires waiting
   {  natTime - waitBegan >= Min as nat }
}
```

```
predicate implEnoughWait (time: bits)
requires waiting
{ (time - implWaitBegan) % T >= Min }
```

an 8-bit vector, values 0-255

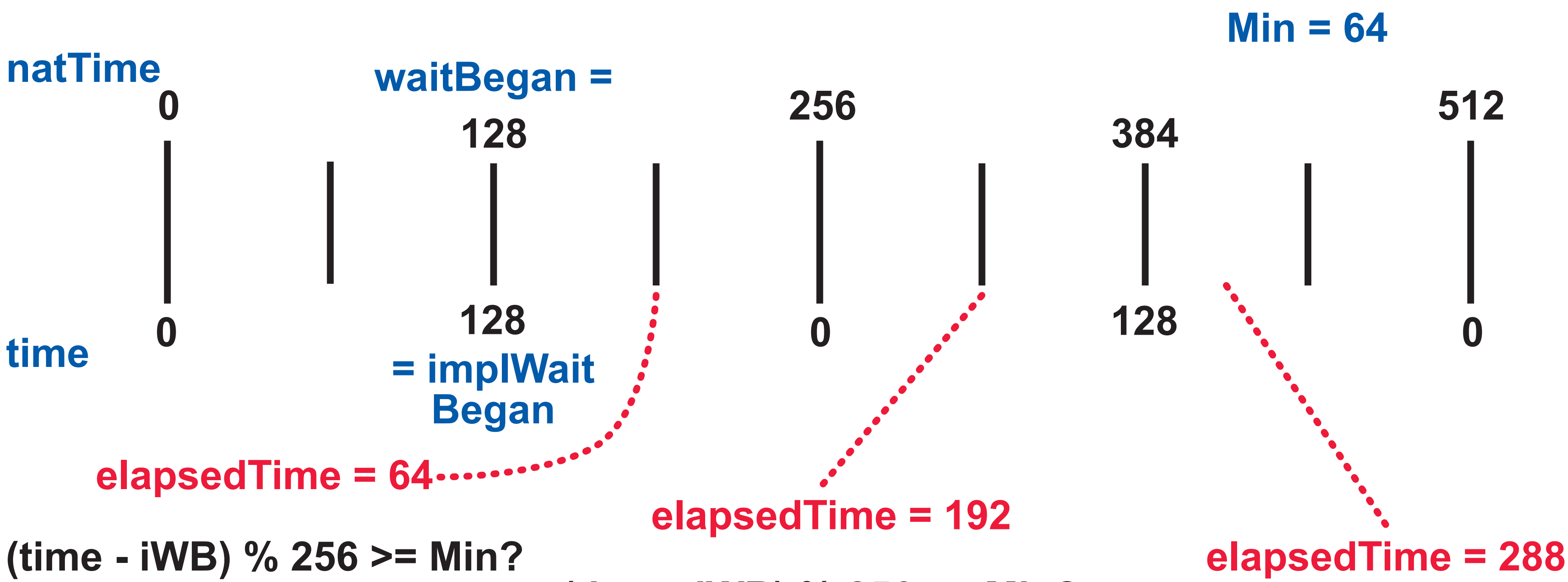
for every packet,
time == natTime % T

verified as invariant, so
the more concrete specification
implements the abstract specification!

abstract specification

more concrete specification

SOMETHING IS BOTHERING ME—
COULD THAT HAVE BEEN JUST A LITTLE TOO EASY?



(time - iWB) % 256 >= Min?

(192 - 128) % 256 >= 64?

64 >= 64?

true

(time - iWB) % 256 >= Min?

(64 - 128) % 256 >= 64?

192 >= 64?

true

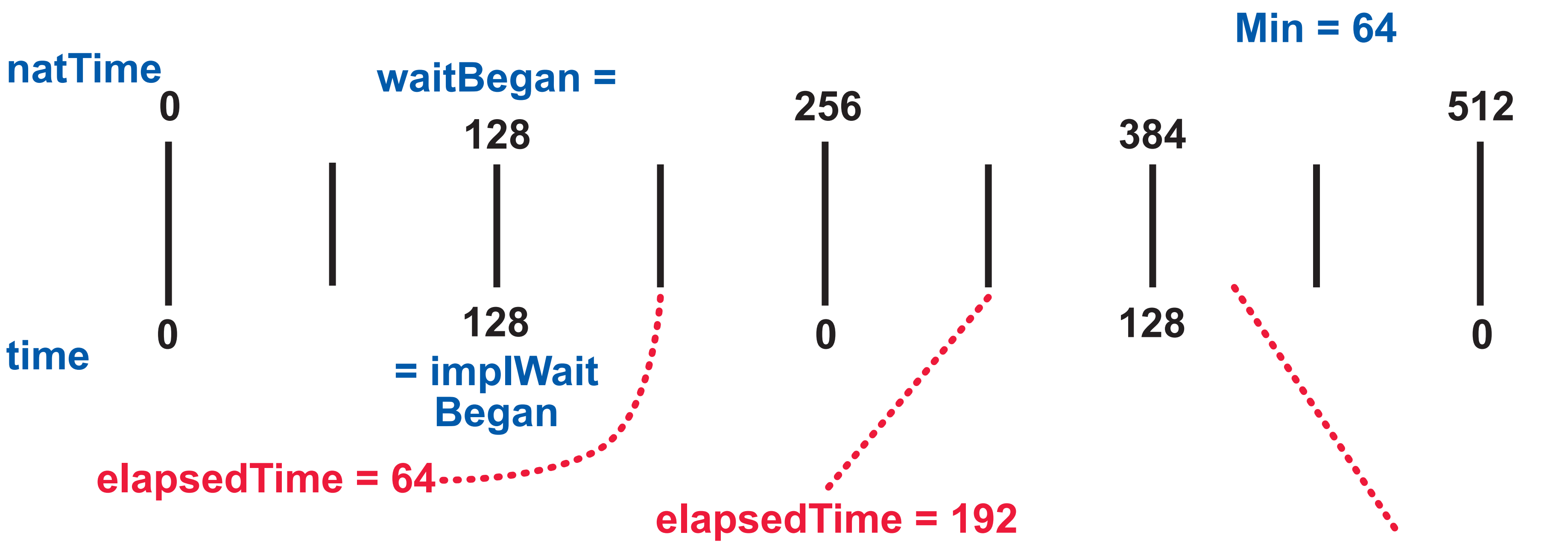
(time - iWB) % 256 >= Min?

(160 - 128) % 256 >= 64?

32 >= 64?

FALSE

SOMETHING IS BOTHERING ME—
COULD THAT HAVE BEEN JUST A LITTLE TOO EASY?



elapsedTime = 288

(time - iWB) % 256 >= Min?

(192 - 128) % 256 >= 64?

64 >= 64?

true

(time - iWB) % 256 >= Min?

(64 - 128) % 256 >= 64?

192 >= 64?

true

(time - iWB) % 256 >= Min?

(160 - 128) % 256 >= 64?

32 >= 64?

FALSE

Now I remember!
To measure time accurately, the
ELAPSED TIME MUST BE LESS THAN 256
(the timestamp rollover period)

Does it matter?

Why is program verified as correct?

TIMESTAMP ROLLOVER MATTERS—
BECAUSE A NANOSECOND IS VERY SHORT!

Packets have 48-bit nanosecond timestamps.

These timestamps roll over in 78 hours or about 3.25 days.

In programming, we save timestamps in 32-bit words.

If a clock tick is still a nanosecond, these timestamps roll over in 4.3 seconds!

We save the high-order 32 bits, which loses resolution but is OK for us.

Bounding wait times by 3 days is acceptable, 4 seconds is not.

HOW CAN THIS INVARIANT BE VIOLATED?

```
class PacketProcessing
{  // Parameters
   const T : nat := 256
   const Min : bits
   // State Variables
   var lastTime : nat
   var waiting : bool
   var waitBegan : nat
   var implWaitBegan : bits

   predicate stateInvariant (time: bits, natTime: nat)
   {  ( implWaitBegan == waitBegan % T )
   && ( waiting ==>
         (    enoughWait (natTime)
         <==> implEnoughWait (time) ) ) }

   method turnWaitingOn (time: bits, natTime: nat)
      requires ! waiting
      requires stateInvariant (time, natTime)
      ensures stateInvariant (time, natTime)
   {  waiting := true;
      waitBegan := natTime;
      implWaitBegan := time;
      lastTime := natTime;         }

   predicate enoughWait (natTime: nat)
      requires waiting
   {  natTime - waitBegan >= Min as nat }
}
```

for every packet,
time == natTime % T

how can this be verified correct,
when we know it is not?

abstract
specification

more concrete
specification

```
predicate implEnoughWait (time: bits)
requires waiting
{ (time - implWaitBegan) % T >= Min }
```

THE INVARIANT IS VIOLATED BY THE PASSAGE OF TIME

**When natTime reaches implWaitBegan + T, the invariant becomes false.
After that, it is true and false intermittently.**

Packet-processing code is effectively instantaneous.

**Therefore nothing models the passage of time,
which occurs in the domain, not in the system.**

**Verification yields garbage because
the model has insufficient domain knowledge.**

```
method clockTick (time: bits, natTime:nat)
   requires time == natTime % T
   requires natTime >= lastTime

   requires waiting => (natTime < waitBegan + T)

   requires waiting => ((natTime + 1) < waitBegan + T)

   requires stateInvariant (time, natTime)
   ensures stateInvariant (time, natTime)

{  var timePlus : bits := (time + 1) % T;
   var natTimePlus : int := natTime + 1;

   assert timePlus == natTimePlus % T;

   assert stateInvariant (timePlus, natTimePlus);
}
```

domain knowledge (annotation on the method signature)

new precondition, for all methods (the one I forgot) (annotation on `requires waiting => (natTime < waitBegan + T)`)

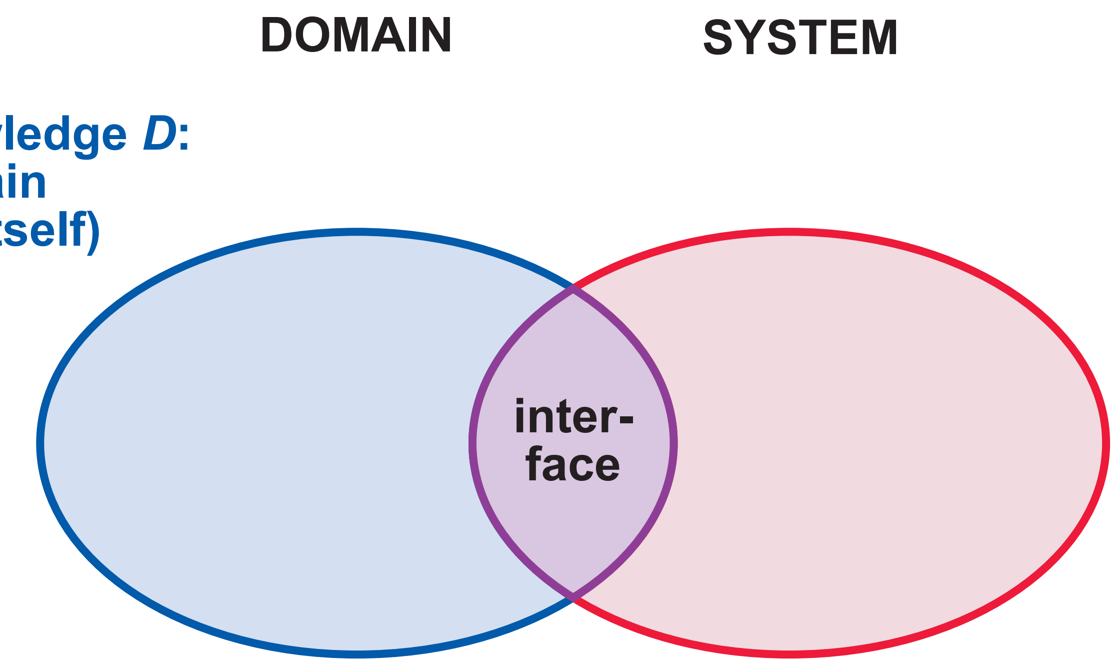
without this special precondition, this method does not preserve invariant (annotation on `requires waiting => ((natTime + 1) < waitBegan + T)`)

SOMETIMES IT IS TRICKY TO FIND THE DOMAIN . . .

. . . BUT THAT DOES NOT MAKE IT ANY LESS IMPORTANT!

For example, when the system is a small addition to an enormous base of hardware, software, and human practices.

Domain Knowledge *D*: how the domain behaves (by itself)



Requirements *R*: how the domain should behave (with the system in place)

Specification *S*: how the system behaves (as observed at its interface)

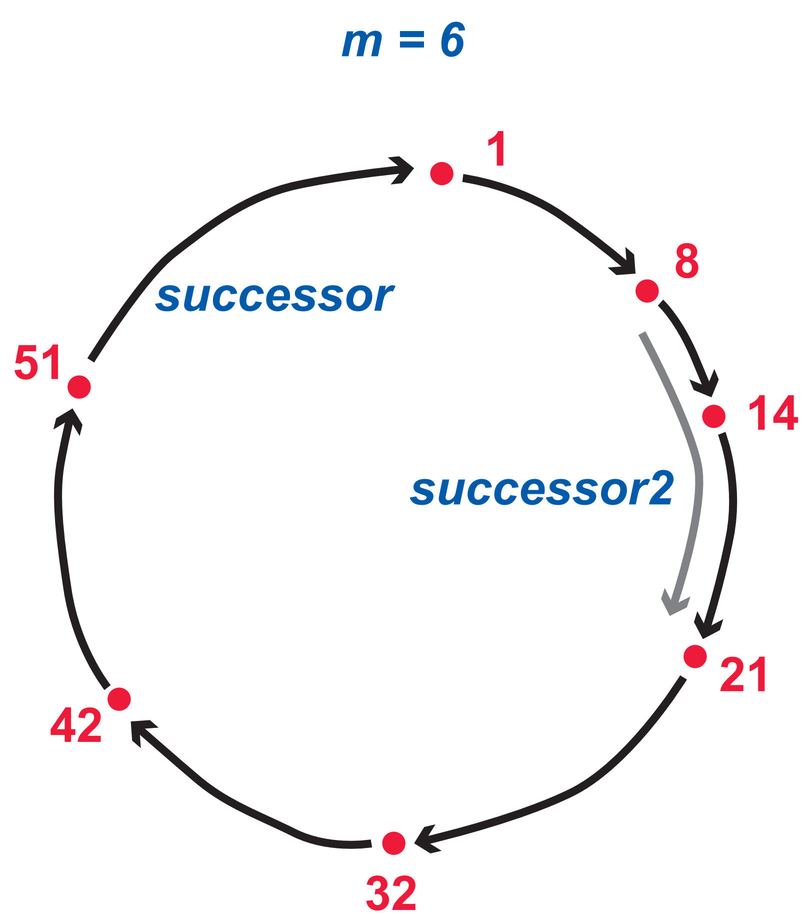
THE CHORD PROTOCOL MAINTAINS A PEER-TO-PEER NETWORK

identifier of a node (assumed unique) is an m-bit hash of its IP address

nodes are arranged in a ring, each node having a successor pointer to the next node (in integer order with wraparound at 0)

redundant pointers (extra successors) support fault-tolerance

the protocol preserves the ring structure as nodes join, leave silently, or fail



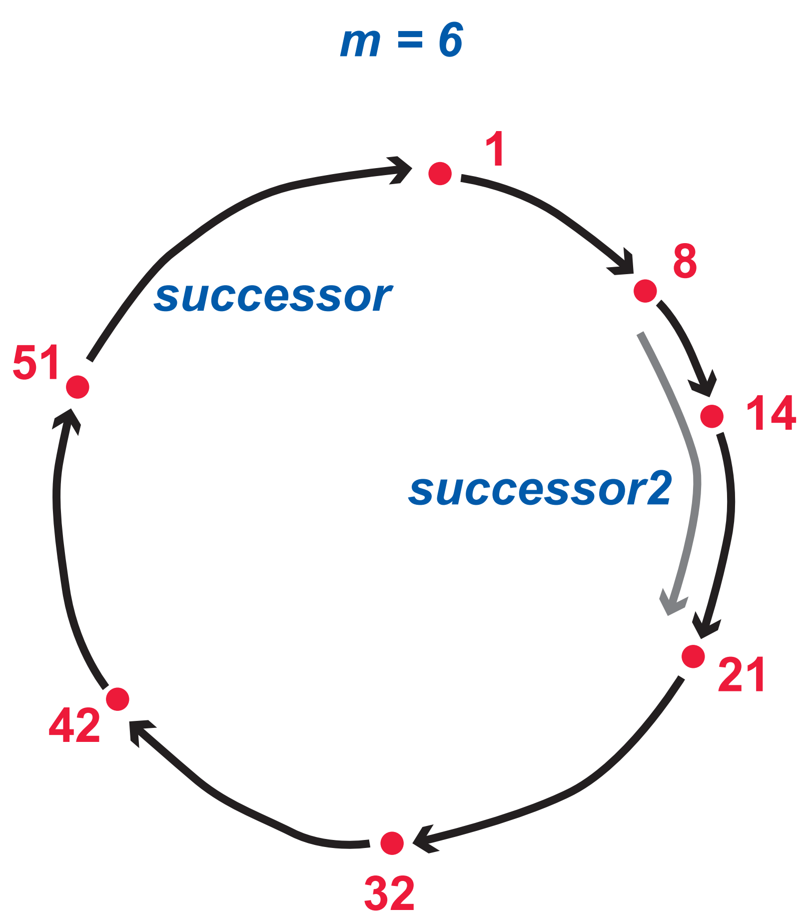
THE CHORD PROTOCOL MAINTAINS A PEER-TO-PEER NETWORK

identifier of a node (assumed unique) is an m-bit hash of its IP address

nodes are arranged in a ring, each node having a successor pointer to the next node (in integer order with wraparound at 0)

redundant pointers (extra successors) support fault-tolerance

the protocol preserves the ring structure as nodes join, leave silently, or fail



THE PROTOCOL IS INTERESTING!

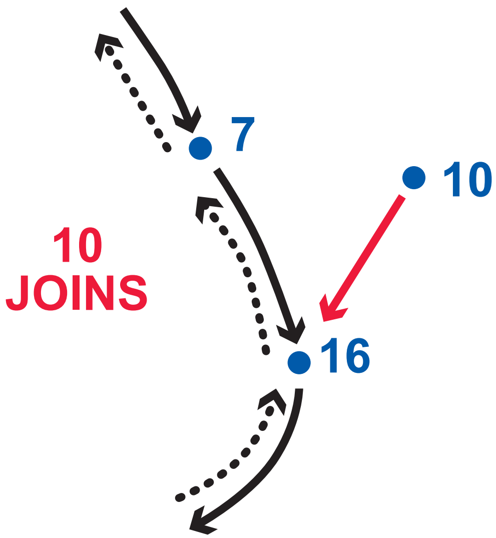
- no central administration (almost)
- communication in the network is fast because of chords
- protocol operations are simple and fast

  no multi-node atomic operations

- protocol is highly fault-tolerant

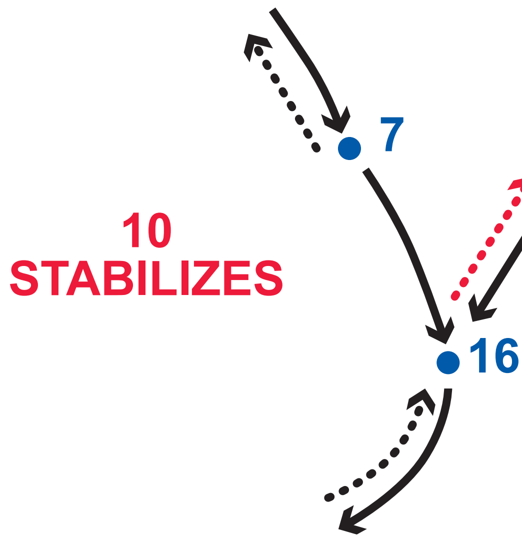
OPERATIONS OF THE PROTOCOL (SIMPLIFIED)

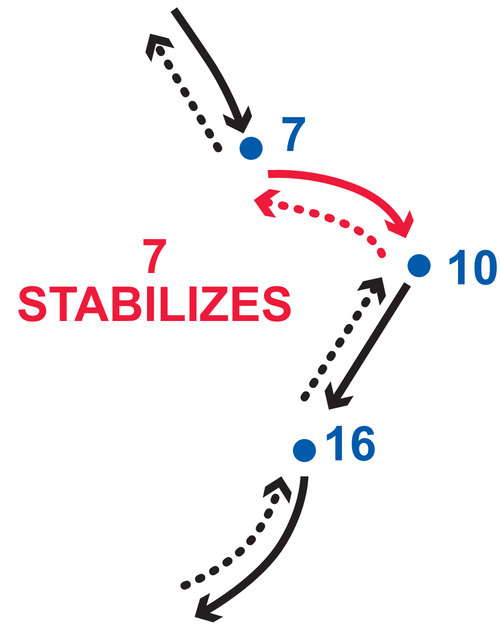
A new member can Join at any time.



A member can Fail (or leave) silently.

Each member Stabilizes periodically.
This updates its successor list.





If a Stabilizing member contacts its first successor and gets no answer, then the successor is presumed dead and the member promotes its second (and other) successors.

WHY IS CHORD IMPORTANT?

the 2001 SIGCOMM paper introducing Chord is one of the most-referenced papers in computer science, . . .

. . . and won SIGCOMM's 2011 Test of Time Award

APPLICATIONS

- allows millions of *ad hoc* peers to cooperate
- used as a building block in fault-tolerant applications
- often used to build distributed key-value stores (where the key space is the same as the Chord identifier space)
- the best-known application is BitTorrent

RESEARCH ON PROPERTIES AND EXTENSIONS

- protection against malicious peers
- key consistency (all nodes agree on which node owns which key), replicated data consistency

"Three features that distinguish Chord from many other peer-to-peer lookup protocols are . . .

. . . its simplicity,
. . . provable correctness,
. . . and provable performance."

THE CLAIMS

Correctness Property:

In any execution state, IF there are no subsequent Join or Fail events, . . .

. . . THEN eventually . . .

. . . all pointers in the network will be globally correct, and remain so.

THE REALITY

- even with simple bugs fixed and optimistic assumptions about atomicity, the original protocol is not correct
- of the seven properties claimed invariant of the original version, not one is actually an invariant

not surprisingly, due to sloppy informal specification and proof

I found these problems by analyzing a small Alloy model

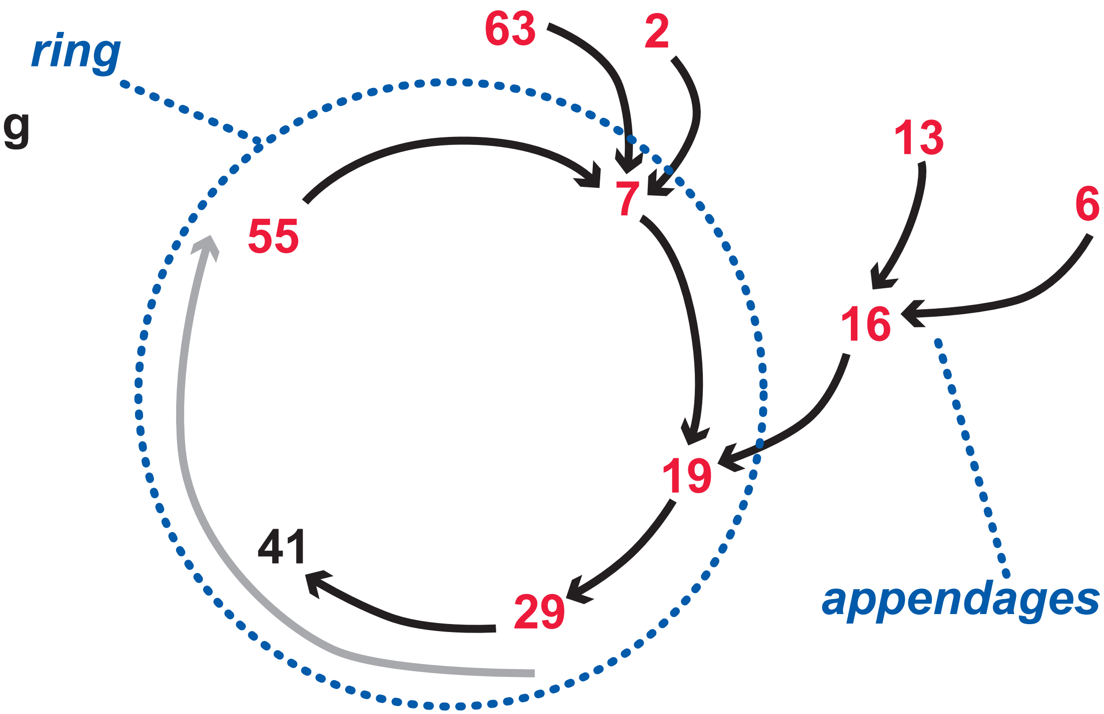
Chris Newcombe and others at Amazon credit this work with overcoming their bias against formal methods, which they now use to find bugs.

[CACM, April 2015]

WHAT DO WE KNOW?

NODE OPERATIONS

- nodes can Join or Fail (including leave silently) at any time
- this leads to appendages outside the ring
- each node will Stabilize periodically, making repairs



PARAMETERS OF THE PROTOCOL

- the length of successor lists L
- the frequency of stabilization F

REQUIREMENT (Eventual Consistency)

In any execution state, IF there are no subsequent Join or Fail events, . . .

. . . THEN eventually . . .

. . . all pointers in the network will be globally correct, and remain so.

no appendages

WHAT DO WE NEED?

A PROOF THAT: (*D* and *S*) implies *R*

A NEW SPECIFICATION
OF THE NODE OPERATIONS
THAT WORKS

A PROOF!

Getting this was very hard, because . . .

. . . there was no known invariant

. . . the true invariant (when found) looks nothing like the expected invariant properties

A FORMAL MODEL OF
REALISTIC DOMAIN KNOWLEDGE
THAT MAKES THE PROOF POSSIBLE

but this is not today's story!

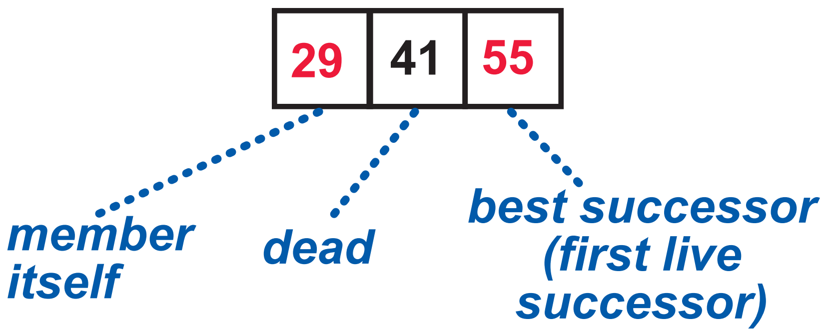
This operating assumption has always been used:

No failure leaves a member without a live successor.

this is simple and convenient, but nothing justifies it

ACHIEVING CORRECTNESS

extended successor list (ESL)
of 29 (with L = 2):



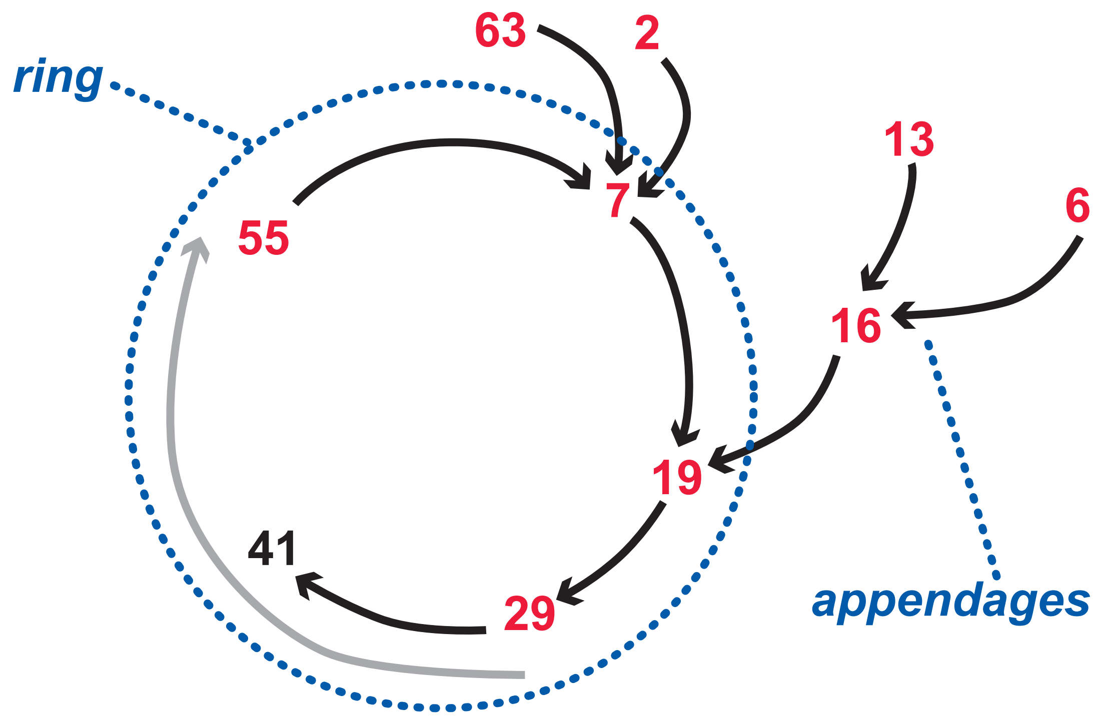
so the extra successor is providing
a backup in case of failure . . .

. . . but the original protocol is sloppy about successor lists, allows empty and duplicated entries:

29	41	29

. . . which means that the redundancy in the data structure is being thrown away, . . .

. . . and the so-called operating assumption says that 41 cannot fail!



MANY CAREFUL CHANGES ENSURE THAT:

every ESL has L + 1 distinct entries, each having been a member at one time (if not now)

TOWARD A REALISTIC DOMAIN MODEL

TRULY REALISTIC

For each use of Chord . . .

. . . there is a probability distribution for inter-Join gaps

. . . there is a probability distribution for inter-Fail gaps

. . . there is L (length of successor lists)

. . . there is F (frequency of stabilization at each node)

HOWEVER, . . .

- I don't know how to get such information.
- Even if I had it, I wouldn't know how to use it in a proof.
- Even if I knew how to use it in a proof, I could not do a proof for each use of Chord.

TOWARD A REALISTIC DOMAIN MODEL

TRULY REALISTIC

For each use of Chord . . .

. . . there is a probability distribution for inter-Join gaps

. . . there is a probability distribution for inter-Fail gaps

. . . there is L (length of successor lists)

. . . there is F (frequency of stabilization at each node)

HOWEVER, . . .

- I don't know how to get such information.
- Even if I had it, I wouldn't know how to use it in a proof.
- Even if I knew how to use it in a proof, I could not do a proof for each use of Chord.

A REASONABLE APPROACH

Use the specification guaranteeing that every ESL has L + 1 distinct entries.

good: the full potential for fault-tolerant redundancy is being used

bad: requires that every network have at least L + 1 nodes (which is 4-6 among thousands or millions)

Domain model: With very high probability (approximately always), failure never leaves a member without a live successor.

ASSUMPTION IS JUSTIFIED BECAUSE . . .

If the operating assumption is not satisfied in real operation, the administrator can increase L or decrease F, which will solve the problem.

real justification for the assumption!

HOW EVEN THE EXPERTS CAN GET IT WRONG

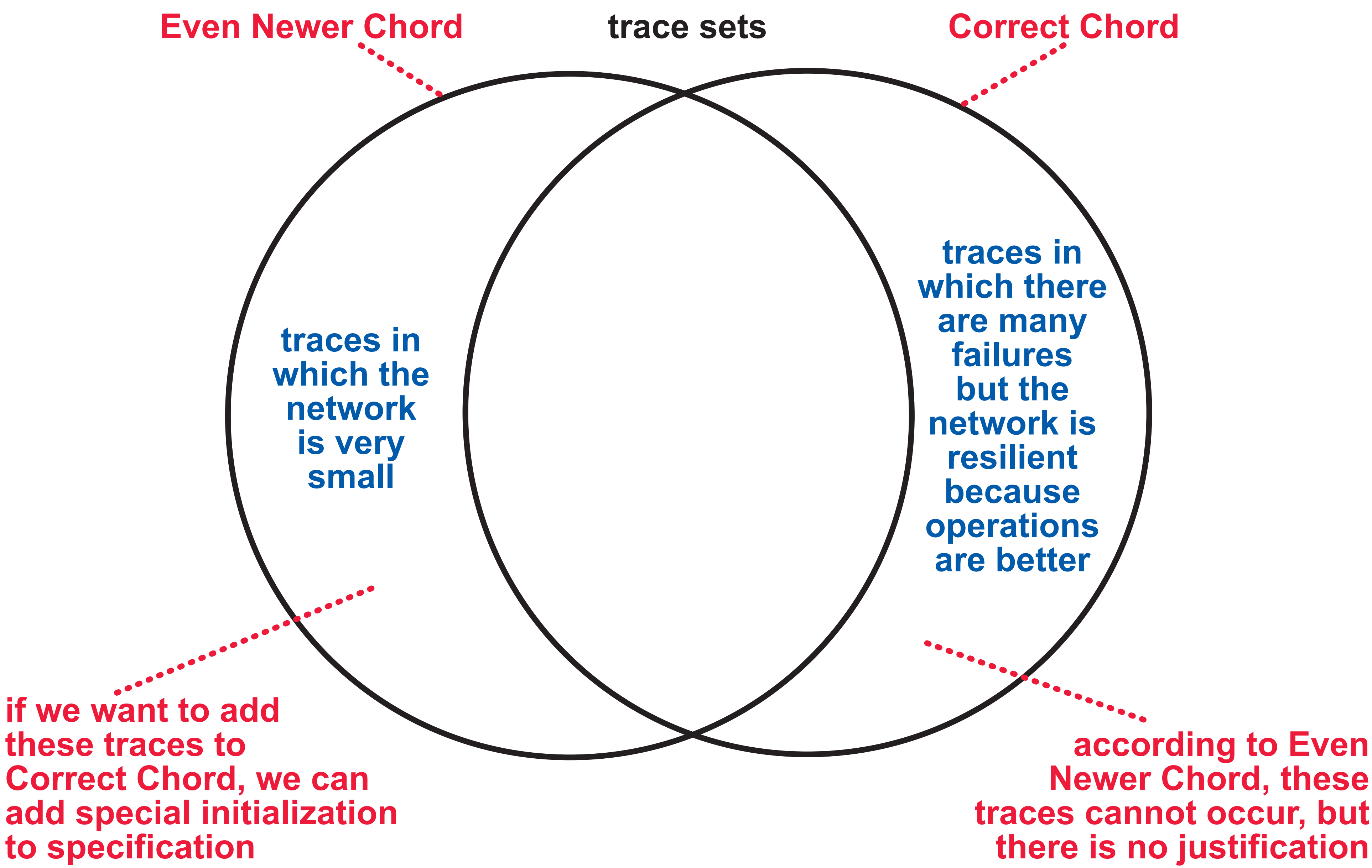
IN FORMAL METHODS

SUBSEQUENT RESEARCH

- uses a specification with some of my changes but not others
- as a result, specification does not maintain the property that each ESL has L + 1 distinct entries
- as a result, specification does not require a minimum number of nodes
- researchers claim: we have a better result!

WHICH IS BETTER?

A trace is determined by a sequence of Joins and Fails in the domain, to which the system responds according to its specification.



SUMMARY OF THIS SHORT COURSE

There will be pain.

You can learn from your own pain, or someone else's pain.

Without validation, formal models and the results of verifying them can be garbage.

Think of your formal model as a domain model—relevant to a family of systems—no matter how specific your goals really are.

generalize whatever you can

think about the domain knowledge and requirements as well as the specification

Predicates are great for validation.

many predicates, many instances, and the instances are focused and meaningful

instances can be compared to the real world being described

predicates help you think of better specifications

Even the experts make mistakes—and when they do, it is almost always due to faulty domain knowledge.

domain knowledge does not have to be large, just appropriate

OTHER TOPICS FOR INQUIRING MINDS

- How can tools support validation better?
- How important are scopes in Alloy, and how do you choose them?
- ChatGPT
-
-