

Proving Cloud controls execution over a compliance period

We propose an approach to produce **formal, unsupervised proofs** that custom Cloud controls operate completely over a specific certification period. It is expected to **ease and accelerate the yearly certification processes** of Cloud customers' deployments (eg: SOC2).

The solution leverages a technology called Satisfiability Modulo Theories (SMT), it has been battle hardened by companies like AWS and Azure for supporting an ever growing part of their internal security assurance frameworks.

SMT proofs are state-of-the art artifacts that deliver undisputable evidence to certification auditors, regulators and compliances authorities in an automated way.

Our approach supports **a wide range of IT Security and Compliance controls** seamlessly:

- event-driven or scheduled controls execution
- bursting strategies (scaling many shortlived controls in parallel during activity peaks)
- healing and recovery strategies (retries, restores)
- duplicated events
- overlapping control runs, long-running control runs, ...

For the customer, impact on her existing code base is minimum since it depends on a small amount of highly reusable (aka control independent) code.

Time to disrupt our mindset!

From « running controls »...

... to « **proving** control objectives are **met** »

This is what:

- * regulators*
- * compliance departments*
- * CISOs*
- * SaaS customers*
- * supply chain providers*

expect

Problem statement

Control objective: capture **ALL** anomalies over **complete** period $A \Rightarrow Z$

\neq

Ground implementation 1 (« vertical »):
capture **SOME** anomalies over **complete** period $A \Rightarrow Z$

Ground implementation 2 (« horizontal »):
capture **ALL** anomalies in **arbitrary** intervals between A and Z

Proposed solution:

Leverage Cloud automation and SMT solvers
to prove objectives are met

For vertical implementation: prove that, in **each** control run:
(**ALL** anomalies) $\wedge \neg$ (**SOME** anomalies) is UNSAT.

For horizontal implementation:

- * Replace all **arbitrary** intervals by **certified** ones
- * Prove that, across **all** control runs:

$\bigcup_{\substack{A < \text{control} \\ \text{run} < B}} (\text{certified intervals}) \neq [A, B]$ is UNSAT.

Proposed solution (continued)

*Proving vertical, in-control logic is code dependent and **out of our scope**.
This, however, is a mature subject with plentiful literature.*

For vertical implementation: prove that, in **each** control run:
(ALL anomalies) $\wedge \neg$ (SOME anomalies) is UNSAT.

- ⇒ We will focus on horizontal implementation and provide:
- two full classical solutions
 - a hypothetical hybrid quantum computing draft

Anatomy of a control

Let's scope exactly what we want to achieve by examining common control features.

I – Payload

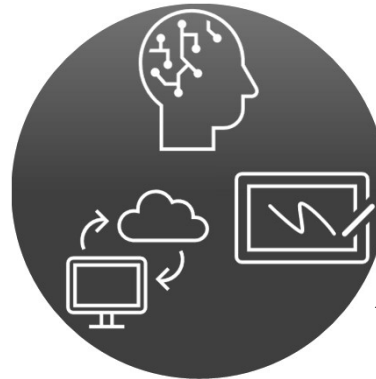
II – Telemetry

III – Posture: detect or prevent?

IV – Execution timeline & datapoints

V - Recap

I – Payload



A notional control

1 Business logic



- Data ingestion & transformation
- Processing window calculation
- Anomaly detection logic

2 Reporting



- Emits proven KPIs

3 Proven KPIs



- First processed event
- Last processed event
- Status code calculation

What happens « inside » the business logic can be either provably accurate (it captures 100% of positives within window), or inaccurate (it may miss positives).

The KPIs, however, must be provable. The 3 calculations are language dependent but easy to implement and highly reusable. They are **out of scope** of this document.

II - Telemetry



The sole purpose of telemetry is to feed the proof with metrics which are themselves 100% proven.

What could go wrong?

Status code

Control fails to launch

N/A (no datapoint)

Crash during execution

N/A (no datapoint)

Cannot load data

KO

Not enough time to process all data within processing window

KO

Some required data are outside of the processing window (eg: long-running process terminating in the future)

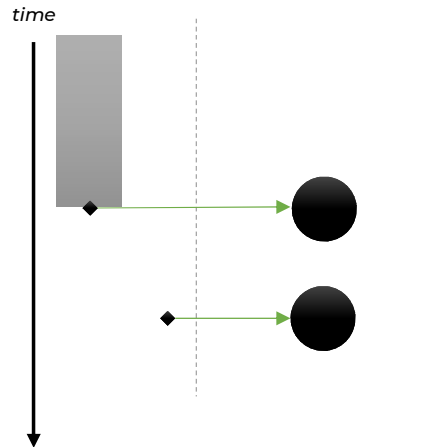
OK

Nothing's wrong

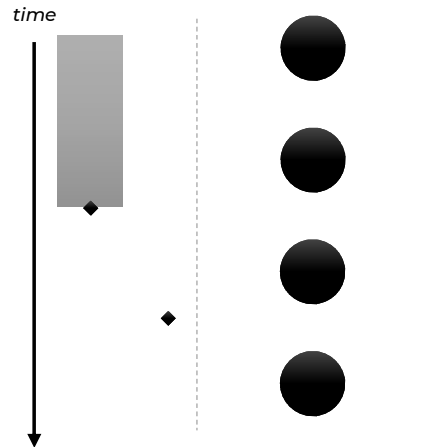
OK

III - Posture: Detective or preventive?

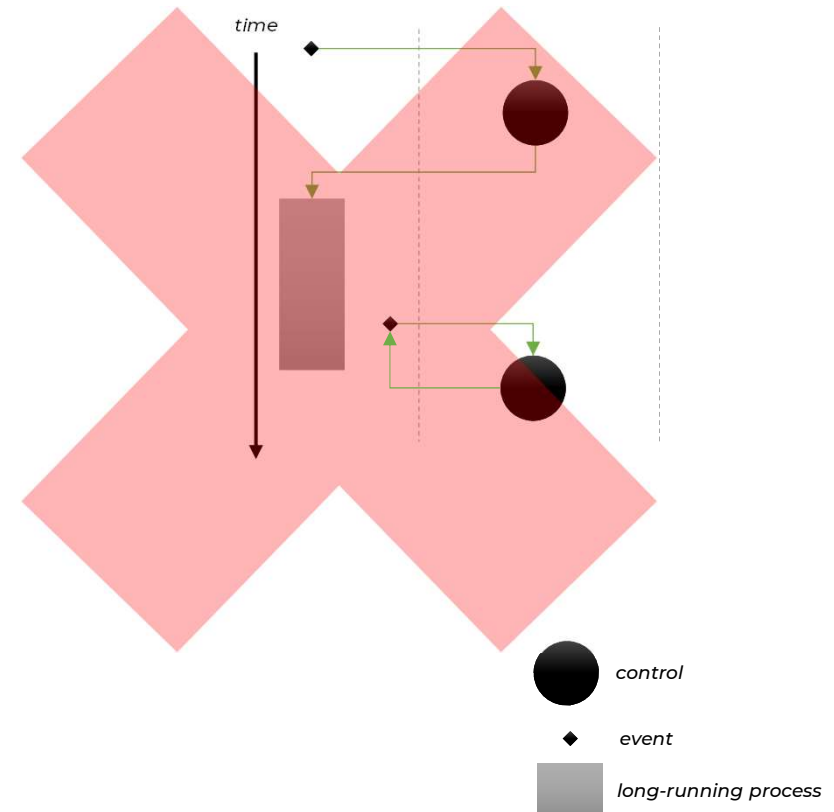
1 Detective & event-driven




2 Detective & scheduled



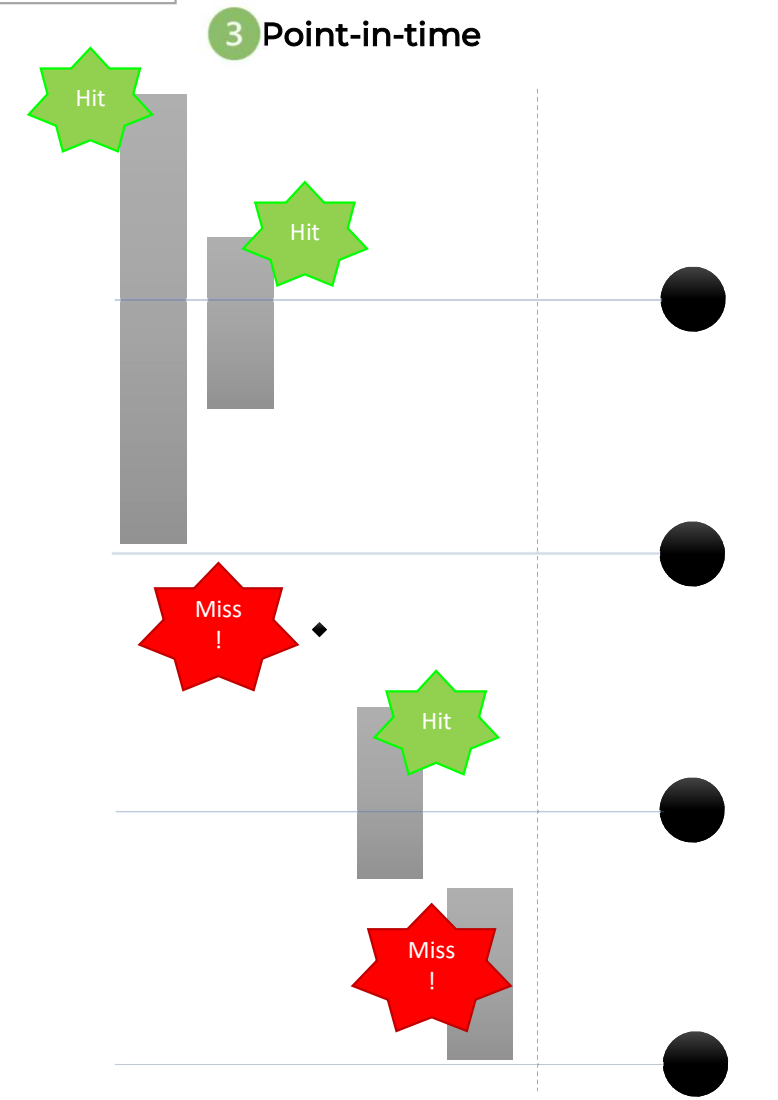
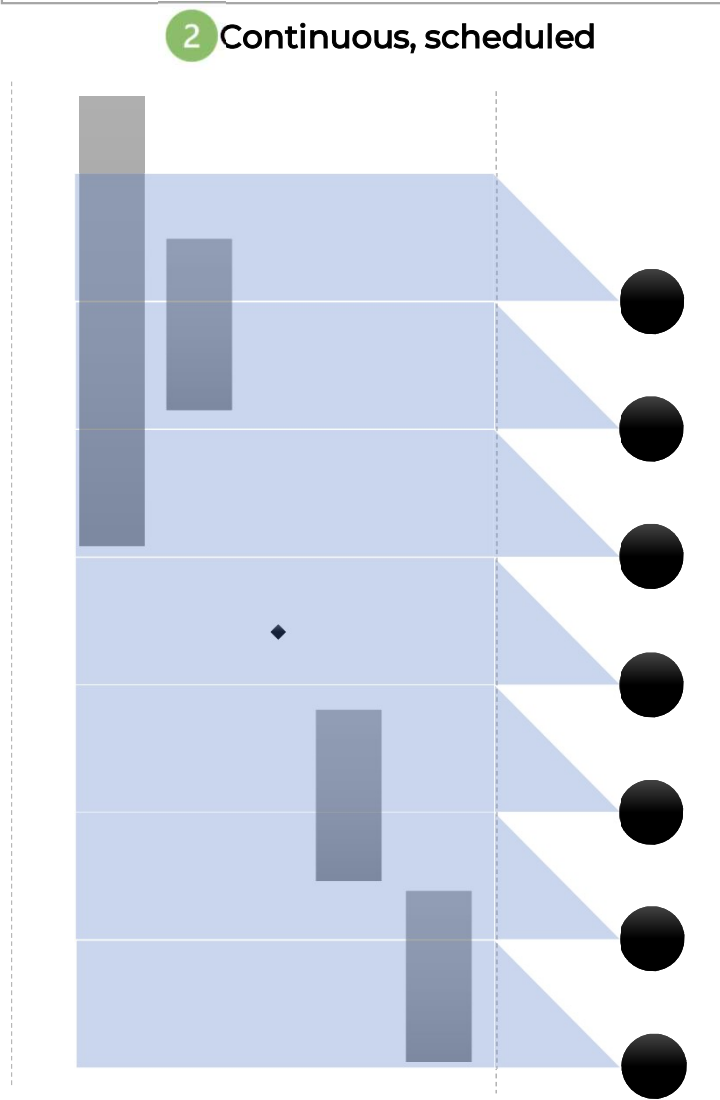
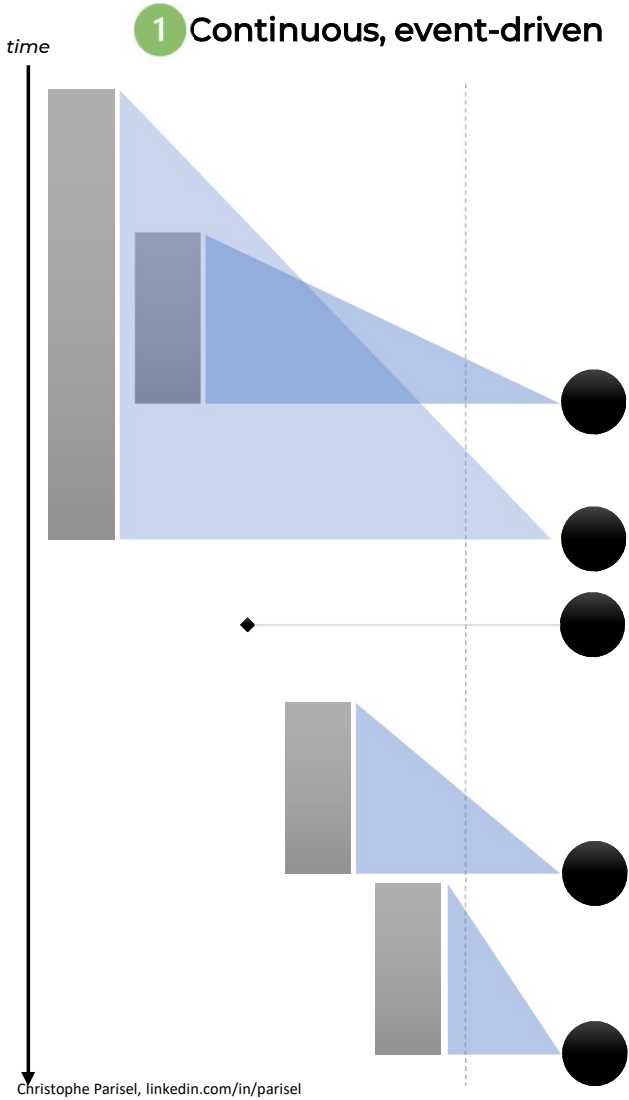
3 Preventive



Most of the time, **preventive** controls are easier to handle: they are **out of scope**.

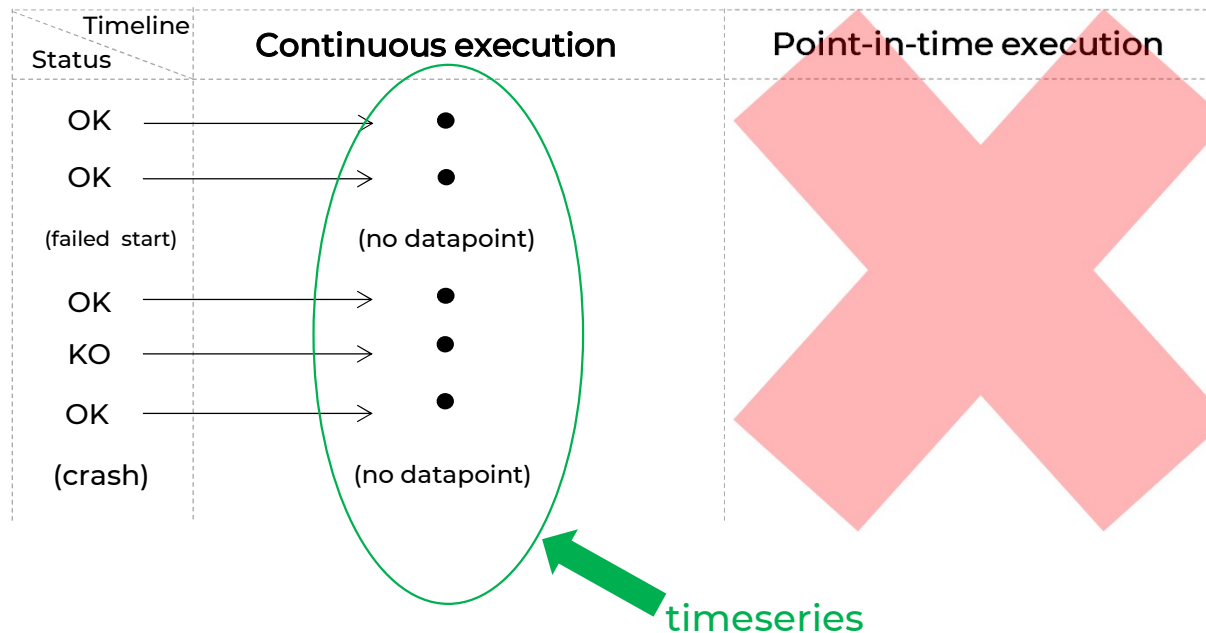
 control's visibility window

IV - Execution timeline: Continuous or point-in-time?



IV – Execution timeline (continued):

Datapoints = status (or lack thereof) \otimes time stamps = timeseries



Point-in-time controls incur built-in event « misses » (see previous slide), so it doesn't make sense to prove they capture all anomalies. They are **out of our scope**.

V – Recap

We are interested in:

- detective controls
- either event-driven, or scheduled
- which are not point-in-time
- which process events within a time interval included into the interval [A,B]
- which capture anomalies with arbitrary accuracy in that interval (proving 100% accuracy is beyond the scope of this document)
- able to emit 3 metrics: time of first processed event, time of last processed event, status code
- able to prove the accuracy of the 3 metrics (by using low footprint, reusable code)

Based on the above, we propose two designs that prove all events are processed by the control from A to B, so the control run is complete over that period

Control design

*We propose two SMT proofs of completeness::
One based on **equality theory** (section II), the other one based on **bitVectors**
(section III)*

But first things first, section I addresses common pre-requisites.

- I – Intervals certification**
- II – SMT proof with equality theory**
- III – SMT proof with bitVectors theory**
- IV – Dealing with compliance gaps**
- V – Hybrid quantum computing draft**

I – Intervals certification: why datapoint intervals are bad

In period [A,B], we have seen that datapoints make up a timeseries of « random » intervals:



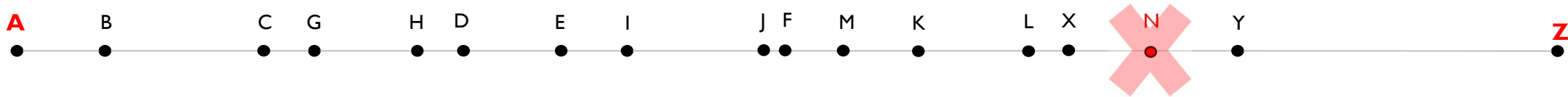
These intervals give us no assurance that events were **completely captured**, and no reasons to believe that controls were actually **execured in that order**.

But the metrics born in each datapoint do.

I – Intervals certification: datapoint metrics

Each datapoint bears a **status code**: if the status is not OK, we can confidently remove this datapoint from the timeseries.

In the timeline below, datapoint « N » has a KO status. We **remove N from the series**



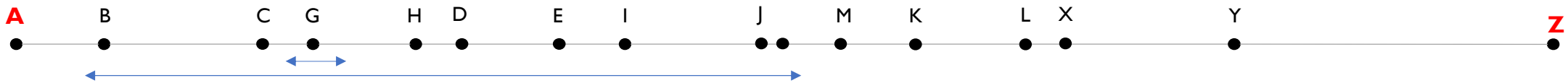
I – Intervals certification:

From datapoints to certified intervals

Each datapoint also bears the time of the **first** and **last** processed event: all events in range have been processed by the control. *This is the ground truth we need to rebuild the thread of control runs in an orderly fashion.*

Say that G processed a tiny amount of events in a short amount of time: G has a very small range: 

Conversely, if B processed a huge amount of scattered events, B has a broad range: 



By virtue of B's range, chances are that B and G ranges overlap. Conversely, if, say, C's range is moderate, it's likely that G and C don't overlap, because G's range is so small.

If we can patch the processing ranges of all datapoints together from A to Z, we will have a formal proof of completeness

The validity of the proof relies on the fact that the times of first and last processed events were proved as valid, so the ranges themselves are also formally true.

We call such ranges **certified intervals**.

II – SMT proof with equality

Graph creation rules

Since **certified intervals** provide a proper ordering of intervals which encompass all events processed by the control over period $[A,B]$, we can attempt to **form a graph** by stitching them together.

To that end, we follow two very simple rules:

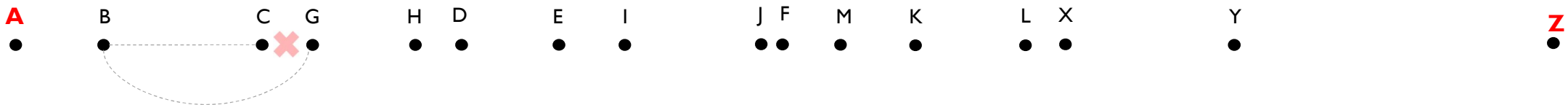
Vertex creation rule: for all datapoints in the timeseries, we create a vertex

Edge creation rule: if the certified intervals of two vertices overlap, we draw an edge

Let's apply this rule set to the (proved) knowledge we have acquired from B, C and G in our timeseries:

- 1) First, we create 16 vertices (one for each datapoint from A to Z)
- 2) Then, we link B and G together
- 3) Suppose C is within B's reach, then we link B and C together

Note: even if C and G look like close neighbors, we don't draw an edge between them



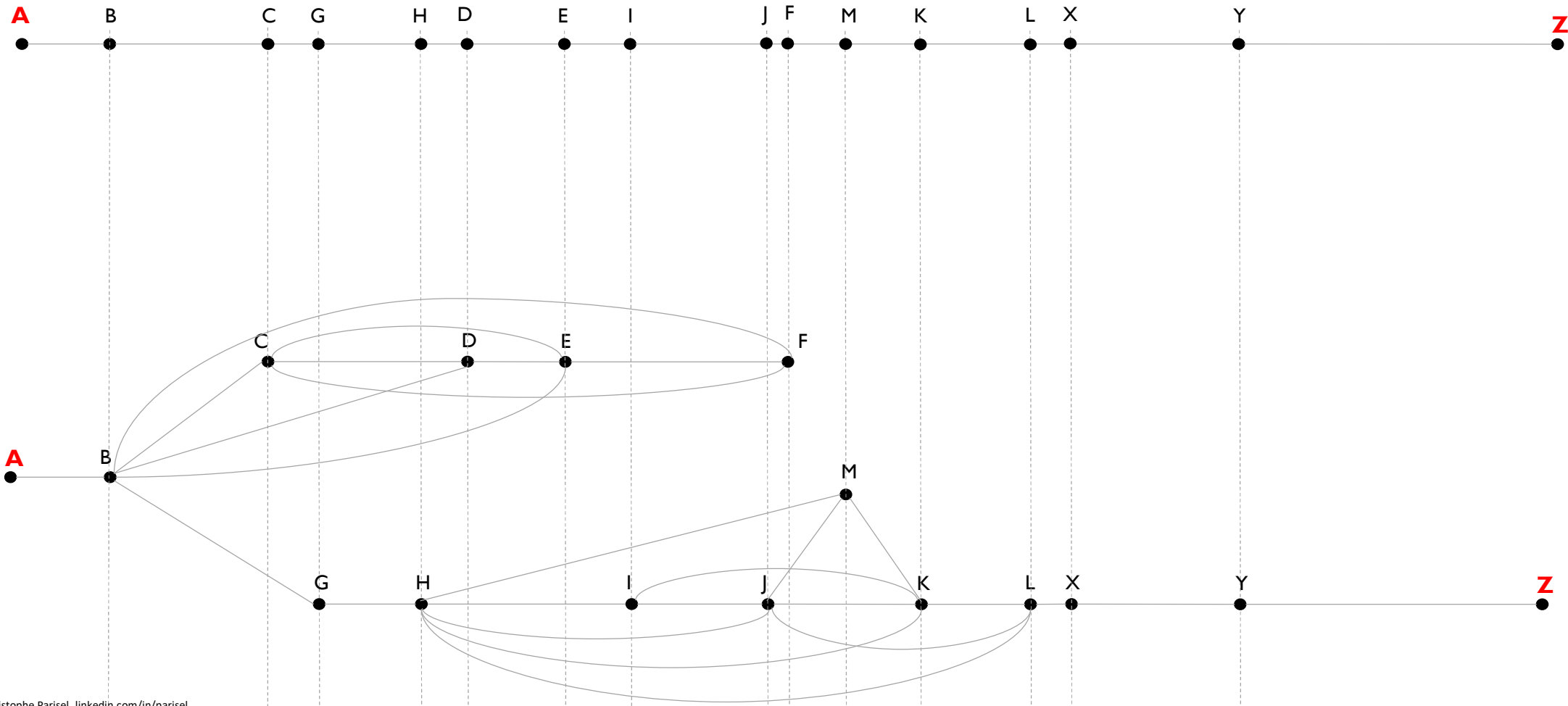
We carry on until we have reviewed all certified intervals.

A possible resulting graph is displayed **on the next slide**

II – SMT proof with equality

sample graph

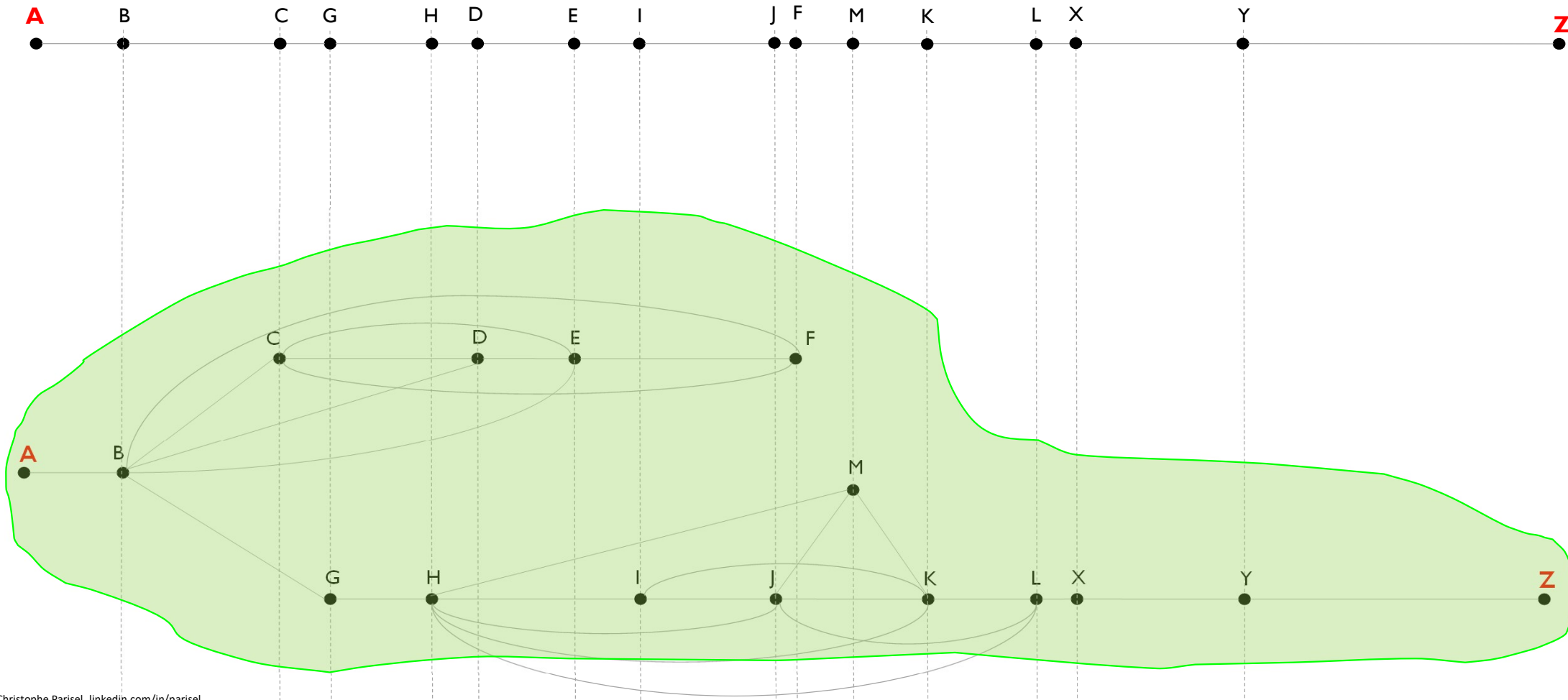
There are no compliance gaps between A and Z because the graph is connected.



II – SMT proof with equality

Single equivalence class = no gaps

Put it another way: all datapoints belong to the same **equivalence class** (in green, below).

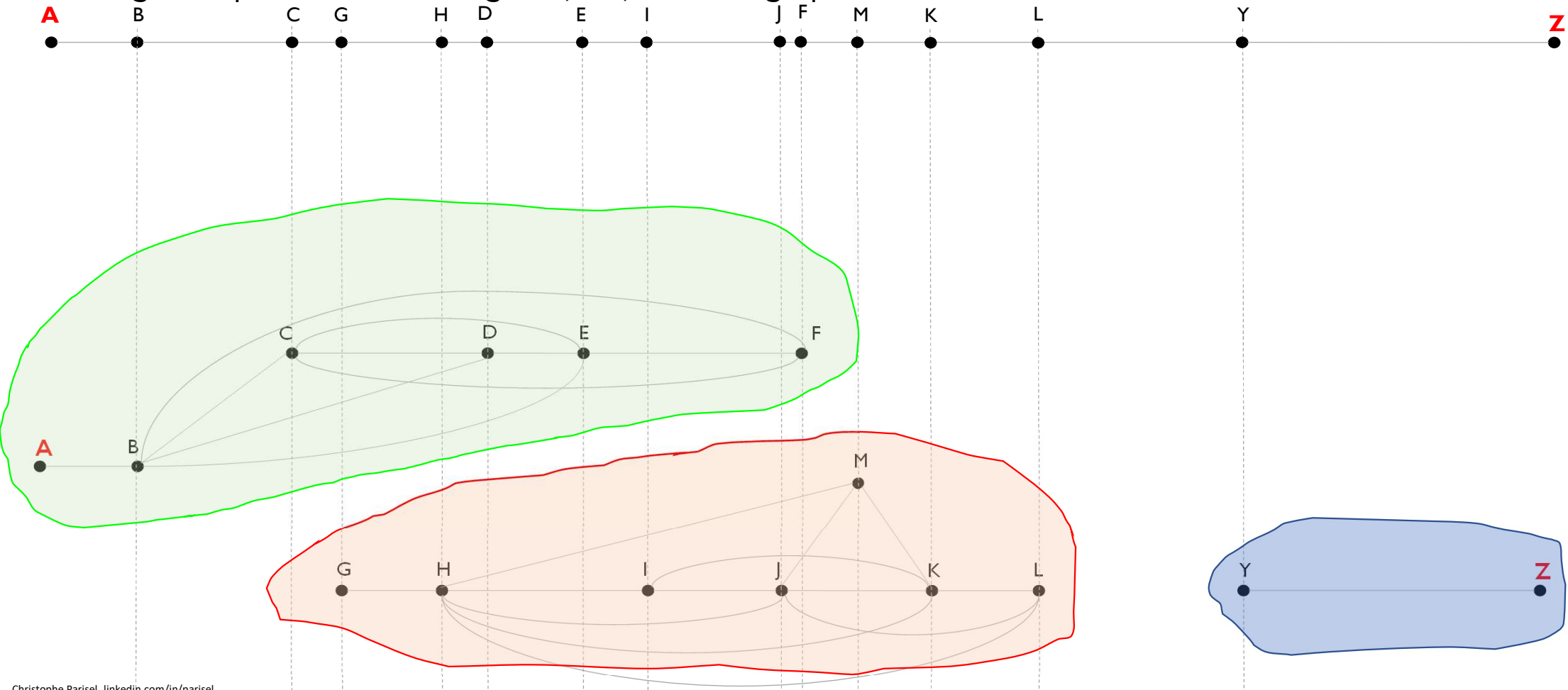


II – SMT proof with equality

Multiple equivalence classes = gaps

Suppose G isn't within B's range and X crashed (it doesn't show up in the timeseries)

⇒ we get 3 equivalence classes: green, red, blue. The graph is **disconnected**.



II – SMT proof with equality

Z3 SMT solver + python pseudo code (unoptimized for clarity!)

Question: prove that vertices A and Z do not belong to the same equivalence class.

If Z3 **says** it's unsatisfiable, then the graph is connected and there are **no compliance gaps**.

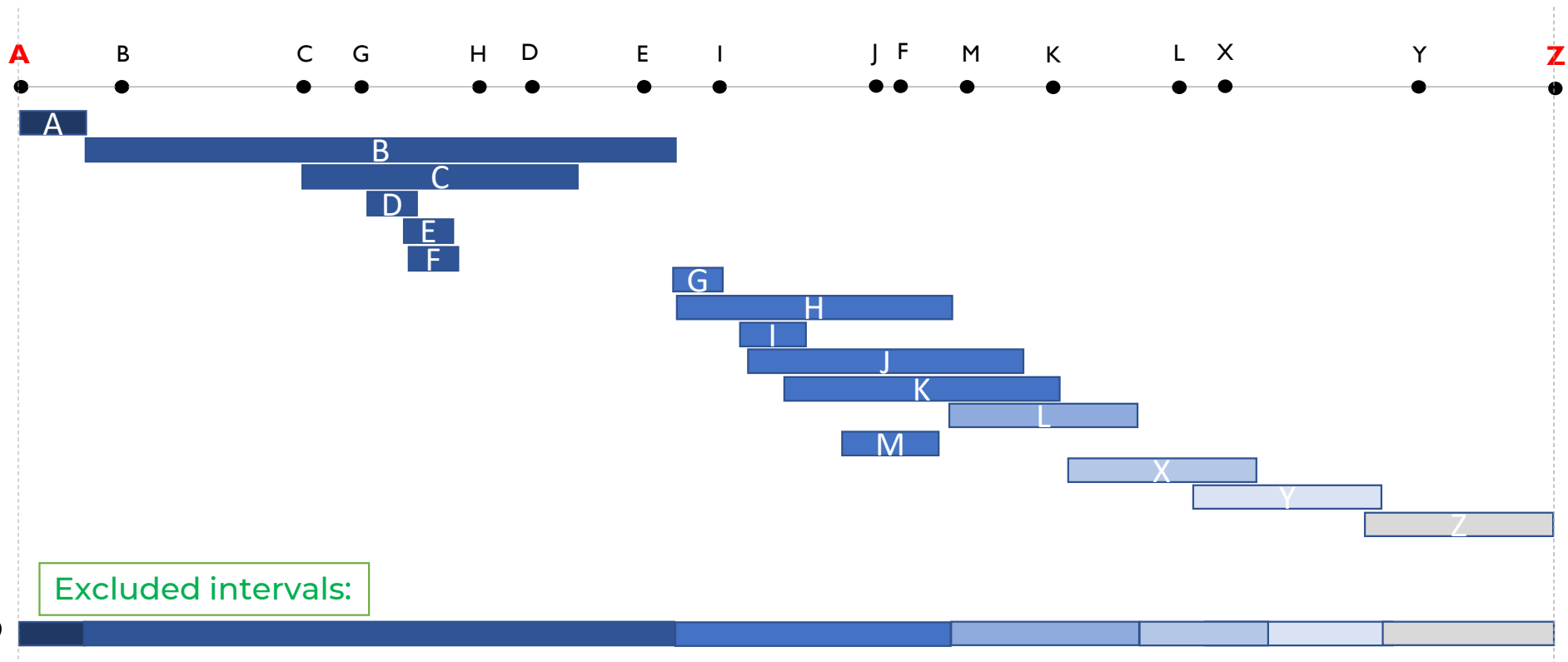
PROOF

```
def proveCompletenessWithEquality(A,Z):  
    global timeSeries  
    global sol  
    for datapoint in timeSeries:  
        if datapoint[' status ']!='OK':  
            break  
        for otherDatapoint in timeSeries:  
            if (datapoint['last']>otherDatapoint['first']):  
                sol.add(datapoint==otherDatapoint)  
    sol.add(A!=Z)  
    sol.check()
```

III – SMT proof with bitVectors

No gaps: $\varphi = \emptyset$

We enumerate all certified intervals and use a formula φ to **exclude them** one at a time from $[A,Z]$

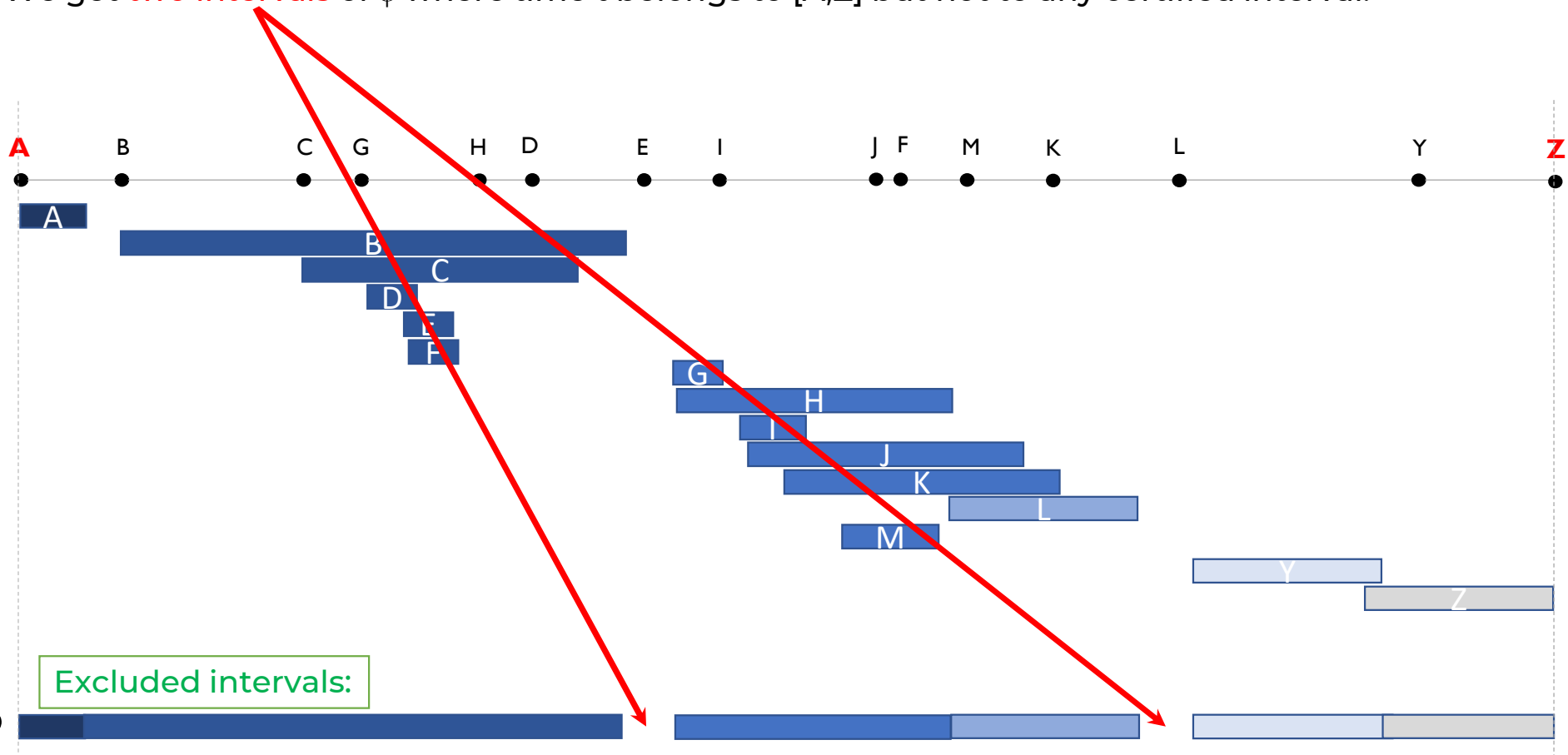


φ

III – SMT proof with bitVectors

Gaps: $\varphi \subseteq [A,Z]$ and $\varphi \neq \emptyset$

Suppose G isn't within B's range and X crashed (it doesn't show up in the timeseries)
We get **two intervals** of φ where time t belongs to $[A,Z]$ but not to any certified interval.



III – SMT proof with bitVectors

Z3 SMT solver + python pseudo code (unoptimized for clarity!)

Question: find a time $A \leq t \leq Z$ that doesn't belong to any certified interval.
If Z3 says it's unsatisfiable, there are **no compliance gaps**.

PROOF

```
def proveCompletenessWithBitVectors(A,Z):
    global timeSeries
    global sol
    t=BitVec('t',32)
    phy=True
    for datapoint in timeSeries:
        if datapoint[' status ']!='OK':
            break
        if datapoint[' uuid ']==A:
            phy=And(phy, UGE(t, datapoint['last']))
        elif datapoint[' uuid ']==Z:
            phy=And(phy, ULT(t, datapoint['first']))
        else:
            phy=And(phy, Or(ULT(t, datapoint['first']),UGE(t, datapoint['last'])))
    sol.add(phy==True)
    sol.check()
```


IV – Dealing with compliance gaps

To find compliance gaps, the easiest way is with Bitvectors and a model counting algorithm[*].

While the SMT solver can find a time assignment « t » in φ , we:

- 1) extend t downwards to t_{\min} and upwards to t_{\max}
- 2) rewrite $\varphi := \varphi \wedge \neg [t_{\min}, t_{\max}]$

Compliance gaps is the set of $[t_{\min}, t_{\max}]$ intervals

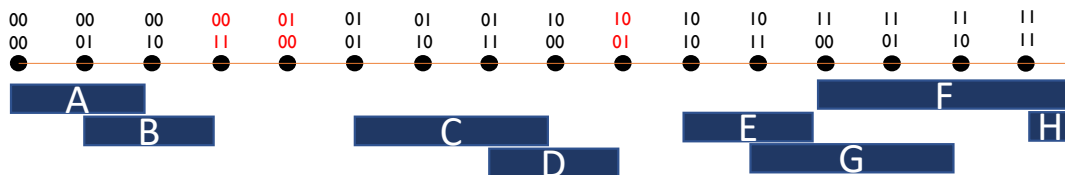
[*]: <https://www.aaai.org/Papers/AAAI/2006/AAAI06-009.pdf>

A draft for leveraging quantum speedup

With a Classical Computer, we've seen that to prove control gaps, we ask a SAT solver to check whether $\phi() == \text{True}$ is satisfiable.

Now with a Quantum Computer, we use Grover search as a super-fast SAT solver. We invert logical expression $\phi() == \text{True}$. Compliance gaps are the preimages of ϕ .

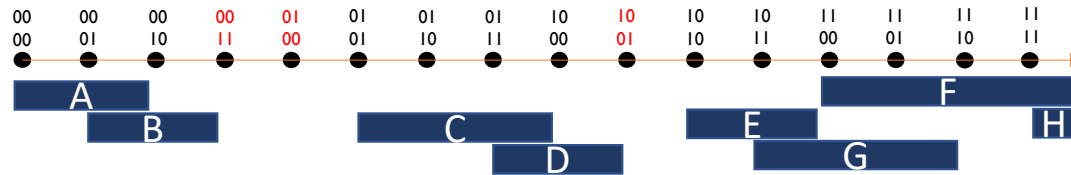
The compliance period is: [0000, 0001, ..., 1111]. Say the control executed 8 times, labelled A to H:



Christophe Parisel, [linkedin.com/in/parisel](https://www.linkedin.com/in/parisel)

V – Hybrid quantum computing

Toy example (continued)



Now let's build our formula. We **exclude** all successful runs from [0000, ... ,1111]:

$$\varphi := \neg A \wedge \neg B \wedge \neg C \wedge \neg D \wedge \neg E \wedge \neg F \wedge \neg G \wedge \neg H$$

Since we span 16 timestamps, we need 4 bits (a, b, c and d) to address them all.

- timestamp 0000 can be represented as $(\neg a \wedge \neg b \wedge \neg c \wedge \neg d)$
- A, which is interval [0000,0001], can be represented as $(\neg a \wedge \neg b \wedge \neg c \wedge \neg d) \vee (\neg a \wedge \neg b \wedge \neg c \wedge d)$
- $\neg A$ can be represented as $\neg((\neg a \wedge \neg b \wedge \neg c \wedge \neg d) \vee (\neg a \wedge \neg b \wedge \neg c \wedge d))$

Eventually, φ can be fully represented:

$$\varphi(a, b, c, d) := \neg((\neg a \wedge \neg b \wedge \neg c \wedge \neg d) \vee (\neg a \wedge \neg b \wedge \neg c \wedge d)) \wedge \neg((\neg a \wedge \neg b \wedge c \wedge \neg d) \vee (\neg a \wedge \neg b \wedge c \wedge d)) \wedge \neg((\neg a \wedge b \wedge \neg c \wedge \neg d) \vee (\neg a \wedge b \wedge \neg c \wedge d)) \wedge \neg((\neg a \wedge b \wedge c \wedge \neg d) \vee (\neg a \wedge b \wedge c \wedge d)) \wedge \neg((a \wedge \neg b \wedge \neg c \wedge \neg d) \vee (a \wedge \neg b \wedge \neg c \wedge d)) \wedge \neg((a \wedge \neg b \wedge c \wedge \neg d) \vee (a \wedge \neg b \wedge c \wedge d)) \wedge \neg((a \wedge b \wedge \neg c \wedge \neg d) \vee (a \wedge b \wedge \neg c \wedge d)) \wedge \neg((a \wedge b \wedge c \wedge \neg d) \vee (a \wedge b \wedge c \wedge d))$$

We are now ready to invert $\varphi(a, b, c, d) := \text{True}$

V – Hybrid quantum computing

Toy example (continued)

Let's use ϕ as a Qiskit logical expression Oracle[*] and invert it with Grover:

```
from qiskit.providers.aer import AerSimulator
from qiskit.aqua.algorithms import Grover
from qiskit.aqua.components.oracles import LogicalExpressionOracle
backend = AerSimulator()
```

$\phi(a, b, c, d) := \text{True}$



```
expression=''
(~((~a&~b&~c&~d)|(~a&~b&~c&d))&(~((~a&~b&~c&d)|(~a&~b&c&~d))&(~((~a&b&~c&d)|(~a&b&c&~d)|(~a&b&c&d)))&(~((~a&b&c&d)|(a&~b&~c&~d))&(~((a&~b&c&~d)|(a&~b&c&d)))&(~((a&~b&c&d)|(a&b&~c&~d)|(a&b&~c&d)|(a&b&c&~d)))&(~(a&b&c&d))&(~((a&b&~c&~d)|(a&b&~c&d)|(a&b&c&~d)|(a&b&c&d)))
''
```

```
oracle = LogicalExpressionOracle(expression=expression,optimization=True)
grover = Grover(oracle,iterations=1)
result = grover.run(backend, shots=1000)
counts = result['measurement']
print(result)
```

Here are the pre-images obtained in a sample measurement:

```
Results {'0001': 4, '1011': 6, '1101': 4, '1110': 4, '0111': 2, '0000': 6, '0011': 3, '1000': 6, '1001': 318, '0101': 6, '1100': 316, '0010': 308, '1010': 5, '1111': 4, '0110': 3, '0100': 5}
```

Accounting for bits ordering (MSB to LSB), the Quantum circuit yields 3 outstanding results: 0100, 0011, 1001. It is computationally inexpensive to verify that all 3 are **compliance gaps**.

[*]: <https://qiskit.org/documentation/stable/0.26/stubs/qiskit.aqua.components.oracles.LogicalExpressionOracle.html>

V – Hybrid quantum computing

Performance consideration

Theory states that Grover needs \sqrt{N} shots to find at least one performance gap in a period of N timestamps. In our toy example, $N=16$ so a Quantum Computer should only need 4 shots to be nearly 100% accurate.

On a classical computer, the number of shots depends on when the first compliance gap happens during the period. On average, the earliest occurrence of 3 independant variables in range $[0, N-1]$ is about $N/4$.

So a Classical Computer should take about 4 shots to be only 50% accurate.

But this does not account for the need to set-up a circuitry to poll the compliance interval. If « g » is the number of gaps,

- ❑ In a classical computer, a bitmap ranging the timestamp space must be populated (it takes $N-g$ write operations)
- ❑ In a quantum computer, an oracle must be built iteratively for each of the $N-g$ timestamps.