Project Munin

Audit Event Topologies

Tutorial v1.101 - 10th Feb 2023

Background

- Project Munin is tasked with building a Protocol Verifier
- The Protocol Verifier monitors the behaviour of another system but receiving and analysing audit events sent from the Monitored System
- The Protocol Verifier is configured to expect well-defined sequences of audit events
    - Configuration of the Protocol Verifier is done using the PLUS language defined separately though illustrated here
- At run time, audit events are received and compared against the allowed expected sequences
    - Any deviation from an expected sequence logs an error
- The Protocol Verifier can deal with significant variation in the runtime audit event sequences to support error conditions, repeated behaviour, fragmentation, etc.
    - This range of supported variation is known as the Audit Event Topologies
    - That is the focus of this tutorial
- Prior to configuration the Protocol Verifier has no knowledge of any specific protocols
    - The Protocol Verifier is built to support a range of topologies
    - Any protocols that conform to those topologies will be supportable

The Structure of an Audit Event

- Focus on semantics
  - Essential
    - Audit event id – a unique identifier of the audit event
    - Job id - a unique identifier of the Job that the audit event is part of
    - Audit event type – which can be matched to the expected audit event type
    - Previous audit event ids – the list of 0, 1 or many audit event ids from the immediately prior audit event(s)
      - No previous audit event id indicates an Audit Event that starts a new sequence
      - Multiple previous audit events indicate a fan-in or merge point
      - Multiple audit events with the same previous event id indicates a fan-out or fork point
  - Beneficial
    - Job type id – essential if the same audit event type could occur in different job types
    - Application id and/or node id – indicates the source of the audit event allowing additional validation
    - Audit event time – not currently used, establishing order is done using previous audit event id

Some Terminology

- **Audit Event** – a single telemetry event from the system being monitored by the Digital twin
- **Job** – existing term for a set of audit events linked by a common JobId
- **Sequence** – a series of audit events linked by one audit event referring to the previous audit event explicitly.
- **Branch** – a parallel series of audit events within a Sequence
  - Branch Extent – a property of a Sequence that defines the current number of parallel event paths
- **Fork** – the point in a sequence where one or more branches begin
  - Instance or Type forks
  - AND, XOR or IOR forks
- **Merge** – the point in a sequence where 2 or more branches join
- A Job can consist of a series of Sequences
- We need to know the types of Jobs, Sequences and Audit Events that the Digital Twin needs to handle
  - Job Definition (JobDefinition)
  - Sequence Definition (AESequenceDefinition)
  - Audit Event Definition (AEDefinition)

Instances vs Definitions

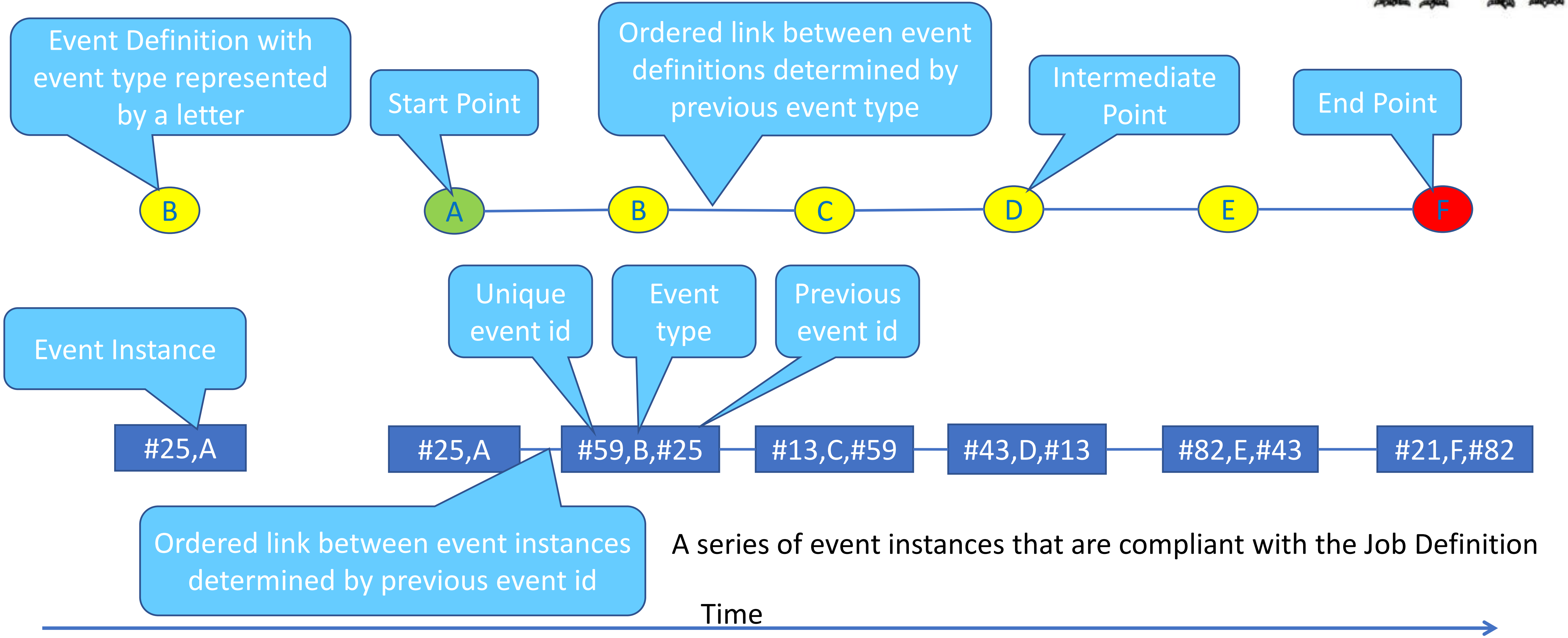
- The Monitored System will send Audit Events to the Digital Twin. We refer to these as event instances.
- The Digital Twin needs to hold a specification of the expected legal sets of event instances so that it can detect illegal event instances or Jobs
- This specification consists of:
  - Job Definitions
  - Sequence Definitions
  - Audit Event Definitions
- The expected audit event instance mesh is specified by a definition graph
- The definition graph constrains the legal set of audit event instance graphs that can be supported
  - The different forms that this definition graph can take are called the Graph Topology

PlantUML Activity Diagram

- The objective is to be able to configure the Protocol Verifier in as easy a way as possible
- The PlantUML Activity Diagram has been identified as the easiest way to do this
- This tutorial introduces each topological feature and then illustrates it with the PlantUML textual and diagrammatic representation
- We only need a subset of PlantUML to do this and so that will be a constrained and parsable subset (named PLUS)
    - Which will be used to generate the Protocol Verifier event definitions and audit event definitions automatically

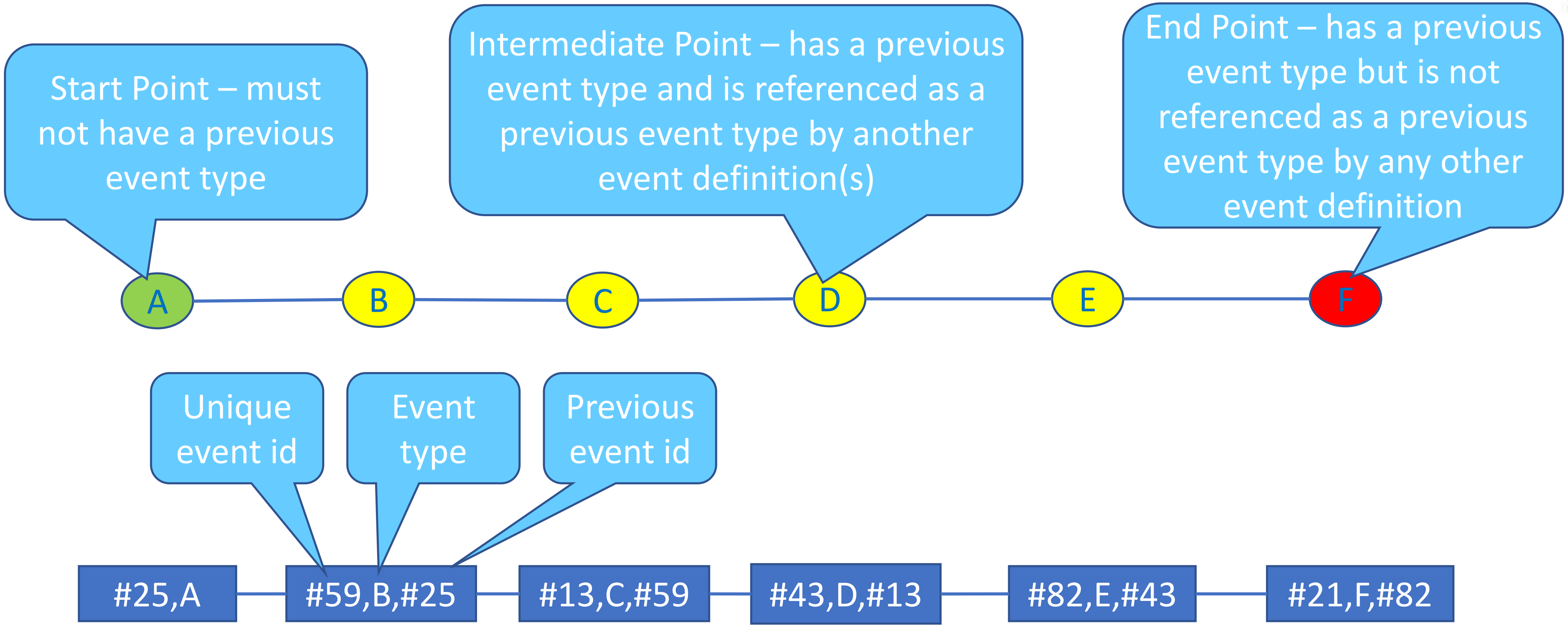
Topology diagram conventions

Note: Only a subset of the audit event fields is shown for clarity


Event Definition with event type represented by a letter
B
Start Point
A
Ordered link between event definitions determined by previous event type
B
C
Intermediate Point
D
E
End Point
F
Event Instance
#25,A
Unique event id
Event type
Previous event id
#25,A
#59,B,#25
#13,C,#59
#43,D,#13
#82,E,#43
#21,F,#82
Ordered link between event instances determined by previous event id
Time

A series of event instances that are compliant with the Job Definition

Significant Points in Audit Event Topology



Start Point – must not have a previous event type

Intermediate Point – has a previous event type and is referenced as a previous event type by another event definition(s)

End Point – has a previous event type but is not referenced as a previous event type by any other event definition

Unique event id

Event type

Previous event id

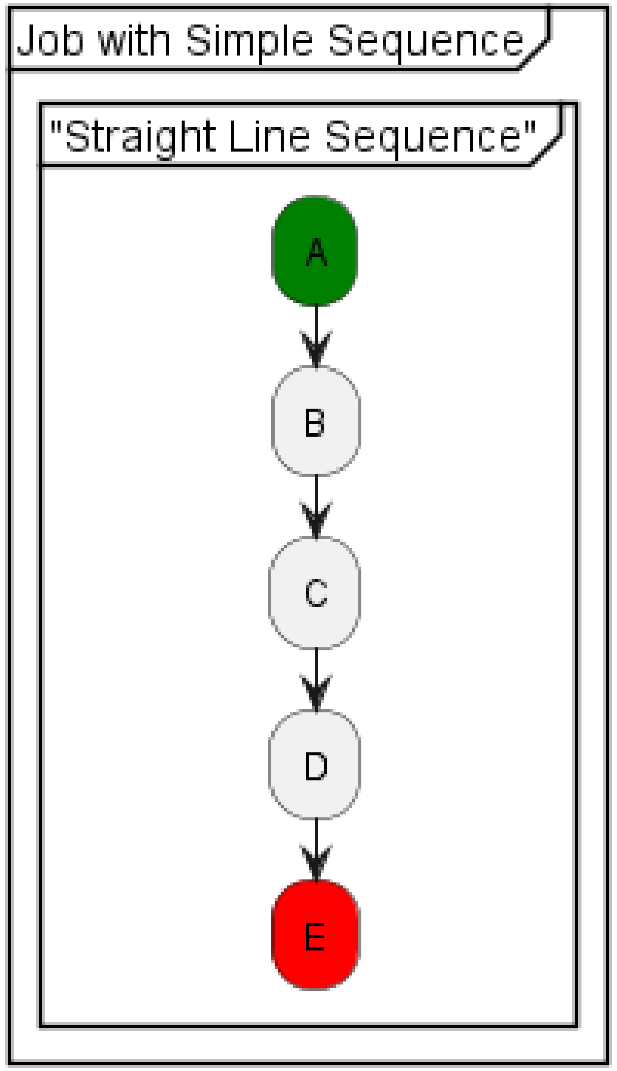
#25,A	#59,B,#25	#13,C,#59	#43,D,#13	#82,E,#43	#21,F,#82

A series of event instances that are compliant with the Job Definition (above)

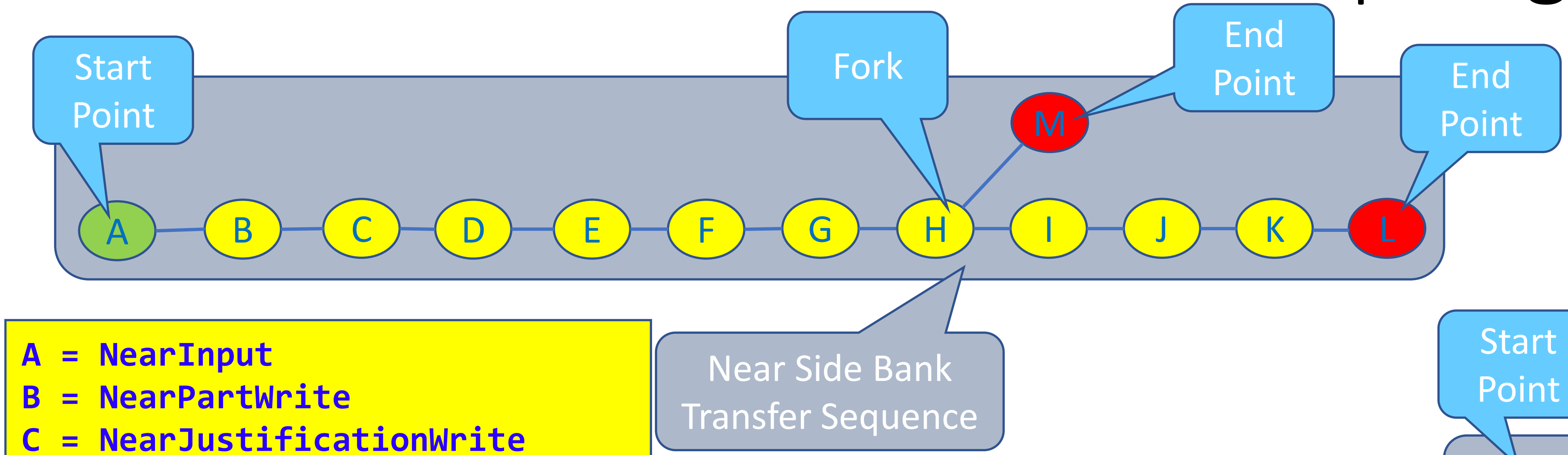
Time

Definition using PlantUML Activity Diagram

```
@startuml
partition "Job with Simple Sequence" {
  group "Straight Line Sequence"
      #green:A;
      :B;
      :C;
      :D;
      #red:E;
  end group
}
@enduml
```



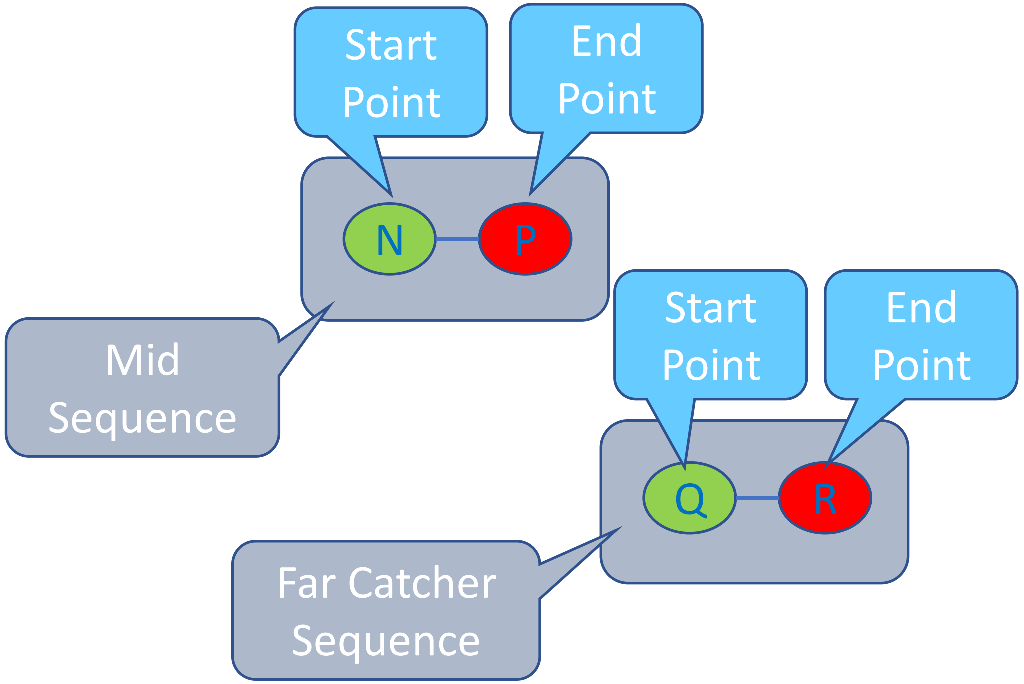
Bank Transfer Job Definition Topology



A	= NearInput
B	= NearPartWrite
C	= NearJustificationWrite
D	= AgentRequestWidgets
E	= AuthoriserReadWidgetRequest
F	= AuthoriserParsedEvidence
G	= AuthoriserSendWidgets
H	= AgentReceiveWidgets
I	= AgentPartRead
J	= AgentWritePayload
K	= GatewayReadPayload
L	= GatewayWritePayload
M	= AgentProcessedRequest

N	= MidProcessPayload
P	= MidWritePayload

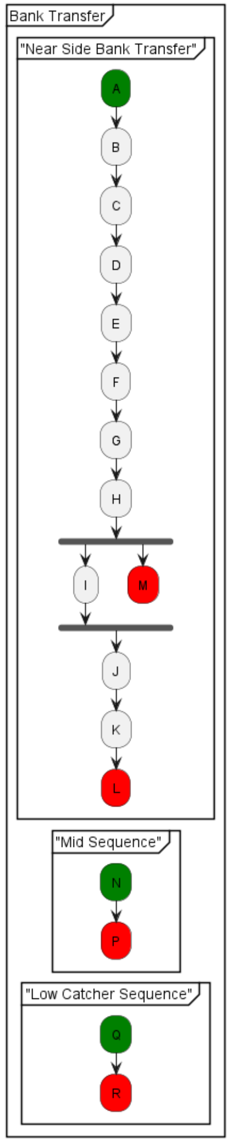
Q	= FarInput
R	= FarOutput



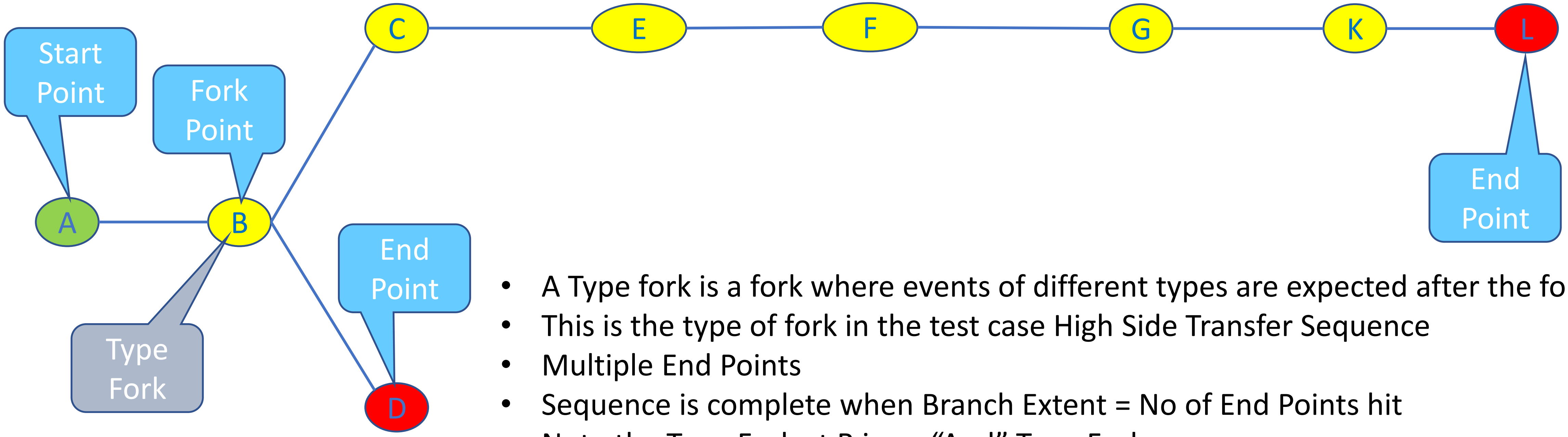
Time

Definition using PlantUML Activity Diagram

```
@startuml
partition "Bank Transfer" {
group "Near Side Bank Transfer"
      #green:A;
      :B;
      :C;
      :D;
      :E;
      :F;
      :G;
      :H;
  fork
      :I;
  fork again
      #red:M;
      detach
  end fork
      :J;
      :K;
      #red:L;
      detach
end group
group "Mid Sequence"
      #green:N;
      #red:P;
      detach
end group
group "Low Catcher Sequence"
      #green:Q;
      #red:R;
end group
}
@enduml
```



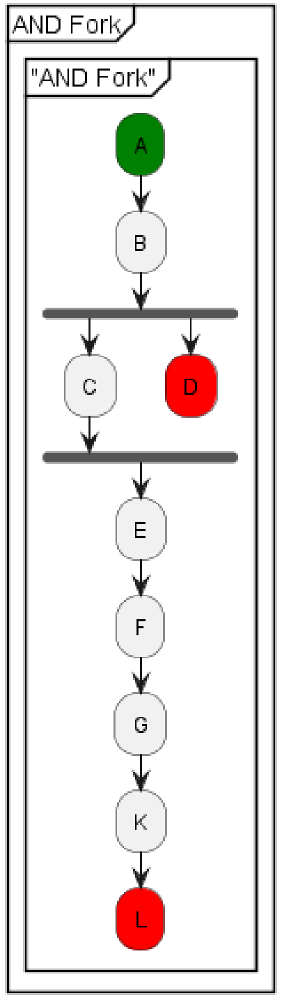
Job Topology – “And” Type Fork



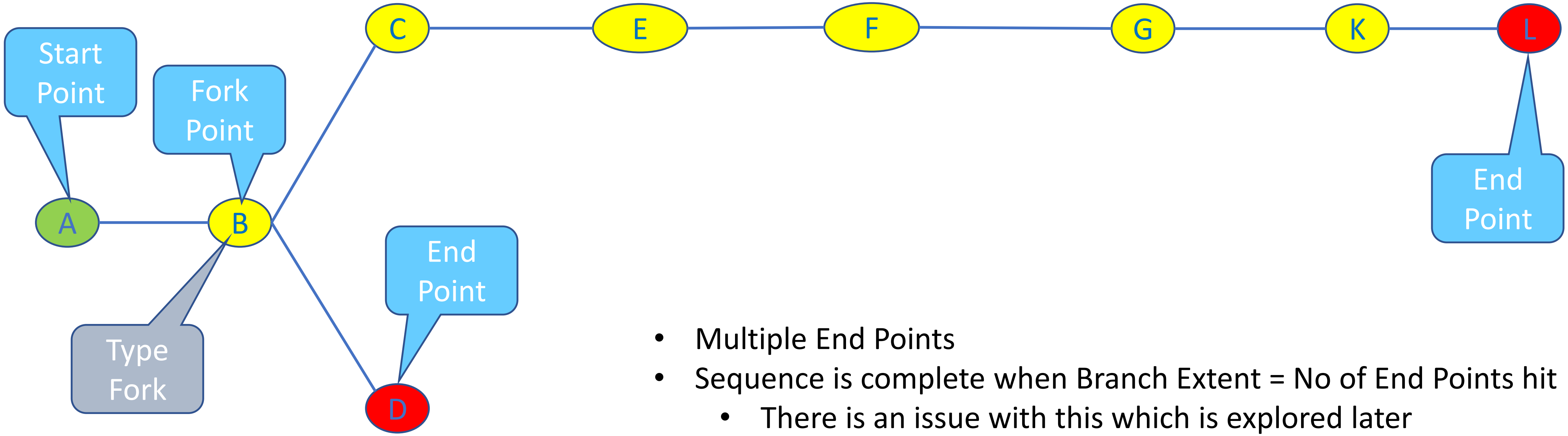
- A Type fork is a fork where events of different types are expected after the fork point
- This is the type of fork in the test case High Side Transfer Sequence
- Multiple End Points
- Sequence is complete when Branch Extent = No of End Points hit
- Note the Type Fork at B is an “And” Type Fork
  - After B we expect both C and D
- This is a constrained fork – both C and D must be observed after B
  - Checked on job completion

AND Fork using PlantUML Activity Diagram

```
@startuml
partition "AND Fork" {
group "AND Fork"
      #green:A;
      :B;
  fork
      :C;
  fork again
      #red:D;
      detach
  end fork
      :E;
      :F;
      :G;
      :K;
      #red:L;
      detach
end group
}
@enduml
```



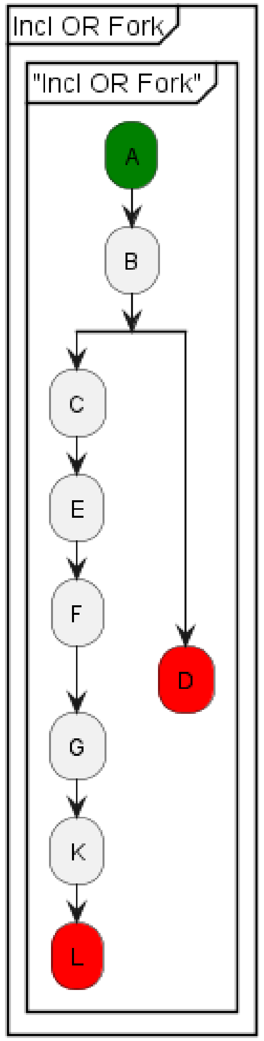
Job Topology – "Inclusive Or" Type Fork



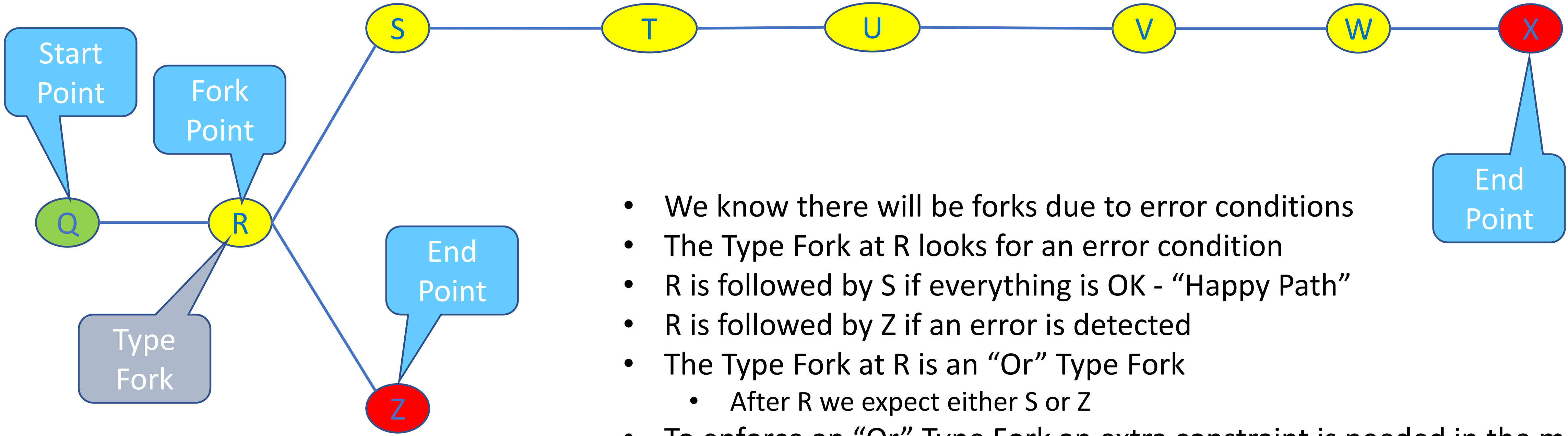
- Multiple End Points
- Sequence is complete when Branch Extent = No of End Points hit
  - There is an issue with this which is explored later
- Note the Type Fork at B is an "Inclusive Or" Type Fork
  - After B we expect C or D or both C and D
- This is an unconstrained fork
  - Adds complexity to determining job completion

Inclusive OR Fork using PlantUML Activity Diagram

```
@startuml
partition "Incl OR Fork" {
group "Incl OR Fork"
      #green:A;
      :B;
  split
      :C;
      :E;
      :F;
      :G;
      :K;
      #red:L;
      detach
  split again
      #red:D;
      detach
  end split
end group
}
@enduml
```



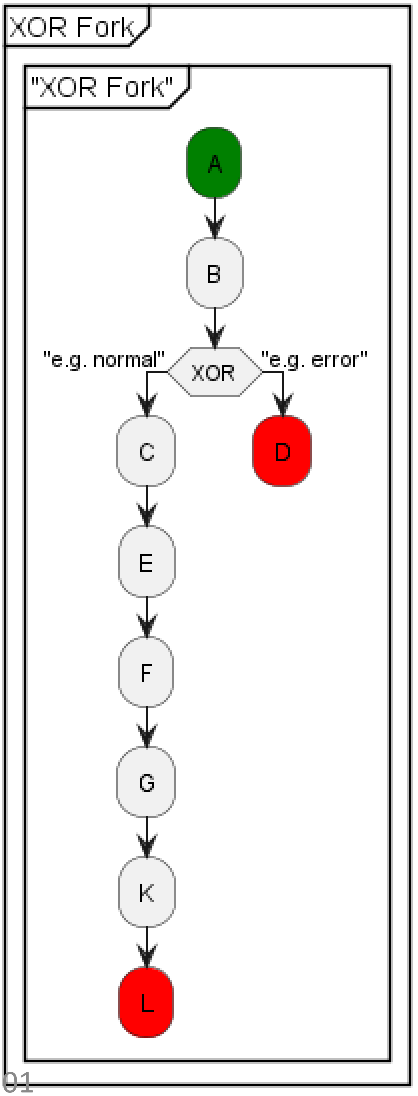
Job Topology – “Exclusive Or” Type Fork



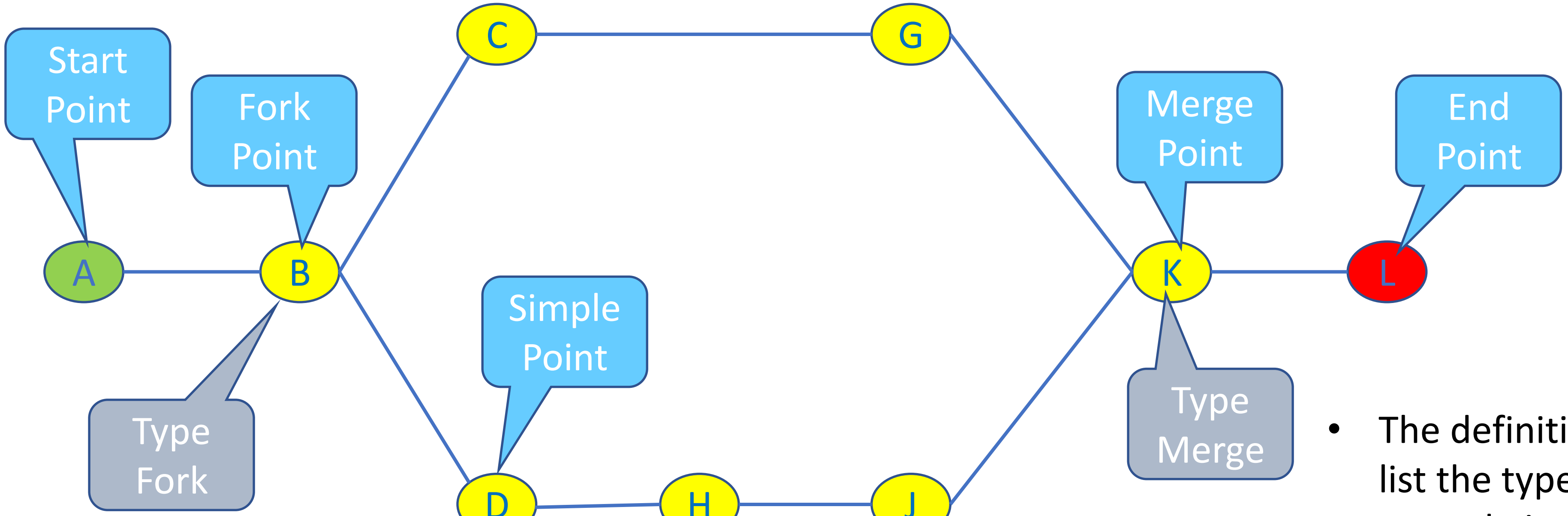
- We know there will be forks due to error conditions
- The Type Fork at R looks for an error condition
- R is followed by S if everything is OK - “Happy Path”
- R is followed by Z if an error is detected
- The Type Fork at R is an “Or” Type Fork
  - After R we expect either S or Z
- To enforce an “Or” Type Fork an extra constraint is needed in the model
  - S and Z are mutually exclusive
- Sequence completion detected by branch extent = no of End Points hit
  - Still works for topologies including “Exclusive Or” Type Forks
- This is a constrained fork – after R we expect S or Z but not both

Exclusive OR Fork using PlantUML Activity Diagram

```
@startuml
partition "XOR Fork" {
group "XOR Fork"
      #green:A;
      :B;
  if (XOR) then ("e.g. normal")
      :C;
      :E;
      :F;
      :G;
      :K;
      #red:L;
      detach
  else ("e.g. error")
      #red:D;
      detach
  endif
end group
}
@enduml
```



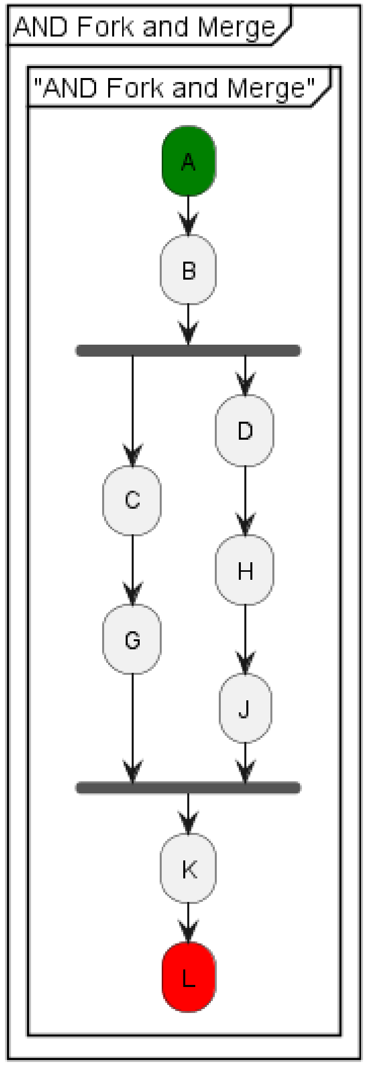
Job Topology – Type Fork and Merge



- The definition of a Type Merge must list the types of events that can precede it.

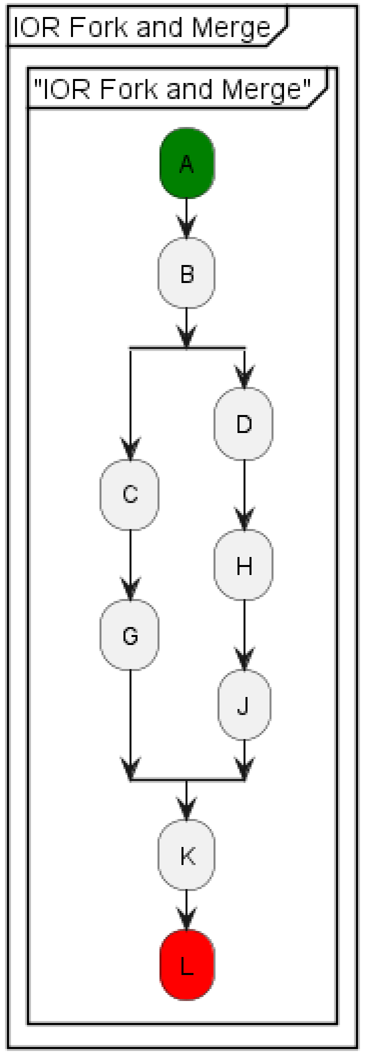
AND Fork and Merge using PlantUML Activity Diagram

```
@startuml
partition "AND Fork and Merge" {
group "AND Fork and Merge"
  #green:A;
  :B;
  fork
      :C;
      :G;
  fork again
      :D;
      :H;
      :J;
  end fork
  :K;
  #red:L;
  detach
end group
}
@enduml
```



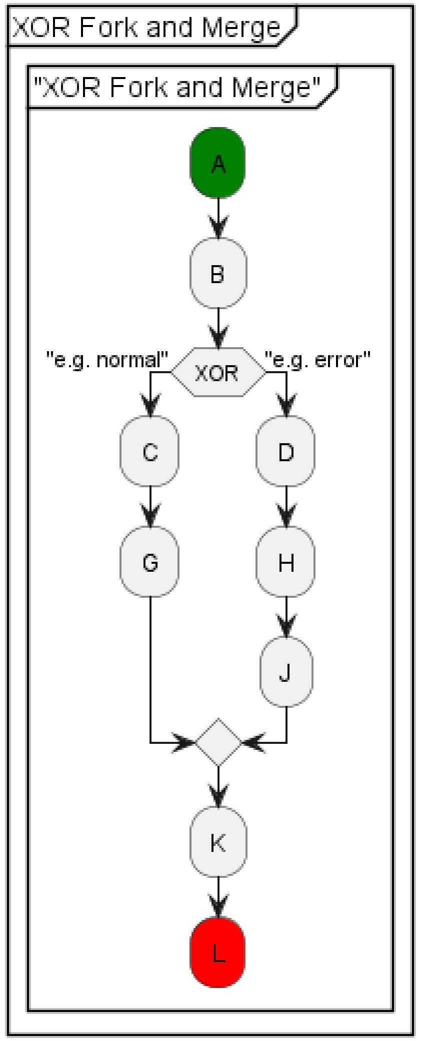
Inclusive OR Fork and Merge using PlantUML Activity Diagram

```
@startuml
partition "IOR Fork and Merge" {
group "IOR Fork and Merge"
  #green:A;
  :B;
  split
      :C;
      :G;
  split again
      :D;
      :H;
      :J;
  end split
  :K;
  #red:L;
  detach
end group
}
@enduml
```

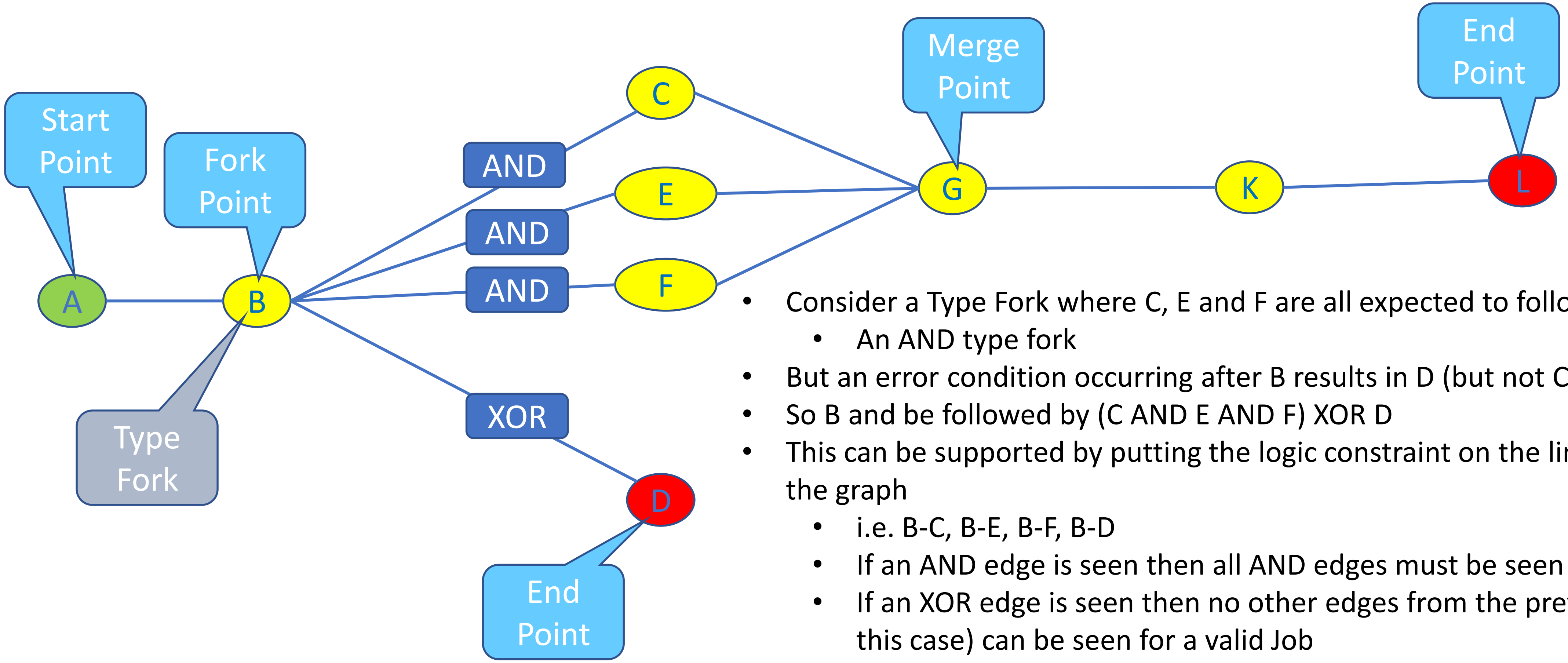


Exclusive OR Fork and Merge using PlantUML Activity Diagram

```
@startuml
partition "XOR Fork and Merge" {
group "XOR Fork and Merge"
  #green:A;
  :B;
  if (XOR) then ("e.g. normal")
      :C;
      :G;
  else ("e.g. error")
      :D;
      :H;
      :J;
  endif
  :K;
  #red:L;
end group
}
@enduml
```



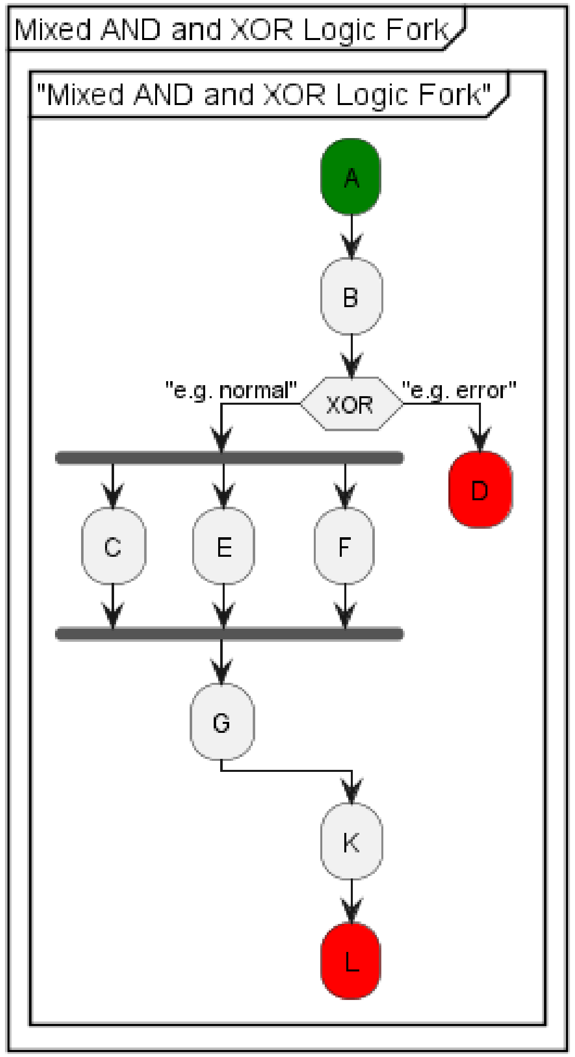
Job Topology – Mixed Logic Type Fork



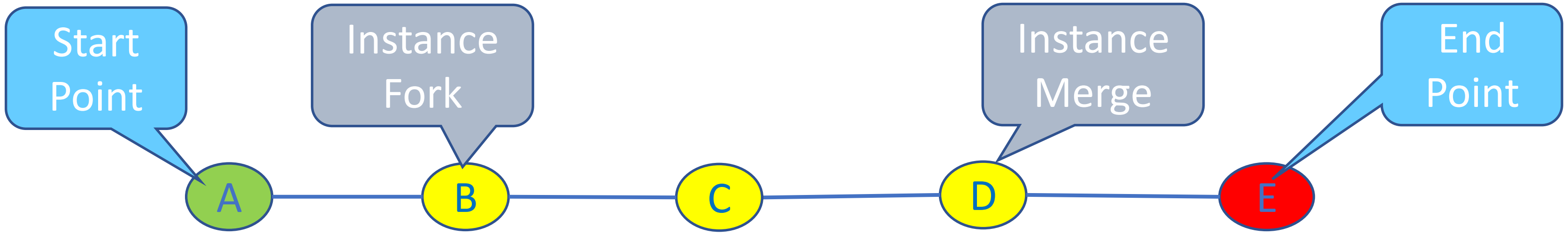
- Consider a Type Fork where C, E and F are all expected to follow B
  - An AND type fork
- But an error condition occurring after B results in D (but not C, E and F)
- So B and be followed by (C AND E AND F) XOR D
- This can be supported by putting the logic constraint on the links or edges of the graph
  - i.e. B-C, B-E, B-F, B-D
  - If an AND edge is seen then all AND edges must be seen for a valid Job
  - If an XOR edge is seen then no other edges from the previous event (B in this case) can be seen for a valid Job

Mixed Logic Fork and Merge using PlantUML Activity Diagram

```
@startuml
partition "Mixed AND and XOR Logic Fork" {
group "Mixed AND and XOR Logic Fork"
      #green:A;
      :B;
  if (XOR) then ("e.g. normal")
    fork
      :C;
    fork again
      :E;
    fork again
      :F;
    end fork
      :G;
  else ("e.g. error")
      #red:D;
      detach
  endif
      :K;
      #red:L;
      detach
end group
}
@enduml
```

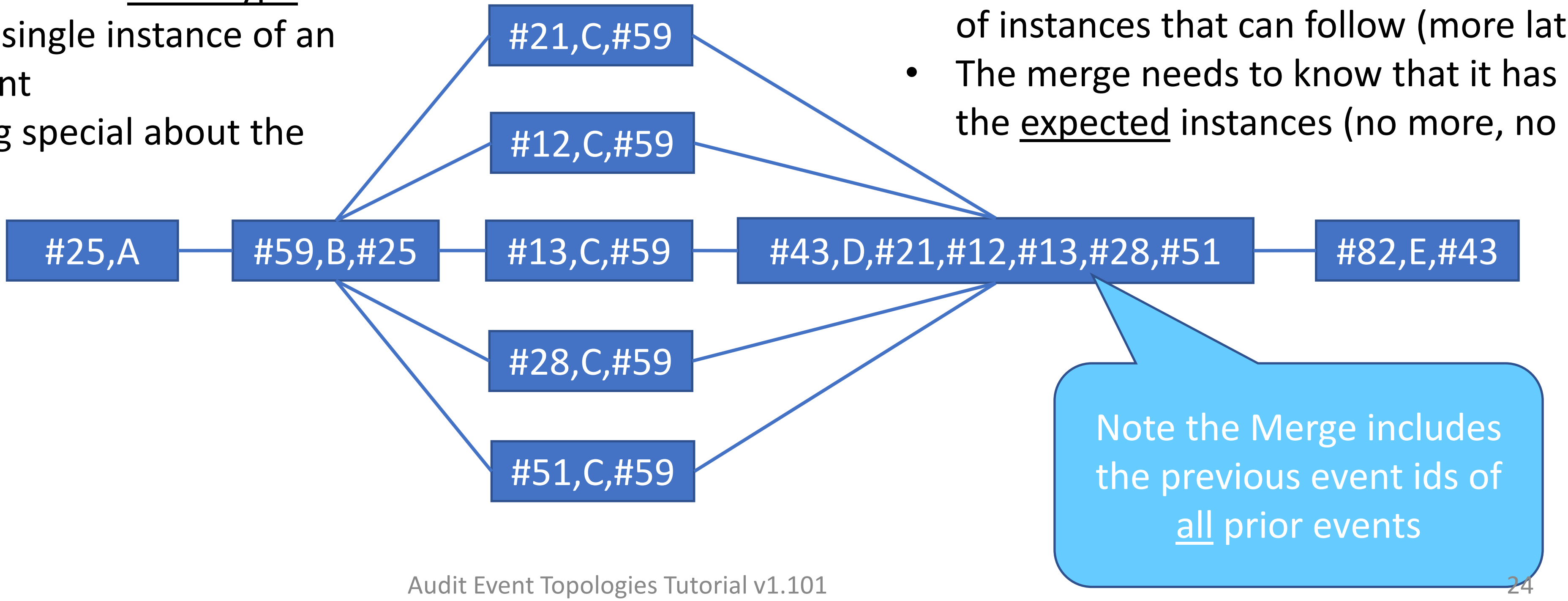


Job Topology – Instance Fork and Merge



- An Instance fork is a fork where multiple instances of an event of the same type are expected after a single instance of an event at the fork point
- Note there is nothing special about the event definition

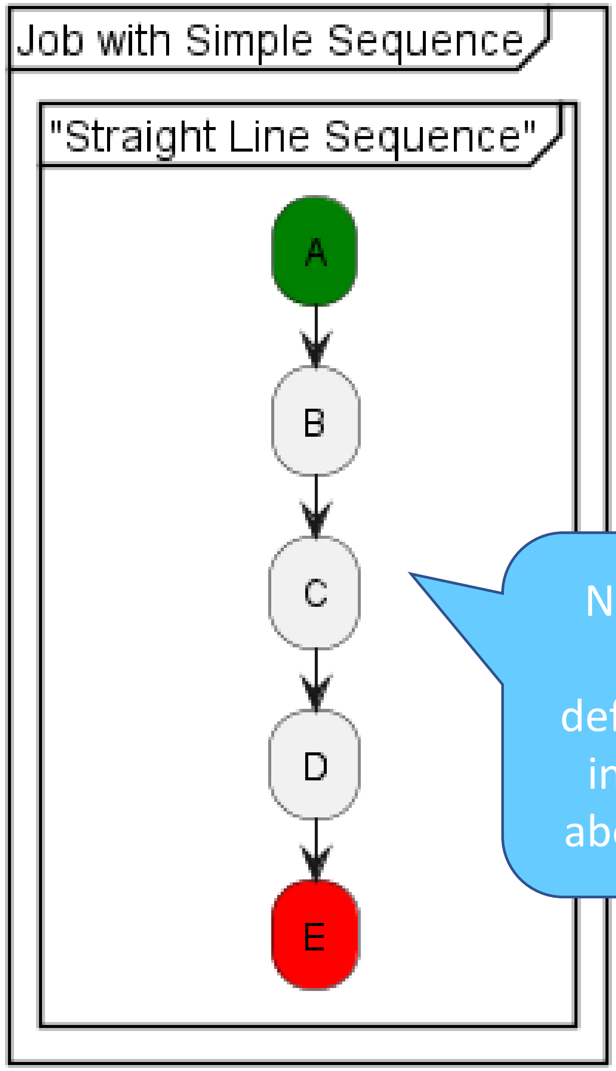
- An Instance fork definition can have a “Branch Count” which states the number of instances that can follow (more later)
- The merge needs to know that it has all of the expected instances (no more, no less)



Note the Merge includes the previous event ids of all prior events

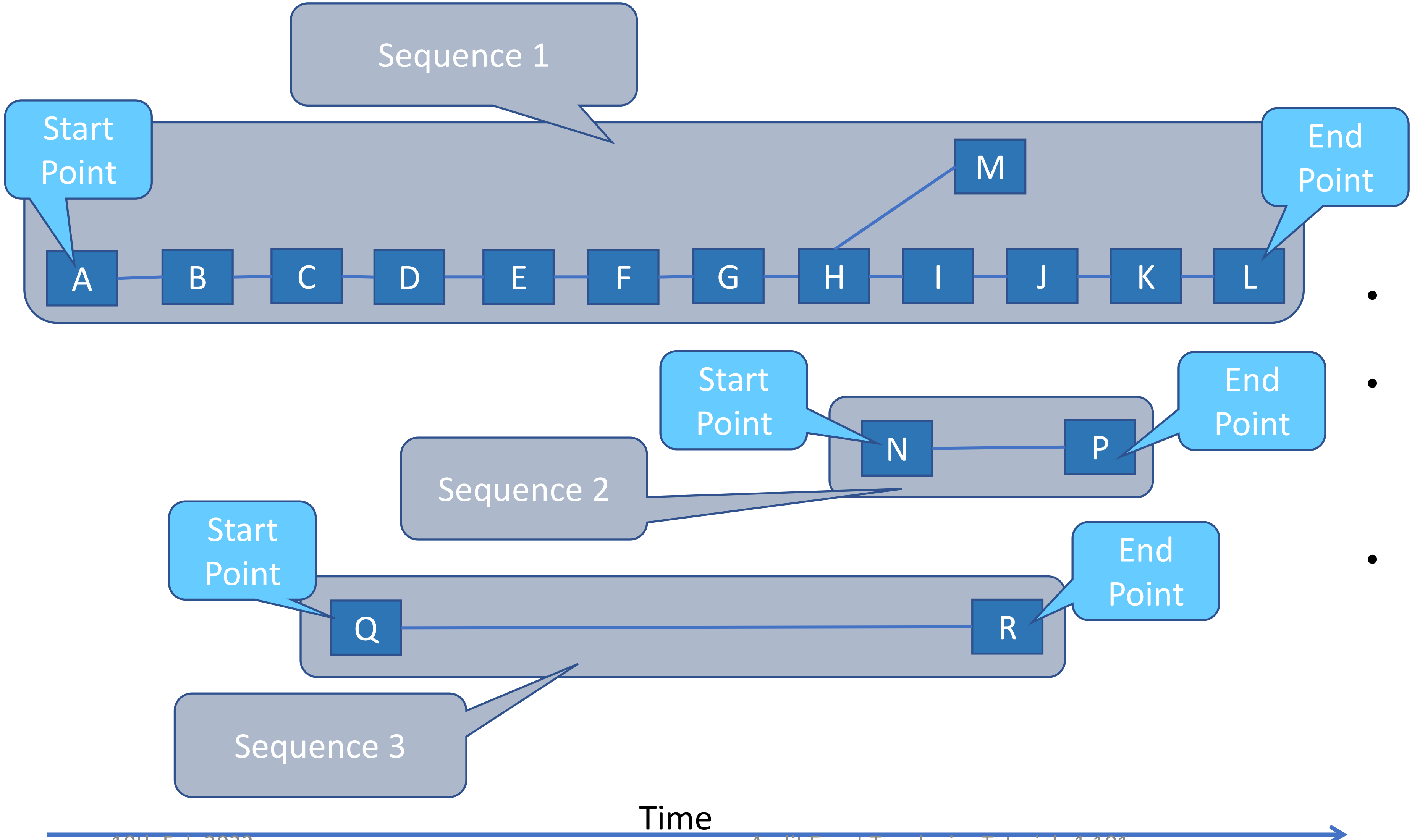
Instance Fork using PlantUML Activity Diagram

```
@startuml
partition "Job with Simple Sequence" {
  group "Straight Line Sequence"
      #green:A;
      :B;
      :C;
      :D;
      #red:E;
  end group
}
@enduml
```



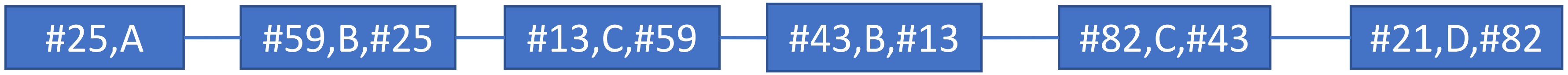
Note there is nothing special about the definition to support an instance fork – it's all about the runtime data

Sequences in a Job are not linked



- Events from different sequences can be interleaved
- There is nothing in the input data that allows the sequences to be correctly ordered
- If the order of sequences within a job is to be verified then additional data is required

Sequences with repeating events - problem



#25,A

#59,B,#25

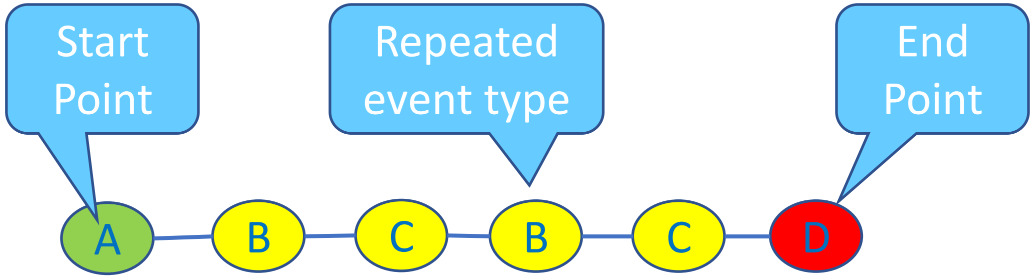
#13,C,#59

#43,B,#13

#82,C,#43

#21,D,#82

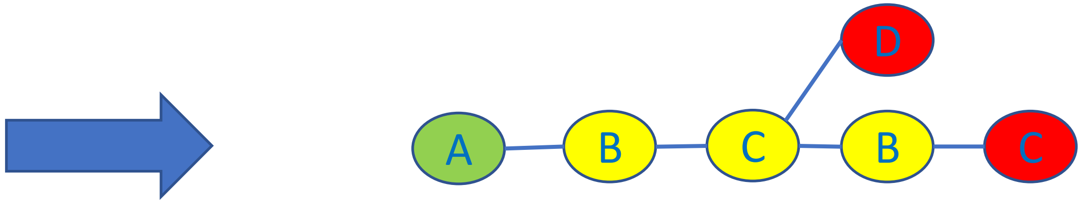
- This issue is present in the example of the CDRA job
- The event definition interface defines the type of the previous event
- So it tells us…
  - Event C is preceded by event B, and
  - Event D is preceded by event C
- With repeated event types the sequence definition is now ambiguous
- For example is the event C that precedes event D the first or second occurrence of event C



Start Point

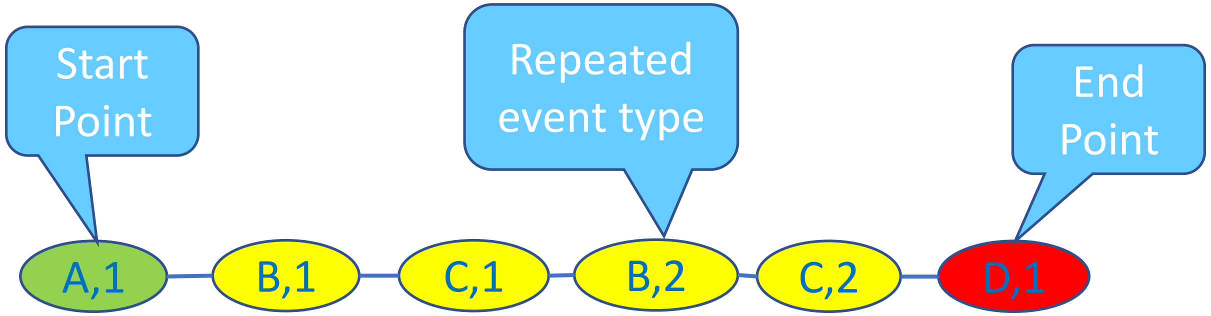
Repeated event type

End Point



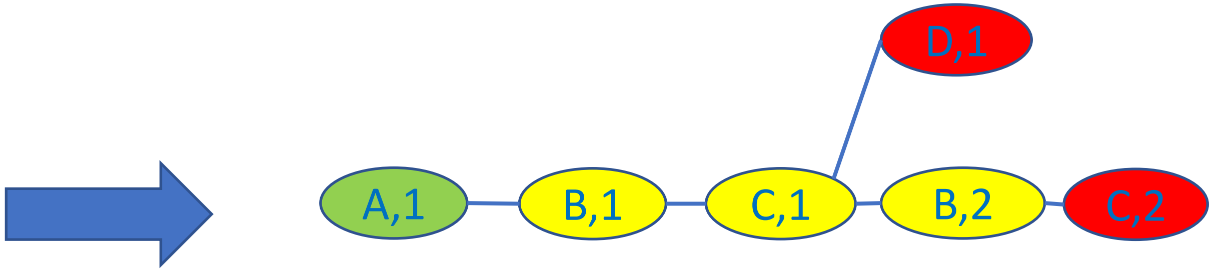
Sequences with repeating events - solution

- So an event definition is identified by:
  - eventType : string
  - occurrenceId : integer
- And in the event definition the previous event is referred to like this
  - previousEventType : string
  - occurrenceId : integer
- Now event B,2 precedes event C,2 and event C,2 precedes event D,1
- No longer ambiguous



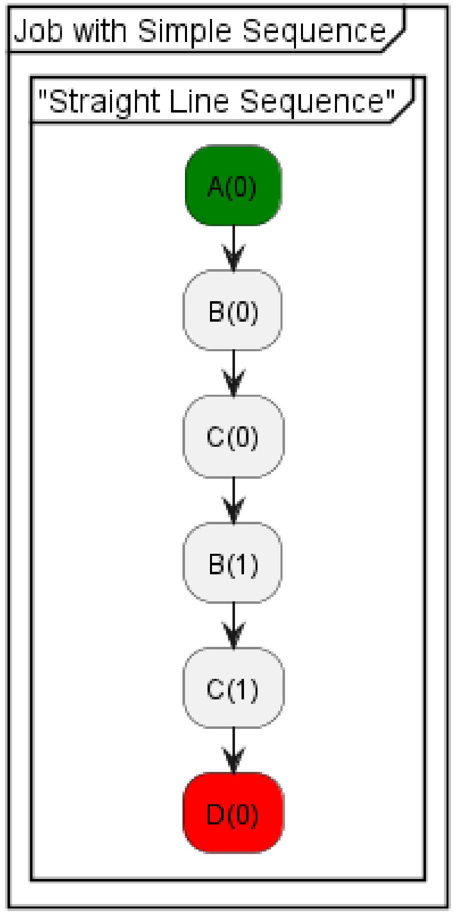
This works fine and is very rigorous for known fixed repetitions. It will not work for variable numbers of repetitions

- Alternatively event C,1 precedes event D,1



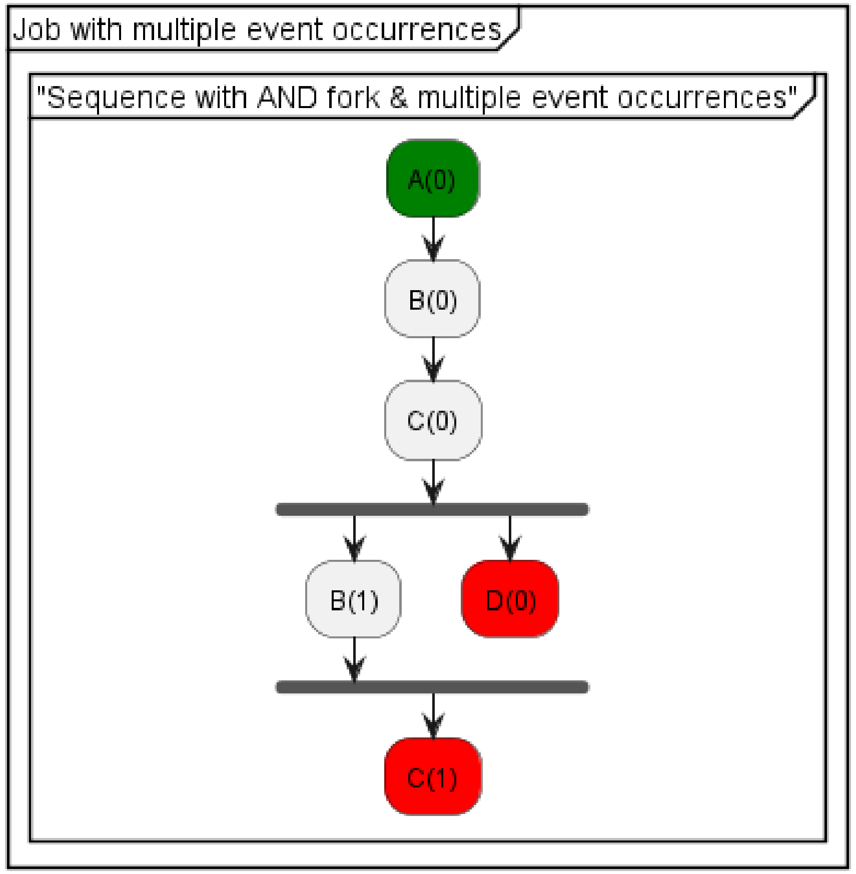
Event Occurrences using PlantUML Activity Diagram

```
@startuml
partition "Job with Simple Sequence" {
  group "Straight Line Sequence"
      #green:A(0);
      :B(0);
      :C(0);
      :B(1);
      :C(1);
      #red:D(0);
  end group
}
@enduml
```



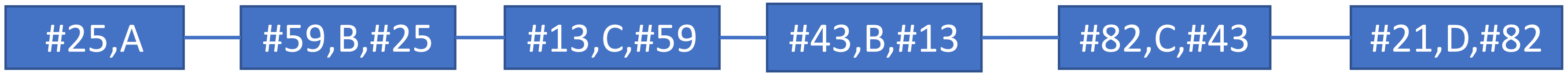
Or

```
@startuml
partition "Job with multiple event occurrences" {
  group "Sequence with AND fork & multiple event occurrences"
      #green:A(0);
      :B(0);
      :C(0);
      fork
        :B(1);
      fork again
        #red:D(0);
        detach
      end fork
      #red:C(1);
  end group
}
@enduml
```



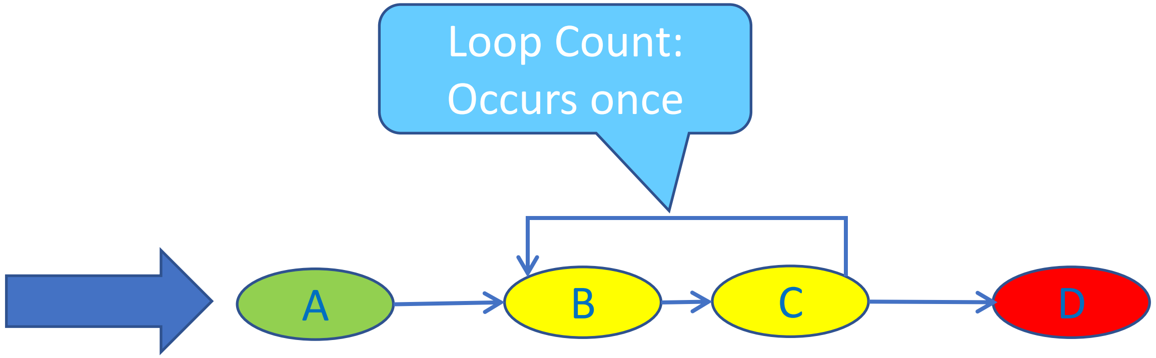
Job with multiple event occurrences
"Sequence with AND fork & multiple event occurrences"
A(0)
B(0)
C(0)
B(1)
D(0)
C(1)

Sequences with repeating events – alternative approach

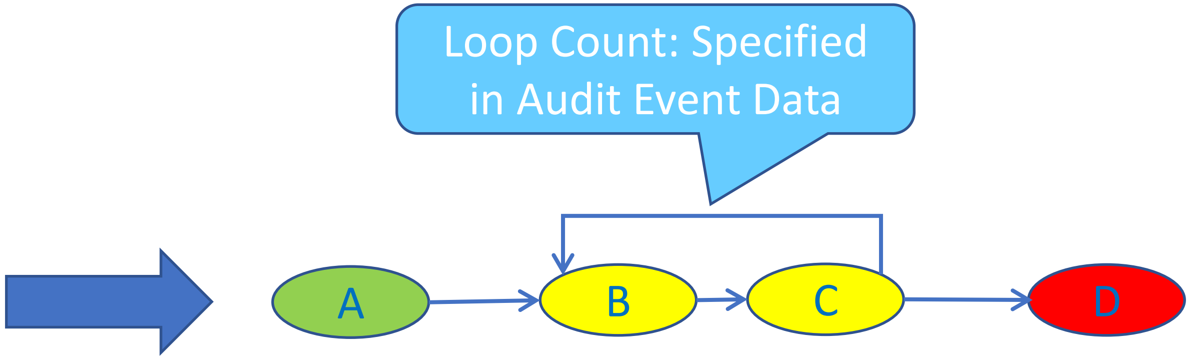


#25,A — #59,B,#25 — #13,C,#59 — #43,B,#13 — #82,C,#43 — #21,D,#82

- An alternative approach is to put a loop in the event mesh
- If the reverse path is constrained then these two topologies are equivalent
- Potential disadvantage is that you would probably only check the constraint had been met at the end of the sequence and so later than in the previous approach
- Major advantage is that the constraint could be dynamic allowing for number of repetitions to be defined at runtime
  - Or even unconstrained



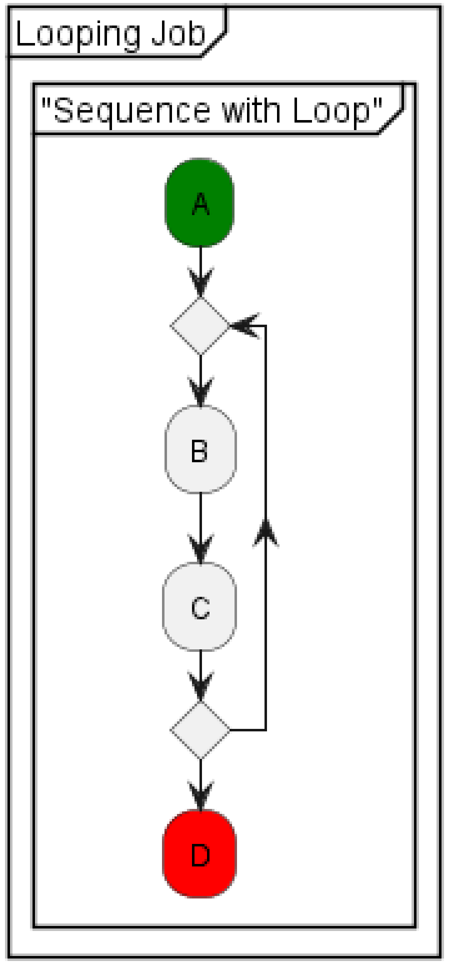
Loop Count: Occurs once



Loop Count: Specified in Audit Event Data

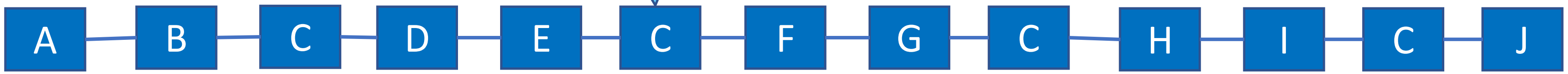
Event loops using PlantUML Activity Diagram

```
@startuml
partition "Looping Job" {
  group "Sequence with Loop"
    #green:A;
    repeat
      :B;
      :C;
    repeat while
    #red:D;
  end group
}
@enduml
```



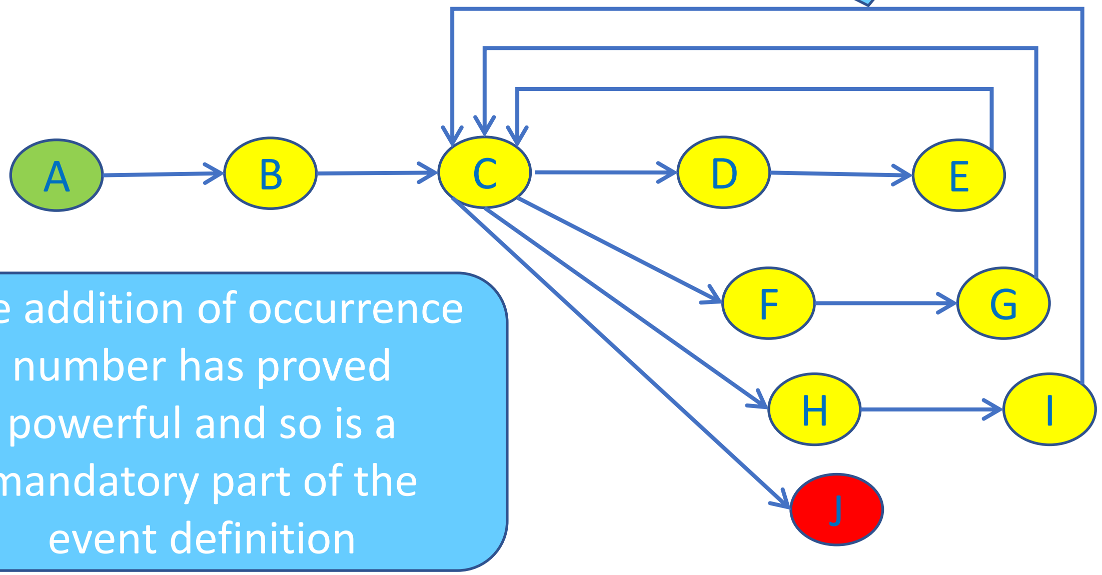
Sequences with repeating events – need both forms

Repeated event

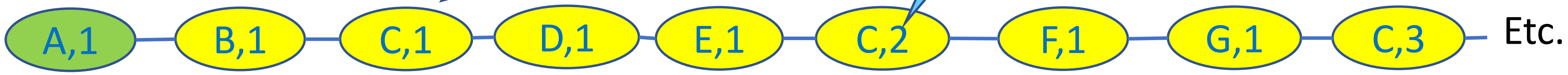


- For this pattern of repetition the loop option is less appealing
  - Each reverse loop could be constrained to happen only once
  - Additional mechanisms would be required to ensure the DE loop occurred before the FG loop etc.
- Using the first approach of C,1 C,2 etc is simpler and stronger

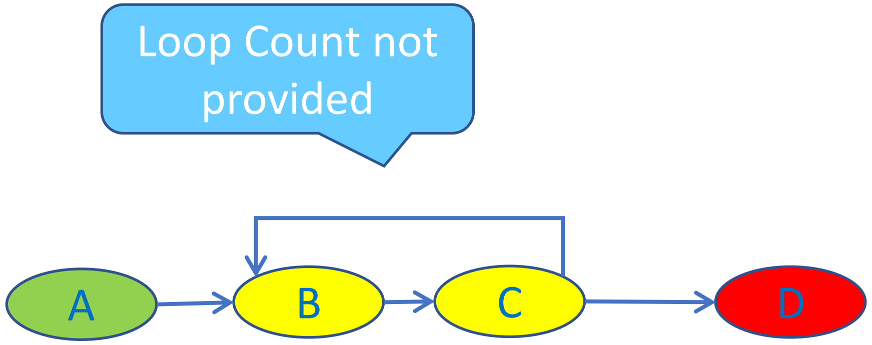
Loop Count: Occurs once



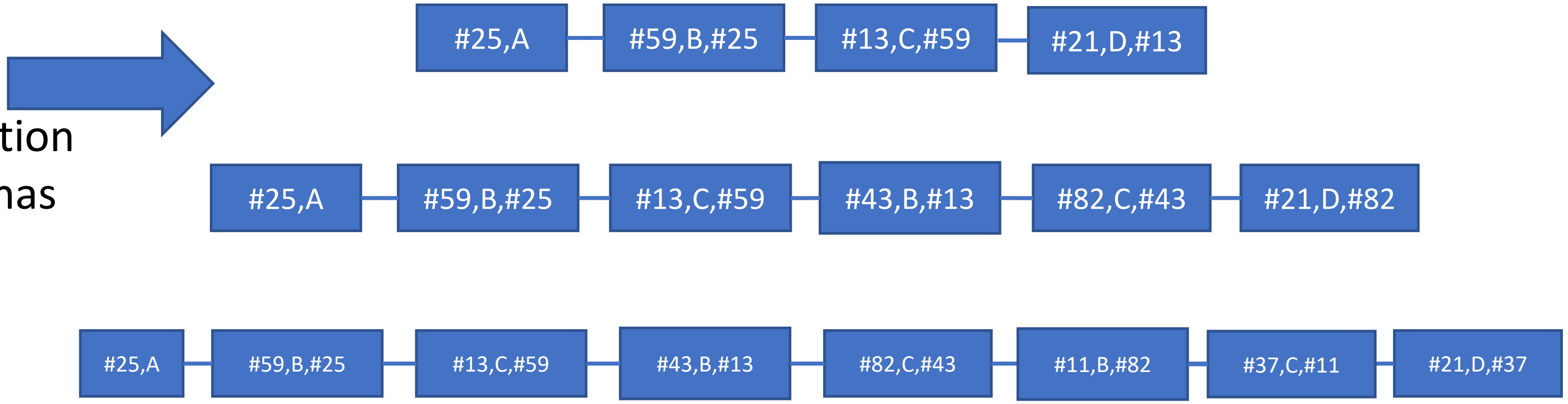
The addition of occurrence number has proved powerful and so is a mandatory part of the event definition



Support for uncounted loop



- This is supported
- These are all valid as a result
- The challenge is setting up the event definition
  - Event B has a previous event C which has not yet been seen
  - Any previous event that is unknown is marked as a placeholder
  - When event C is added it overwrites the placeholder
  - Need to check there are no outstanding placeholders before definition is used
- Note Event Type B is a Type Merge and Event Type C is a Type Fork

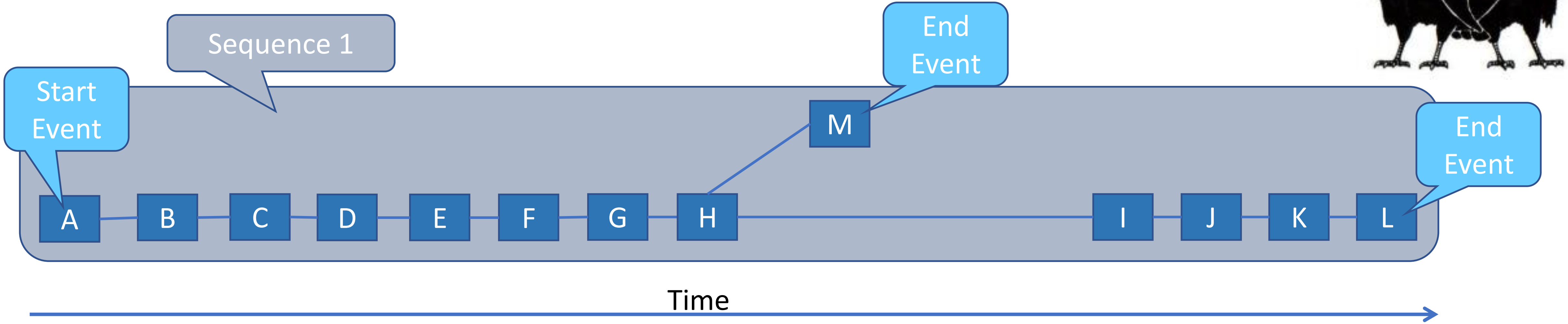


#25,A — #59,B,#25 — #13,C,#59 — #21,D,#13

#25,A — #59,B,#25 — #13,C,#59 — #43,B,#13 — #82,C,#43 — #21,D,#82

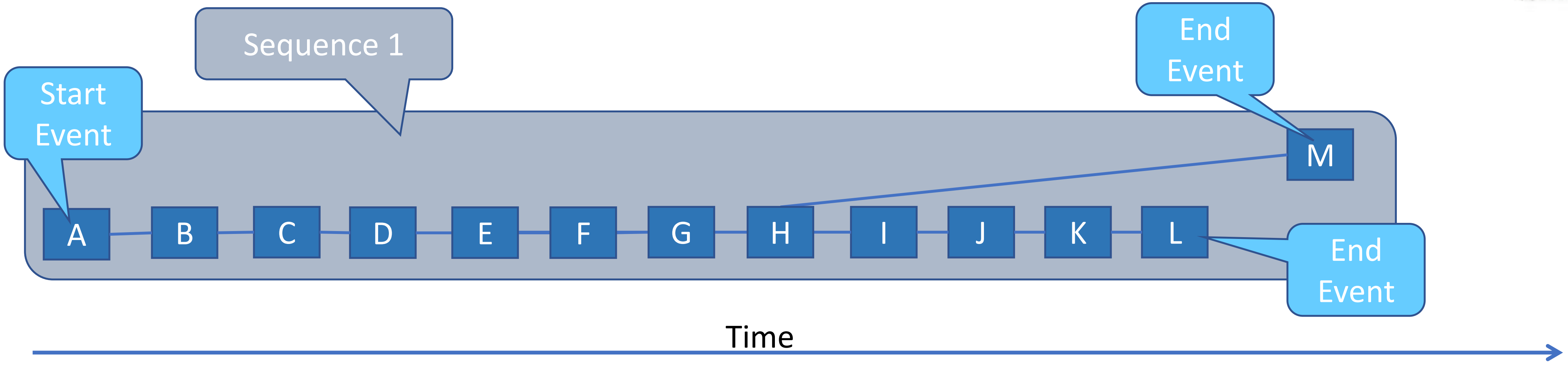
#25,A — #59,B,#25 — #13,C,#59 — #43,B,#13 — #82,C,#43 — #11,B,#82 — #37,C,#11 — #21,D,#37

Issue with Inclusive Or Forks



- Consider the timeline above where H is an Inclusive Or Fork
- When the Digital Twin hits M, I has not occurred and so the sequence has hit an end point with no other branches in progress
    - So the Sequence and potentially the Job look complete
    - Event instances I, J, K, L subsequently arrive for a completed Sequence which looks like an error condition
- I, J, K and L are part of the same sequence so assumed to be coming from the same part of the monitored system
- Hence the currently proposed solution is to wait a short time after M to see if other events in the sequence arrive before deciding the sequence is complete

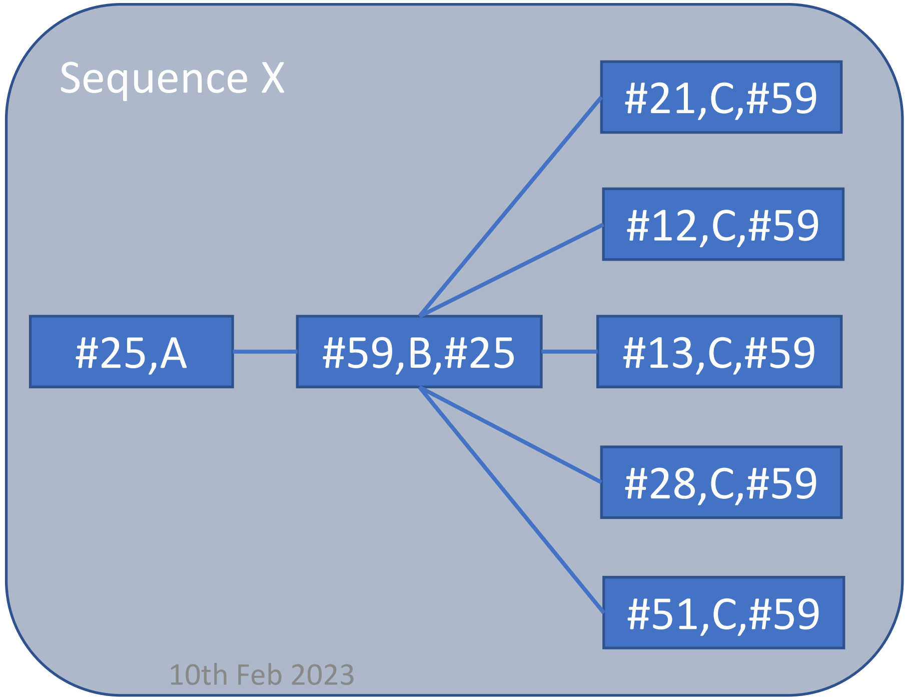
Issue with Inclusive Or Forks 2

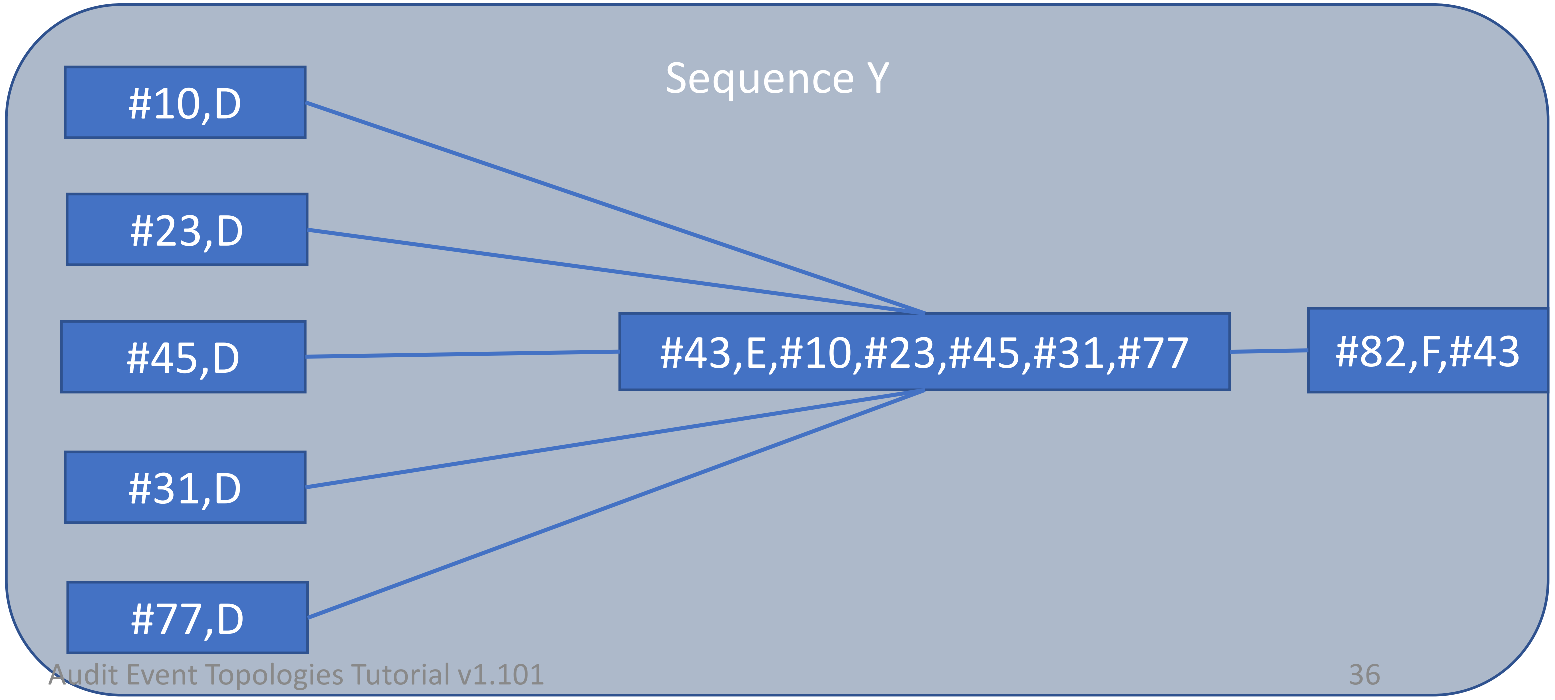


- This same problem occurs if the longer branch finishes first

Sequences with Multiple Start & End Points

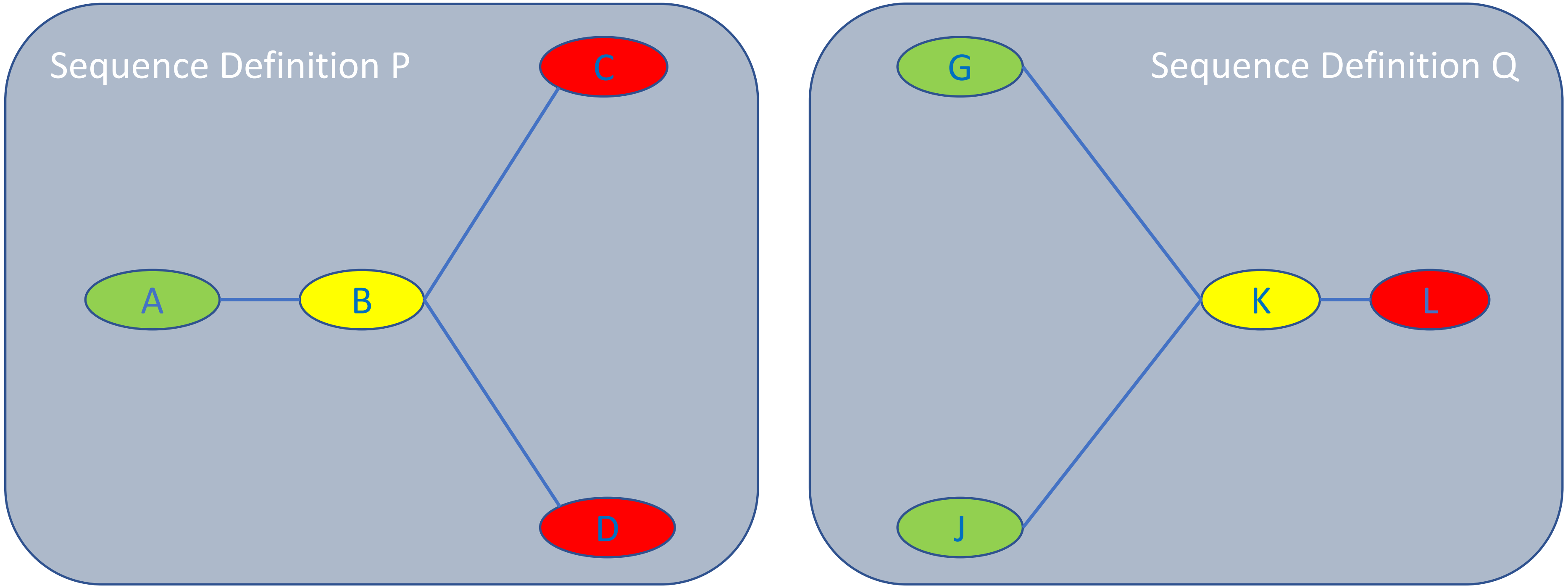
- This situation is supported
- However, it does introduce a vulnerability
  - There is no link between the end events in Sequence X and the start events in Sequence Y
    - Other than Job Id (not shown)
  - If such topologies occur they will need to be supported by the use of invariants (see later)





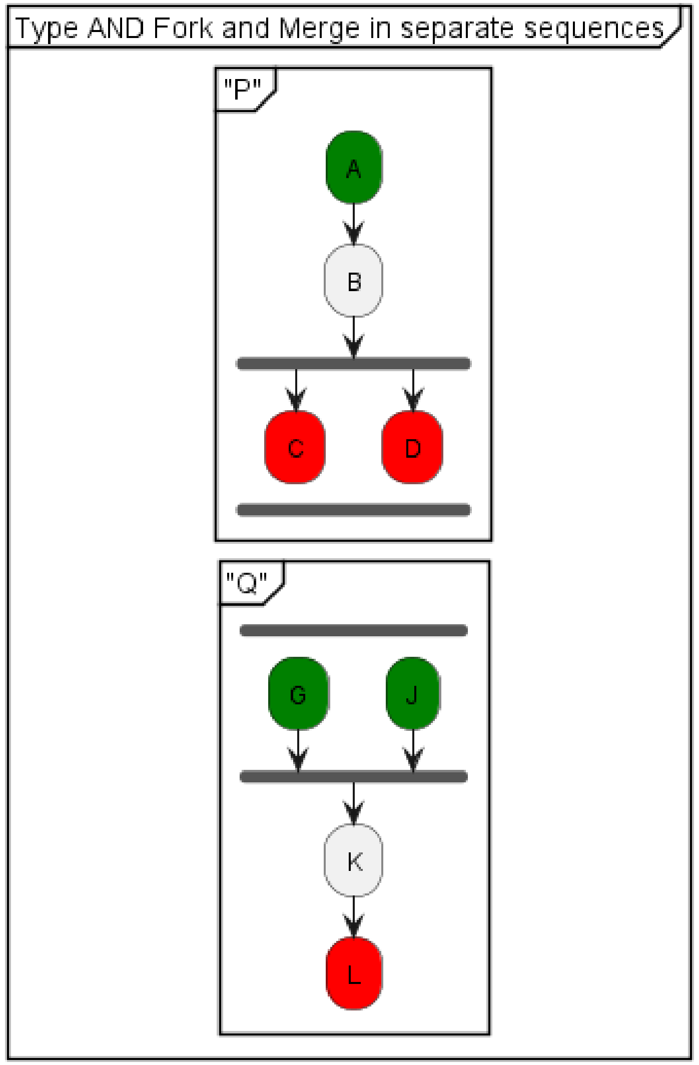
Sequence Definitions with Multiple Start & End Points

- Can they occur?
  - For example, as part of an type fork and merge?
  - How are the event types linked across sequence definition boundaries?
    - How do we know G,1 follows C,1 and not D,1?
  - Does this introduce a vulnerability?



Split Type Fork and Merge using PlantUML Activity Diagram

```
@startuml
partition "Type AND Fork and Merge in separate sequences" {
  group "P"
    #green:A;
    :B;
    fork
      #red:C;
      detach
    fork again
      #red:D;
      detach
    end fork
    -[hidden]->
  end group
  group "Q"
    fork
      -[hidden]->
      #green:G;
    fork again
      -[hidden]->
      #green:J;
    end fork
    :K;
    #red:L;
    detach
  end group
}
@enduml
```



Audit Event Data - Background

- Audit Events may carry supplementary data that can be used to further verify correct behaviour of the monitored system
- Audit Event Data allows dynamic properties to be defined at runtime
- Possible examples include:
  - Audit Event Branch Counts
    - The number of branches following an instance fork can be specified on an audit event before or at the fork
  - Audit Event Loop Counts
    - The number of occurrences of an audit event in a loop can be defined at runtime using audit event data on an event before the loop
  - Invariants in a Job
    - Data associated with one audit event that must match data associated with another audit event later in the same Job
  - Invariants in different Jobs
    - Data associated with one audit event in one Job must match data associated with other audit events later in other Jobs

- This document presents the understanding of the Munin team and is to be used for discussion to clarify and refine the requirements prior to development work
- The conventions used in these slides match those in the Audit Event Topologies material

General Rules for Data carried by Audit Events

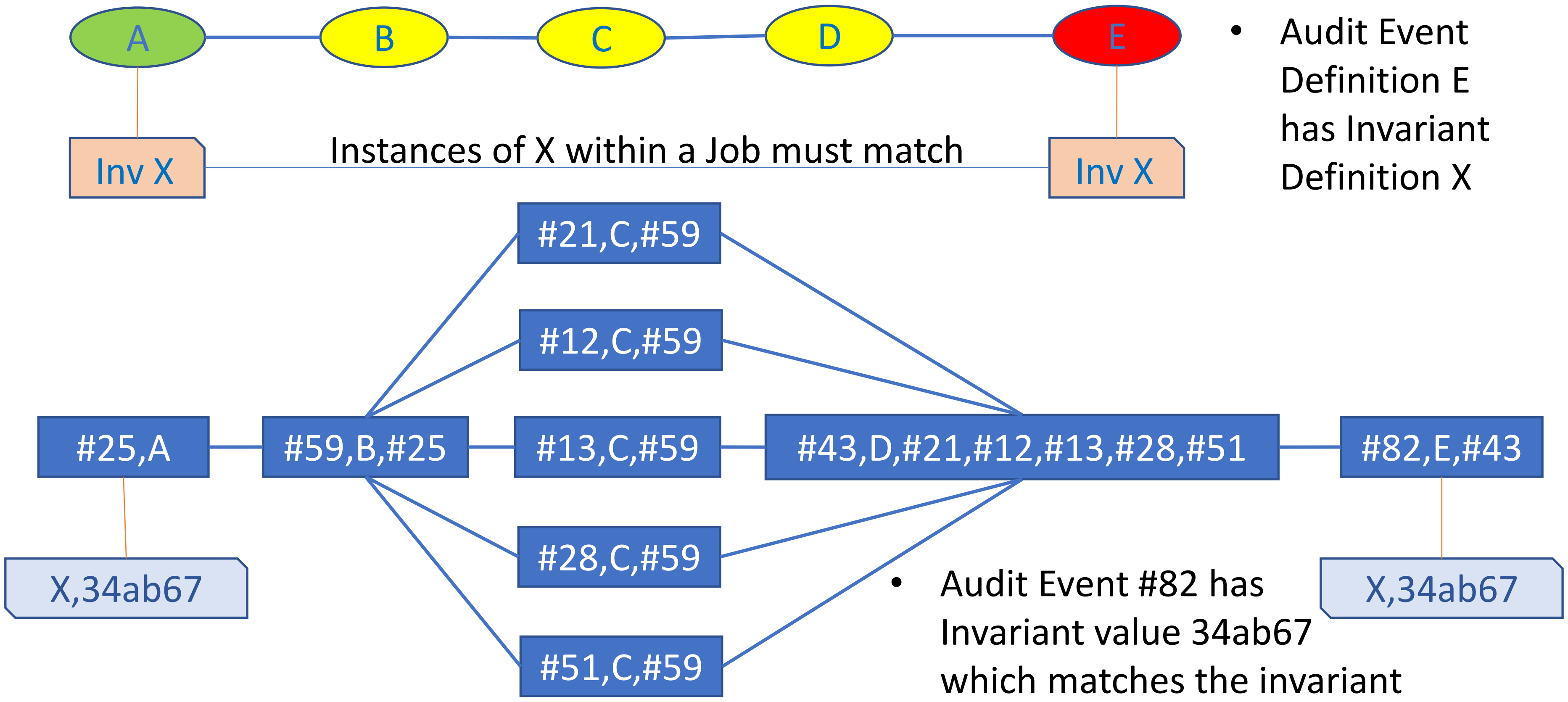
- An item of data that is carried by an audit event must be named
  - i.e. they will occur in name-value pairs or similar
  - Additional a field is required to indicate the kind of audit event data that is represented, e.g.
    - EXTRAJOBINV – invariant between Jobs
    - INTRAJOBINV – invariant within a Job
    - LOOPCOUNT – a dynamic loop count
    - BRANCHCOUNT – a dynamic branch count
- The Digital Twin will maintain knowledge of which types of data will be acceptable with which Audit Event Definitions
  - At the occurrence level since that would support repeated use of audit event types
  - The digital twin will deliberately ignore data associated with an audit event that is not recognised
- Data carried by an Audit Event instance for which there is no corresponding definition shall be treated as an error condition.
- More than one type of data can be carried by an audit event

Invariants

- An item of data carried by one audit event that must match an item of data carried by one or more different audit events
- These can occur in different scopes:.
  - Intra Job - Within a Job though potentially in different sequences within the Job
    - This could be used to link sequences that are not related in any other way than by Job id
  - Extra Job – The value is created in one Job and then the same value is referenced by many Jobs for a defined period of time
    - There is a known example of this type
- In the following examples Invariants of the same type are associated with the different Audit Event Definitions and then matched

Intra-Job Invariants

- Audit Event Definition A has Invariant Definition X



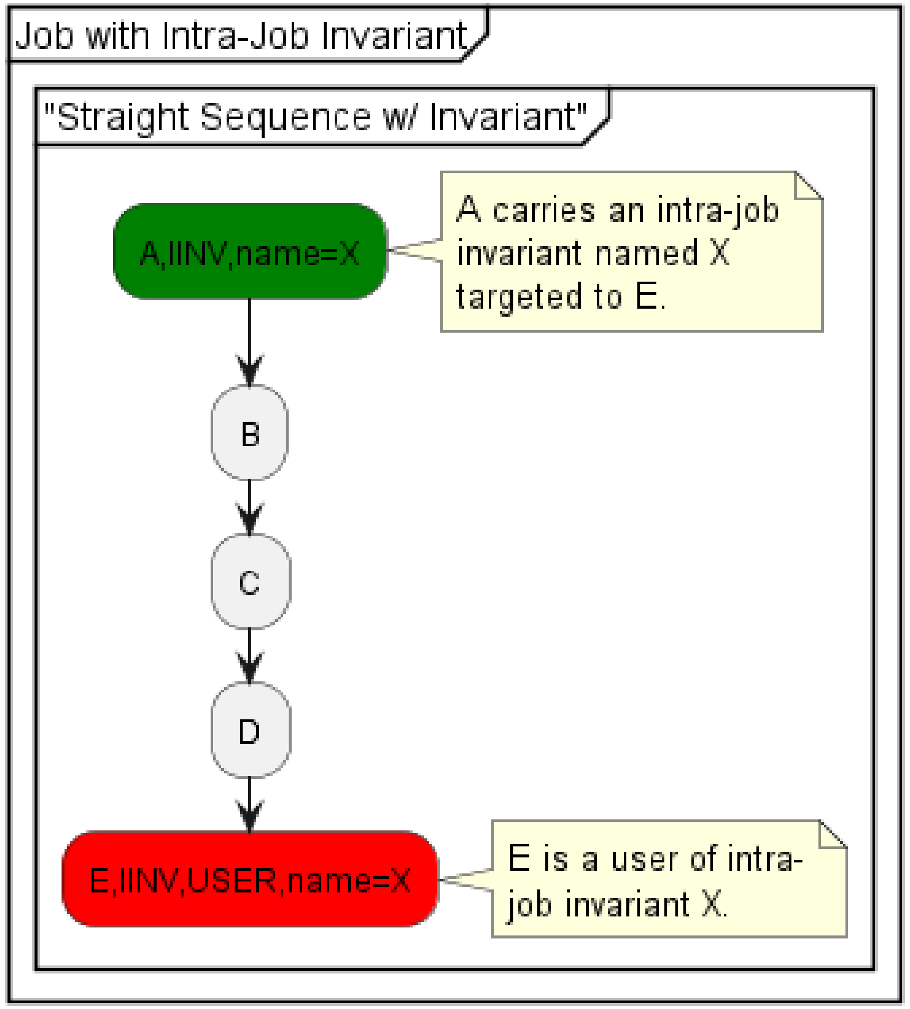
- Audit Event Definition E has Invariant Definition X

- Audit Event #25 has Invariant value 34ab67

- Audit Event #82 has Invariant value 34ab67 which matches the invariant value with Audit Event #25 so the invariant is successful

Intra Job Invariants using PlantUML Activity Diagram

```
@startuml
partition "Job with Intra-Job Invariant" {
  group "Straight Sequence w/ Invariant"
      #green:A,IINV,name=X;
      note right
        A carries an intra-job
        invariant named X
        targeted to E.
      end note
      :B;
      :C;
      :D;
      #red:E,IINV,USER,name=X;
      note right
        E is a user of intra-
        job invariant X.
      end note
  end group
}
@enduml
```



Job with Intra-Job Invariant

"Straight Sequence w/ Invariant"

A,IINV,name=X

A carries an intra-job invariant named X targeted to E.

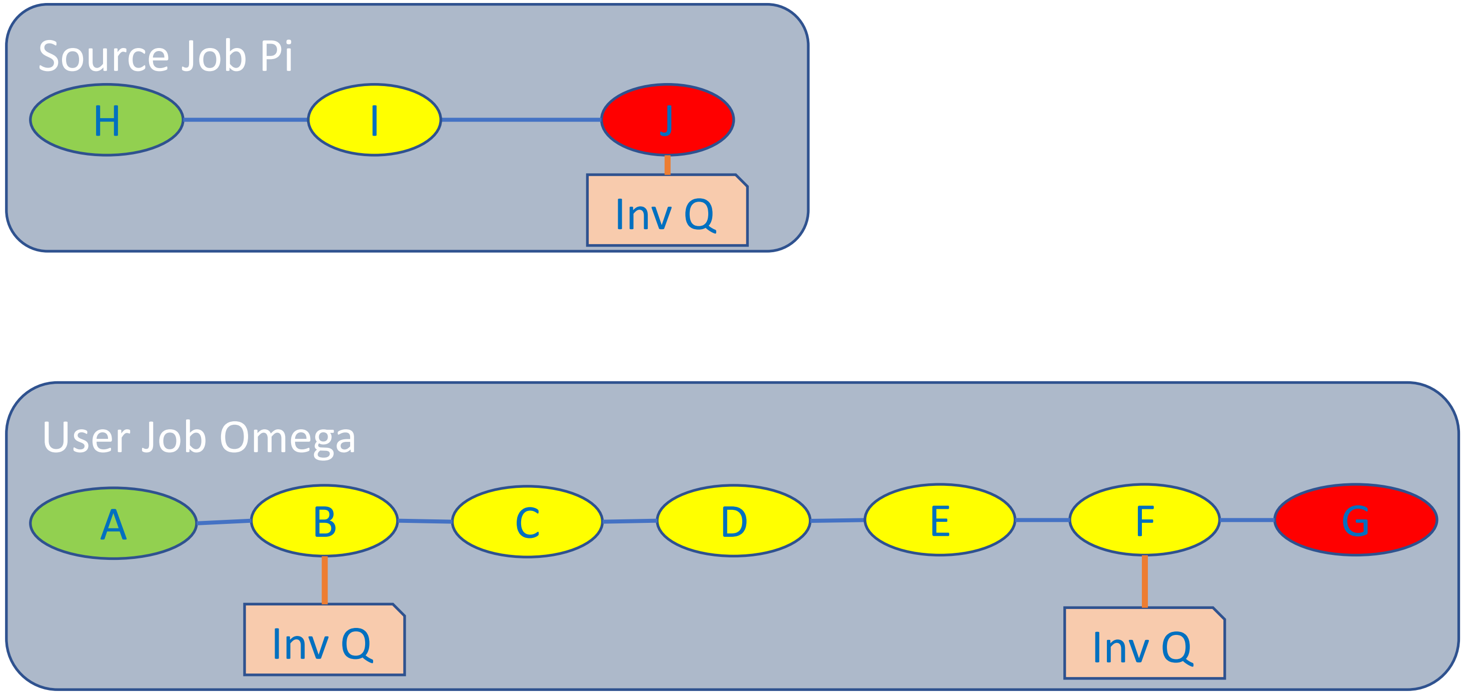
E,IINV,USER,name=X

E is a user of intra-job invariant X.

Intra-Job Invariants

- These must occur in the same Job
- They can occur in different Sequences within the Job
  - This means invariant checking can only be concluded when the Job is completed
  - The invariants may be received in any order (because they can be in different sequences) so they just have to match
- There could be more than 2 matching within a Job
  - Normal sequence verification will determine whether the job is complete in terms of all the expected events so variant checking only has to match the ones it has seen
- There could be more than 1 Invariant within a Job
- There can be only 1 Invariant carried by a single Audit Event
  - This could be relaxed, if necessary, but does add some complexity
- Invariants can be carried by any Audit Event in a Job
- The lifetime of the Invariants is linked to the lifetime of the Job
  - i.e. when the Job is deleted the Invariant is too

Extra-Job Invariants

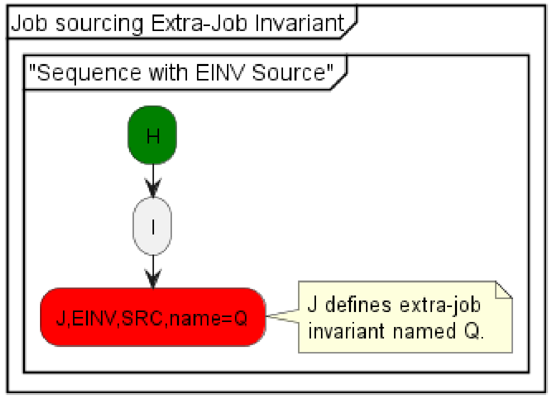


- The Source Job generates a value for the Invariant
- It is then valid for a pre-determined period of time independent of the lifetime of any Job
- Multiple other jobs can then use that value potentially multiple times within each Job
- There is no link between the Jobs
- If multiple instances of the invariant are current at the same time then the using Job is valid with any of them
- The Source Job has to provide the invariant before any of the User Jobs are checked against it

Extra Job Invariants using PlantUML Activity Diagram

```
@startuml
partition "Job sourcing Extra-Job Invariant" {
  group "Sequence with EINV Source"
      #green:H;
      :I;
      #red:J,EINV,SRC,name=Q;
      note right
        J defines extra-job
        invariant named Q.
      end note
  end group
}
@enduml
```

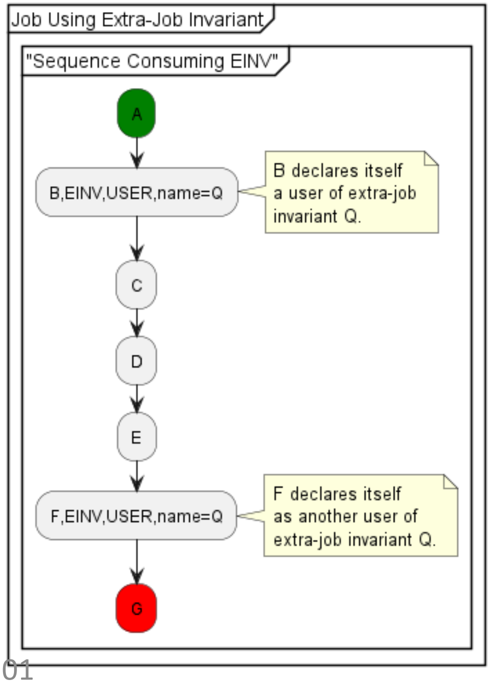
Source Job



Job sourcing Extra-Job Invariant
"Sequence with EINV Source"
H
I
J,EINV,SRC,name=Q
J defines extra-job invariant named Q.

```
@startuml
partition "Job Using Extra-Job Invariant" {
  group "Sequence Consuming EINV"
      #green:A;
      :B,EINV,USER,name=Q;
      note right
        B declares itself
        a user of extra-job
        invariant Q.
      end note
      :C;
      :D;
      :E;
      :F,EINV,USER,name=Q;
      note right
        F declares itself
        as another user of
        extra-job invariant Q.
      end note
      #red:G;
  end group
}
@enduml
```

User Job



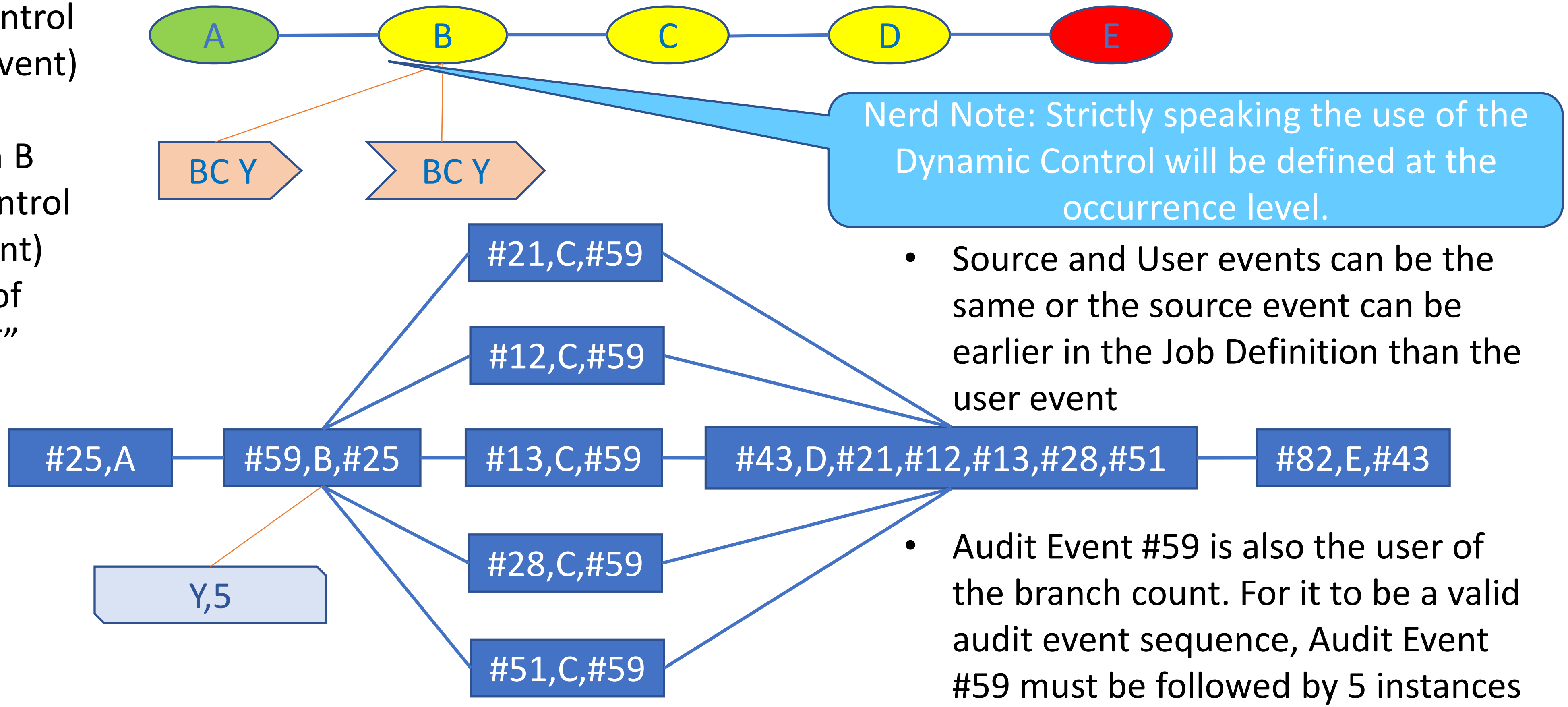
Job Using Extra-Job Invariant
"Sequence Consuming EINV"
A
B,EINV,USER,name=Q
B declares itself a user of extra-job invariant Q.
C
D
E
F,EINV,USER,name=Q
F declares itself as another user of extra-job invariant Q.
G

Extra-Job Invariants

- These Invariants are independent of Jobs
- There must be a single Job Definition and a single Audit Event Definition associated with the creation of all Extra-Job Invariant instances of a given Inter-Job Invariant Definition
  - i.e. an Extra-Job Invariant instance must be 'created' by one audit event in one Job but can be 'referenced’ within many other audit events in other Jobs.
- There could be any number of Jobs that use the Extra-Job Invariant
- There could be more than 1 Invariant associated with a Job
- There can be only 1 Extra-Job Invariant carried by a single Audit Event
  - This could be relaxed, if necessary, but does add some complexity
- Invariants can be carried by any audit event in a Job
- The lifetime of an Extra-Job Invariant is independent of any Job
  - It is a property of the Extra-Job Invariant Definition

Dynamic Control – Branch Count

- Audit Event Definition B is source of Dynamic Control Definition Y (Source Event) and
- Audit Event Definition B also uses Dynamic Control Definition Y (User Event)
- Dynamic Control Y is of type “BRANCHCOUNT”



A — B — C — D — E

BC Y

BC Y

Nerd Note: Strictly speaking the use of the Dynamic Control will be defined at the occurrence level.

#25,A — #59,B,#25

#21,C,#59

#12,C,#59

#13,C,#59

#28,C,#59

#51,C,#59

#43,D,#21,#12,#13,#28,#51 — #82,E,#43

Y,5

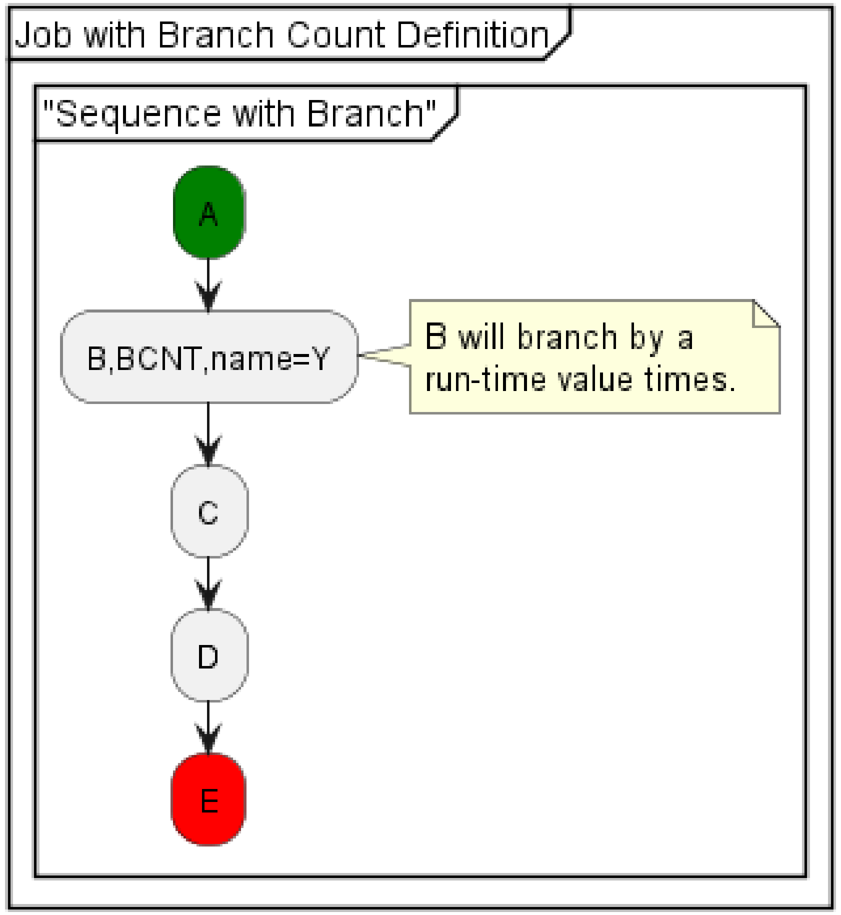
- Source and User events can be the same or the source event can be earlier in the Job Definition than the user event

- Audit Event #59 is also the user of the branch count. For it to be a valid audit event sequence, Audit Event #59 must be followed by 5 instances of Audit Event of type C

- The source event, Audit Event #59 has branch count value 5 which means that the user event will be a fork with 5 branches

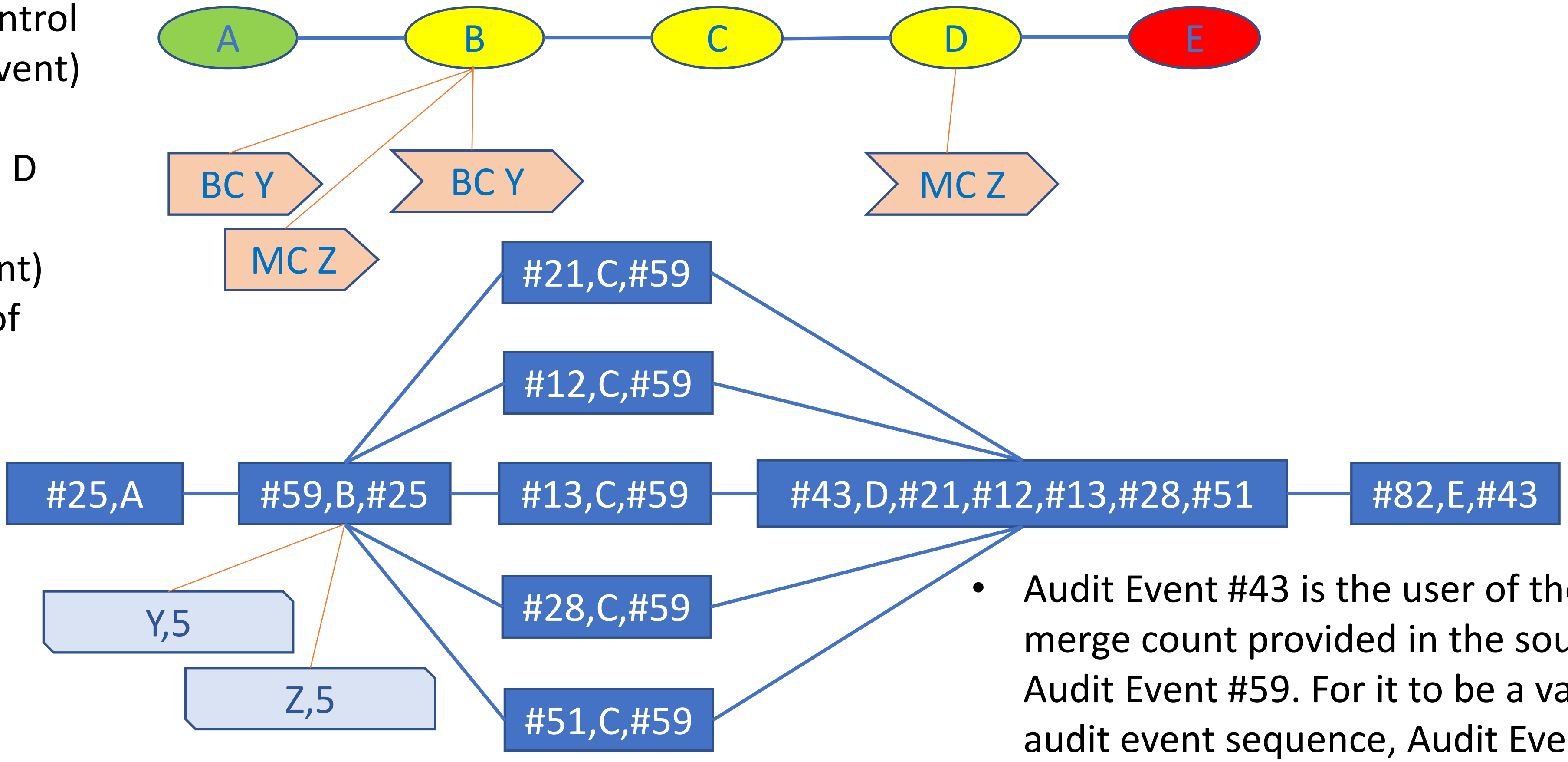
Branch Count using PlantUML Activity Diagram

```
@startuml
partition "Job with Branch Count Definition" {
  group "Sequence with Branch"
      #green:A;
      :B,BCNT,name=Y;
      note right
        B will branch by a
        run-time value times.
      end note
      :C;
      :D;
      #red:E;
  end group
}
@enduml
```



Dynamic Control – Adding Merge Count

- Audit Event Definition B is source of Dynamic Control Definition Z (Source Event) and
- Audit Event Definition D uses Dynamic Control Definition Z (User Event)
- Dynamic Control Z is of type "MERGECOUNT"

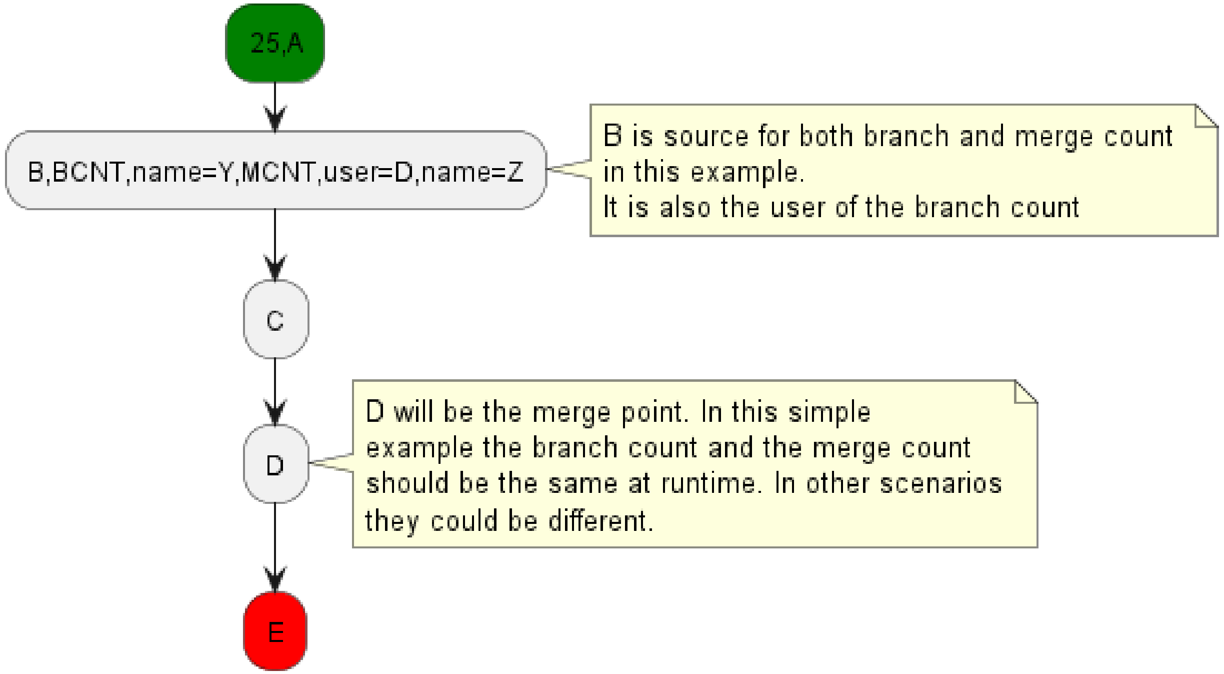


- The source event, Audit Event #59 has merge count value 5 which means that the user event #43 is expected to be a merge of 5 branches

- Audit Event #43 is the user of the merge count provided in the source Audit Event #59. For it to be a valid audit event sequence, Audit Event #43 must have 5 previous audit event ids

Branch and Merge Count using PlantUML Activity Diagram

```
@startuml
partition "Job with Branch and Merge Count Definition" {
  group "Sequence with Branch and Merge"
      #green:A;
      :B,BCNT,name=Y,MCNT,user=D,name=Z;
      note right
        B will branch by a
        run-time value times.
      end note
      :C;
      :D;
      note right
        D will be the merge point. In this simple
        example the branch count and the merge count
        should be the same at runtime. In other scenarios
        they could be different.
      end note
      #red:E;
  end group
}
@enduml
```



B is source for both branch and merge count
in this example.
It is also the user of the branch count

D will be the merge point. In this simple
example the branch count and the merge count
should be the same at runtime. In other scenarios
they could be different.

Branch Count

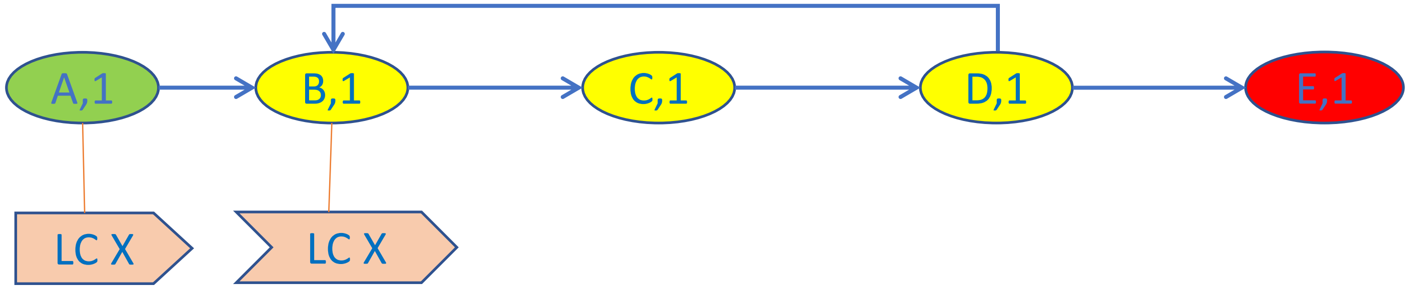
- The branch count source event can be any event in the sequence prior to or including the fork point itself
  - Provided that the user event type is unambiguous in the sequence after the source event
    - i.e. avoid having the same event type (irrespective of occurrence number) on different branches after the source event
- The branch count is evaluated when the Job is deemed to be complete
- The source event sets the expected branch count value
- Each event that carries the fork event as its previous event id increments the current branch count
- At Job completion the current branch count is compared with the expected branch count
  - If they match the branch count is valid
  - If they don’t the Job has failed

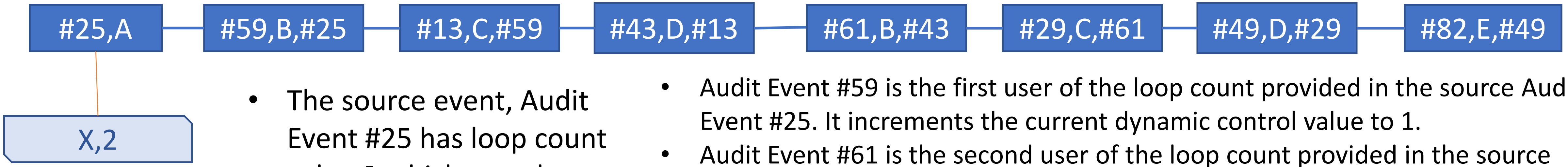
Merge Count

- The merge count source event can be any single event in the sequence prior to the merge point
  - Provided that the user event type is unambiguous in the sequence after the source event
    - i.e. avoid putting the merge count on an event in the parallel branches
- The merge count is evaluated when it is received
  - Since it carries the complete set of previous event ids
- The source event sets the expected merge count value
- The user event compares the number of previous event ids it received with the expected value of the merge count
  - If they match the merge count is valid
  - If they don’t the Job has failed

Dynamic Control – Loop Count

- Audit Event Definition A is source of Dynamic Control Definition X (Source Event) and
- Audit Event Definition B uses Dynamic Control Definition X (User Event)
- Dynamic Control X is of type “LOOPCOUNT”





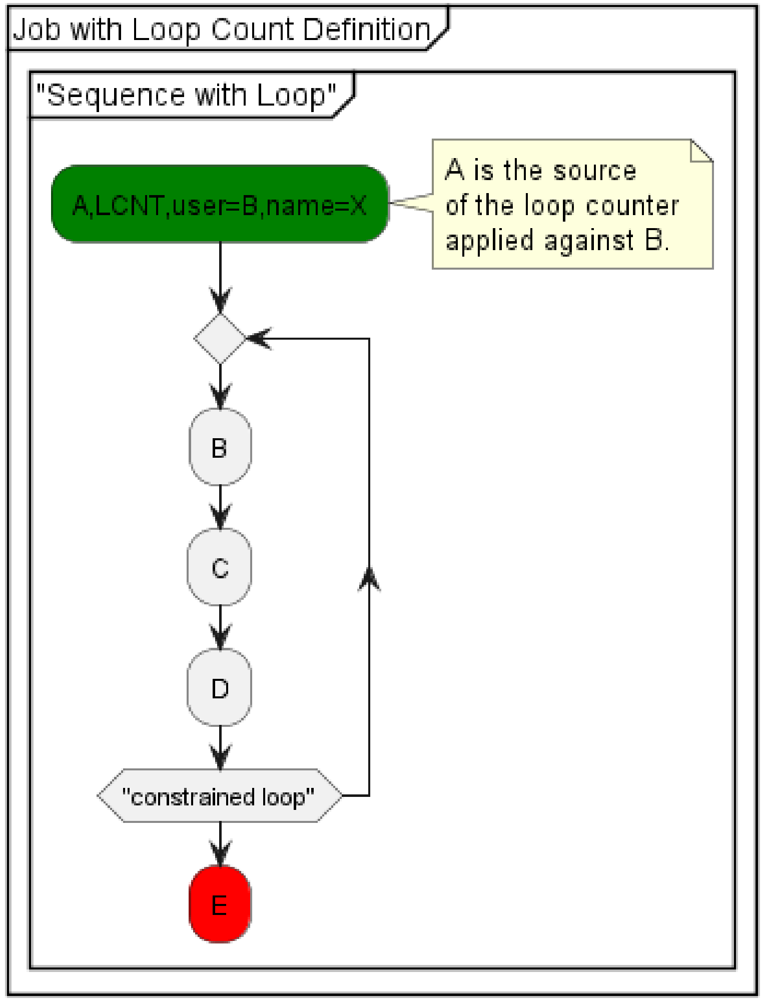
- The source event, Audit Event #25 has loop count value 2 which sets the expected dynamic control value to 2

- Audit Event #59 is the first user of the loop count provided in the source Audit Event #25. It increments the current dynamic control value to 1.
- Audit Event #61 is the second user of the loop count provided in the source Audit Event #25. It increments the current dynamic control value to 2.
- For it to be a valid audit event sequence, the current dynamic control value must equal the expected dynamic control value at the end of the Job
- The instances in this example would pass the test

A,B,… Audit Event Type

Loop Count using PlantUML Activity Diagram

```
@startuml
partition "Job with Loop Count Definition" {
  group "Sequence with Loop"
    #green:A,LCNT,user=B,name=X;
    note right
      A is the source
      of the loop counter
      applied against B.
    end note
    repeat
      :B;
      :C;
      :D;
    repeat while ("constrained loop")
    #red:E;
end group
}
@enduml
```



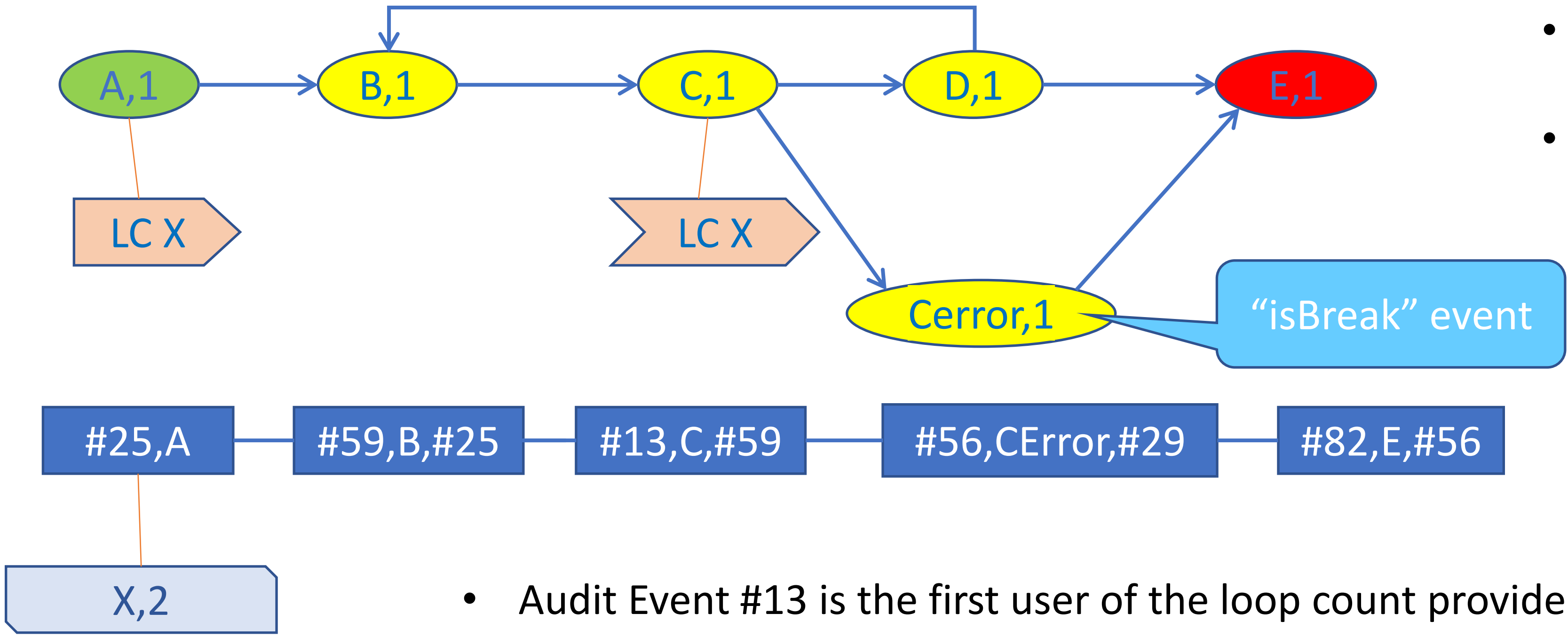
Loop Count

- The loop count source event can be any event in the sequence prior to the loop
- The loop count is evaluated when the Job is deemed to be complete
- The source event sets the expected loop count value
- Each time the user event is seen in the loop the current loop count is incremented
- At Job completion the current loop count is compared with the loop branch count
  - If they match the loop count is valid
  - If they don’t the Job has failed

Dynamic Controls

- A Dynamic Control must be created and used within a single Job
- 1 Audit Event Definition will be defined as the source of a given Dynamic Control and 1, and only 1, Audit Event Definition will be defined as the user of that Dynamic Control
- The Dynamic Control Definition must have a name and a meaning
- The supported set of meanings are:
  - BRANCHCOUNT (for constraining number of branches at instance forks)
  - MERGECOUNT (for constraining number of branches coming together at a merge point)
  - LOOPCOUNT (for constraining the number of times around a loop)
- The Dynamic Control carries a value of type Integer
- There could be more than 1 dynamic control associated with a Job Definition
  - They must have unique names within the Job Definition
- A single Audit Event can be the source of multiple different dynamic controls
- The source of a dynamic control must always precede its usage
  - Which can only be enforced if they are in the same Sequence within a Job
  - For BRANCHCOUNT source and user events can be the same
- The lifetime of the dynamic control is that of the Job or more strictly from source to usage
- Other dynamic controls may be required e.g.
  - next_audit_event_names (for constraining which paths can be taken at a type fork, plural form would mean multiple branches are expected)
  - The type of the value is a Sequence of Audit Event Definition Name for next_audit_event_names

Job Topology – Loop Break



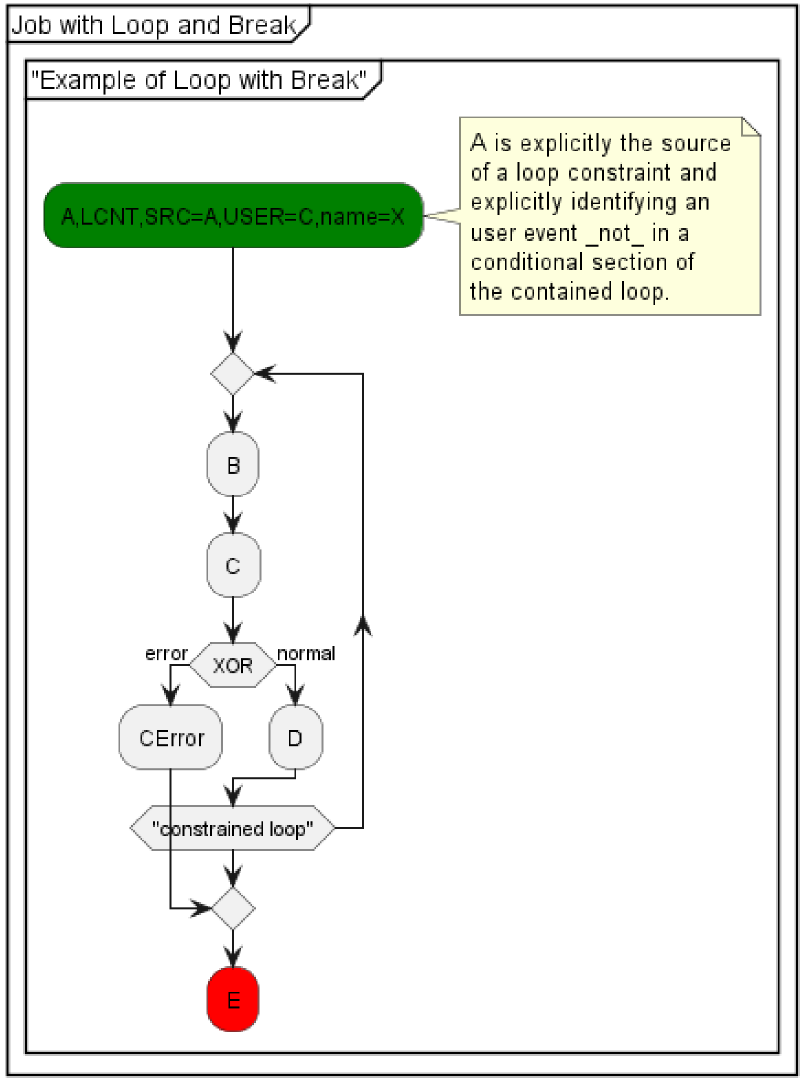
- The “isBreak” event can follow any event in the loop
- It doesn’t have to be the same event as the user event for the dynamic control

- Audit Event #13 is the first user of the loop count provided in the source Audit Event #25. It increments the current dynamic control value to 1.
- Audit Event #56 is an “isBreak” event which denotes it is a valid event that causes an exit from the loop . It clears the expected dynamic control value.
- On Job completion the expected dynamic control value is not set so the loop count is not checked. Therefore this is a valid sequence.

A,B,… Audit Event Type

Loop Count with Break using PlantUML Activity Diagram

```
@startuml
partition "Job with Loop and Break" {
  group "Example of Loop with Break"
    #green:A,LCNT,SRC=A,USER=C,name=X;
    note right
      A is explicitly the source
      of a loop constraint and
      explicitly identifying an
      user event _not_ in a
      conditional section of
      the contained loop.
    end note
    repeat
      :B;
      :C;
      if (XOR) then (error)
        :CError;
        break
      else (normal)
        :D;
      endif
    repeat while ("constrained loop")
    #red:E;
  end group
}
@enduml
```



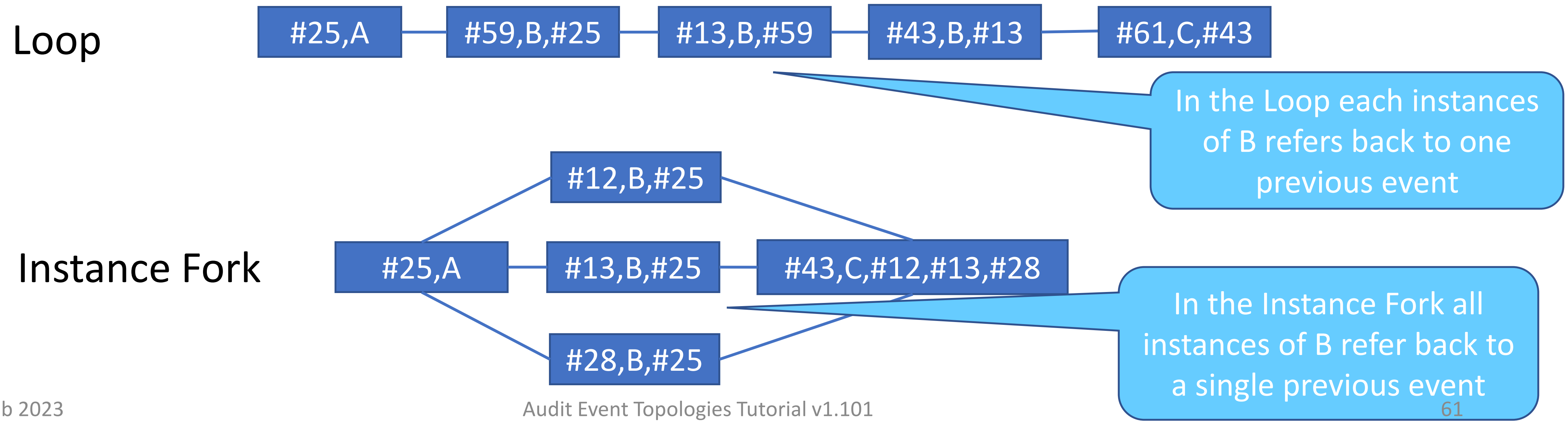
A is explicitly the source of a loop constraint and explicitly identifying an user event _not_ in a conditional section of the contained loop.

Break from a Loop

- If an error occurs within a loop then the loop may be exited prematurely
- The audit event definition that marks the exit from the loop is marked as an “isBreak” event
- The Job Definition must include the isBreak event on one of its allowed paths
- When an isBreak event is encountered it cancels the loop count so no error can occur from a mismatch between current and expected dynamic control values.

Loop vs Instance Fork

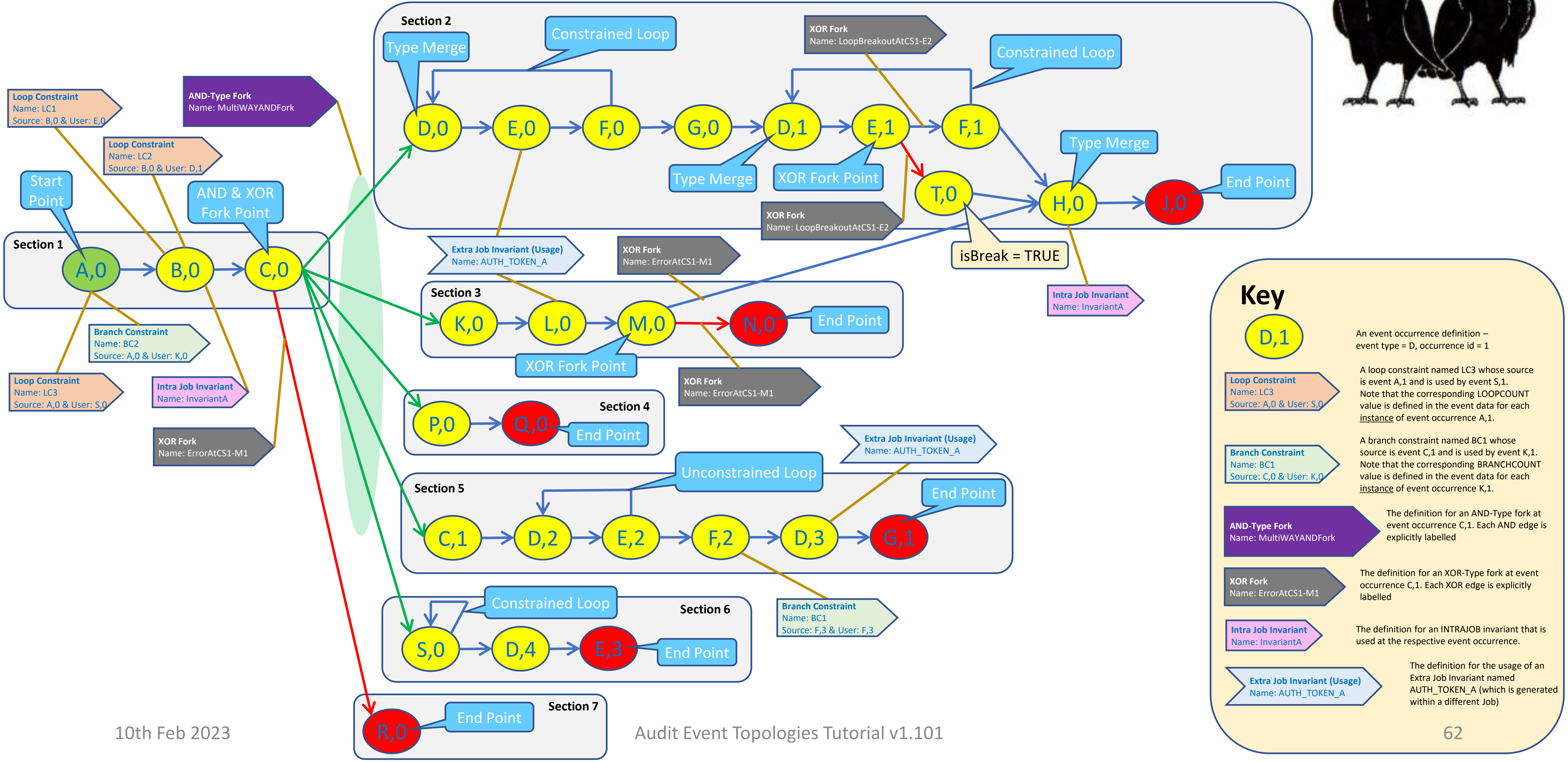
- What’s the difference between a loop and an instance fork?
- If I expect event A followed by 3 instances of event B and then event C do I use a loop or do I use an instance fork – spot the difference:



In the Loop each instances of B refers back to one previous event

In the Instance Fork all instances of B refer back to a single previous event

‘ComplexJob’ Definition – Single Sequence with multiple ‘sections’



Section 1

Start Point

A,0

B,0

C,0

AND & XOR Fork Point

Loop Constraint
Name: LC1
Source: B,0 & User: E,0

Loop Constraint
Name: LC2
Source: B,0 & User: D,1

AND-Type Fork
Name: MultiWAYANDFork

Branch Constraint
Name: BC2
Source: A,0 & User: K,0

Loop Constraint
Name: LC3
Source: A,0 & User: S,0

Intra Job Invariant
Name: InvariantA

XOR Fork
Name: ErrorAtCS1-M1

Section 2

Type Merge

Constrained Loop

D,0

E,0

F,0

G,0

D,1

E,1

F,1

Type Merge

XOR Fork Point

XOR Fork
Name: LoopBreakoutAtCS1-E2

Constrained Loop

XOR Fork
Name: LoopBreakoutAtCS1-E2

T,0

isBreak = TRUE

Type Merge

H,0

J,0

End Point

Intra Job Invariant
Name: InvariantA

Extra Job Invariant (Usage)
Name: AUTH_TOKEN_A

XOR Fork
Name: ErrorAtCS1-M1

Section 3

K,0

L,0

M,0

N,0

End Point

XOR Fork Point

XOR Fork
Name: ErrorAtCS1-M1

Section 4

P,0

Q,0

End Point

Extra Job Invariant (Usage)
Name: AUTH_TOKEN_A

Unconstrained Loop

Section 5

End Point

C,1

D,2

E,2

F,2

D,3

G,1

Branch Constraint
Name: BC1
Source: F,3 & User: F,3

Constrained Loop

Section 6

S,0

D,4

E,3

End Point

Section 7

R,0

End Point

Key

D,1

An event occurrence definition – event type = D, occurrence id = 1

Loop Constraint
Name: LC3
Source: A,0 & User: S,0

A loop constraint named LC3 whose source is event A,1 and is used by event S,1. Note that the corresponding LOOPCOUNT value is defined in the event data for each instance of event occurrence A,1.

Branch Constraint
Name: BC1
Source: C,0 & User: K,0

A branch constraint named BC1 whose source is event C,1 and is used by event K,1. Note that the corresponding BRANCHCOUNT value is defined in the event data for each instance of event occurrence K,1.

AND-Type Fork
Name: MultiWAYANDFork

The definition for an AND-Type fork at event occurrence C,1. Each AND edge is explicitly labelled

XOR Fork
Name: ErrorAtCS1-M1

The definition for an XOR-Type fork at event occurrence C,1. Each XOR edge is explicitly labelled

Intra Job Invariant
Name: InvariantA

The definition for an INTRAJOB invariant that is used at the respective event occurrence.

Extra Job Invariant (Usage)
Name: AUTH_TOKEN_A

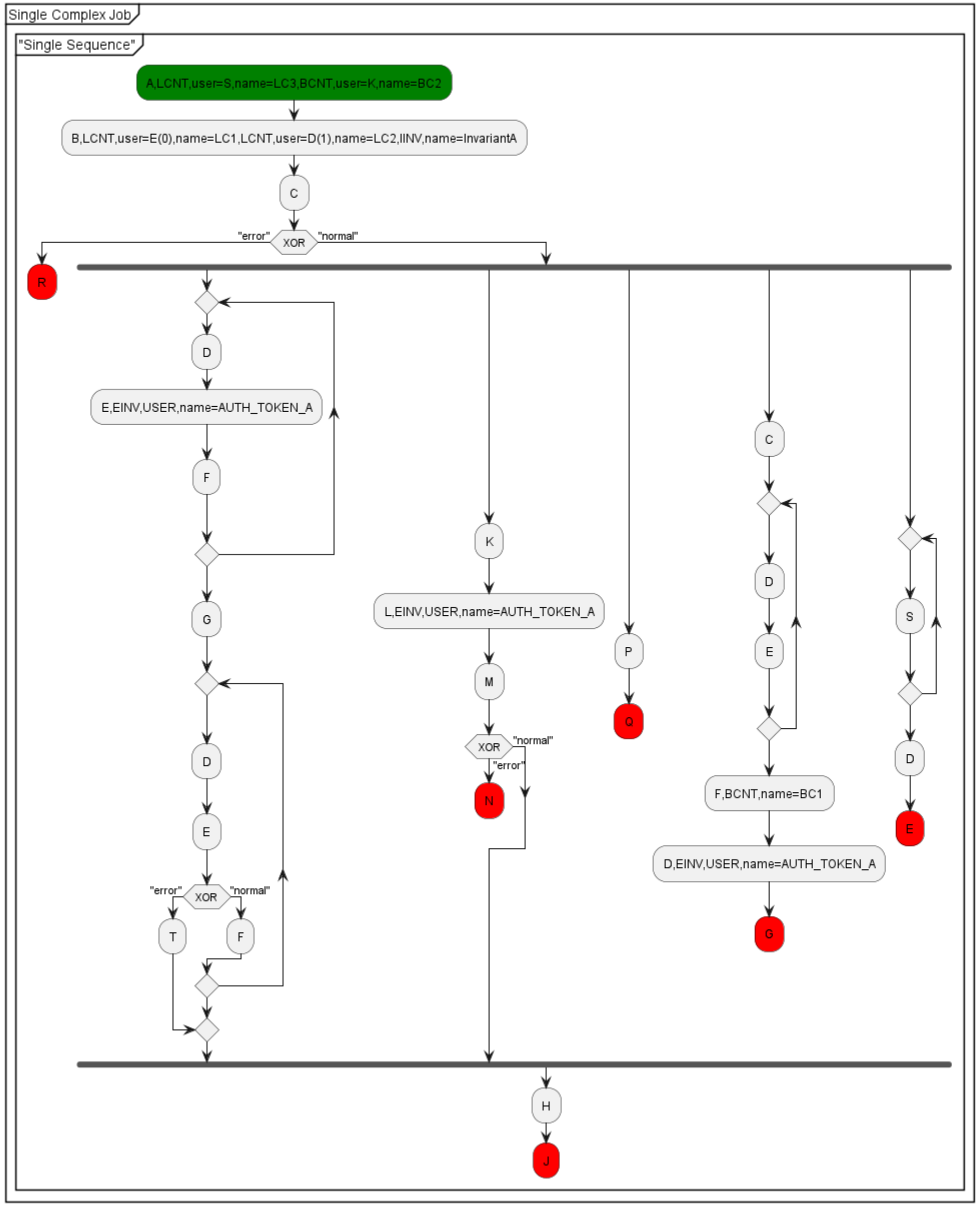
The definition for the usage of an Extra Job Invariant named AUTH_TOKEN_A (which is generated within a different Job)

‘ComplexJob’ Definition – PlantUML Text

```
@startuml
partition "Single Complex Job" {
group "Single Sequence"

  #green:A,LCNT,user=S,name=LC3,BCNT,user=K,name=BC2;
  :B,LCNT,user=E(0),name=LC1,LCNT,user=D(1),name=LC2,IINV,name=InvariantA;
  :C;
  if (XOR) then ("error")
    #red:R;
    detach
  else ("normal")
    fork
      repeat
        :D;
        :E,EINV,USER,name=AUTH_TOKEN_A;
        :F;
      repeat while
      :G;
      repeat
        :D;
        :E;
        if (XOR) then ("error")
          :T;
          break
        else ("normal")
          :F;
        endif
      repeat while
    fork again
      :K;
      :L,EINV,USER,name=AUTH_TOKEN_A;
      :M;
      if (XOR) then ("error")
        #red:N;
        detach
      else ("normal")
      endif
 fork again
      :P;
      #red:Q;
      detach
    fork again
      :C;
      repeat
        :D;
        :E;
      repeat while
      :F,BCNT,name=BC1;
      :D,EINV,USER,name=AUTH_TOKEN_A;
      #red:G;
      detach
    fork again
      repeat
        :S;
      repeat while
      :D;
      #red:E;
      detach
    end fork
    :H;
    #red:J;
    detach
  endif
end group
}
@enduml
```

‘ComplexJob’ Definition – Activity Diagram


Single Complex Job
"Single Sequence"
A,LCNT,user=S,name=LC3,BCNT,user=K,name=BC2
B,LCNT,user=E(0),name=LC1,LCNT,user=D(1),name=LC2,IINV,name=InvariantA
C
"error"
XOR
"normal"
R
D
E,EINV,USER,name=AUTH_TOKEN_A
F
G
D
E
"error"
XOR
"normal"
T
F
K
L,EINV,USER,name=AUTH_TOKEN_A
M
XOR
"normal"
"error"
N
P
Q
C
D
E
F,BCNT,name=BC1
D,EINV,USER,name=AUTH_TOKEN_A
G
S
D
E
H
J

Observations

- Inclusive OR has no clear requirement and it is recommended that it should be dropped
  - It adds complexity to determining Job completion
- Loop counts and branch counts are currently optional
  - If they are omitted from the definition the loops and branches can occur but will not be tested
  - This reduces the verification that the Sequence Verifier is performing
  - Mandating branch counts on instance forks can be supported and is recommended
  - Mandating loop counts could be considered
- Merge counts have been introduced to constrain the allowed runtime options
  - These are currently optional but could be mandated