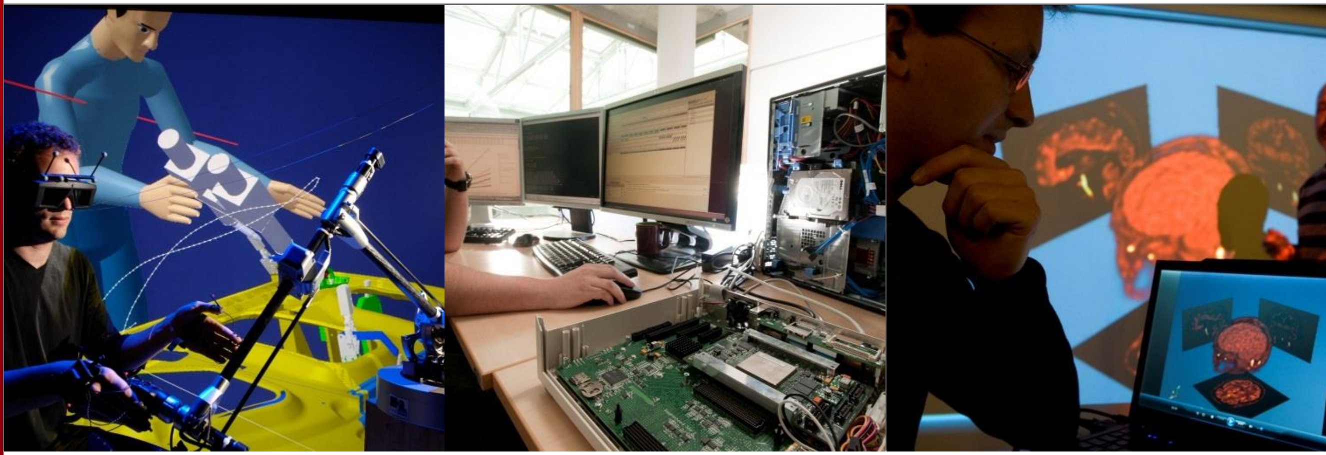
ACTIVITY MODELING

Introduction on UML for Industrial Systems

Jérémie Tatibouët, Shuai Li, François Terrier, Sébastien Gérard, **Asma Smaoui**

{first_name}.{last_name}@cea.fr



CONTENT

FROM RESEARCH TO INDUSTRY
cea tech

CONTENT

list

WHAT IS THE PURPOSE OF ACTIVITIES?

Based on UML 2.5

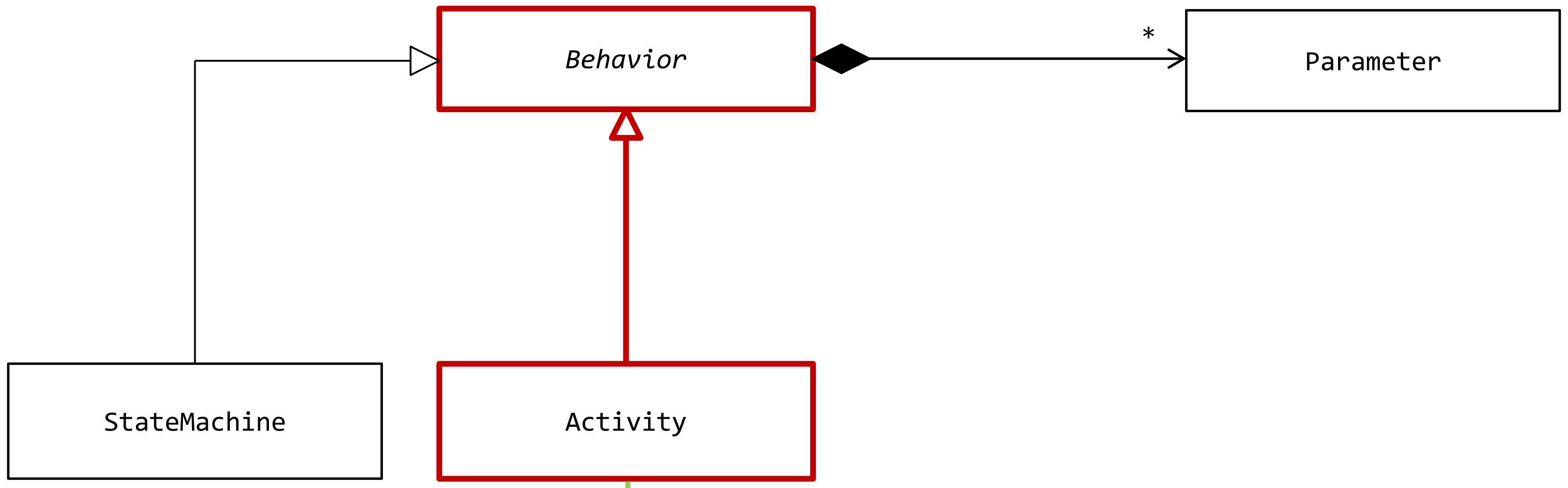
“Activities may describe procedural computation…”

“…in an object-oriented model, they may be invoked indirectly as methods bound to Operations that are directly invoked…”.

“Activities can also be used for information system modeling to specify system level processes…”

ACTVITY USAGES ALLOWED BY UML (1/3)

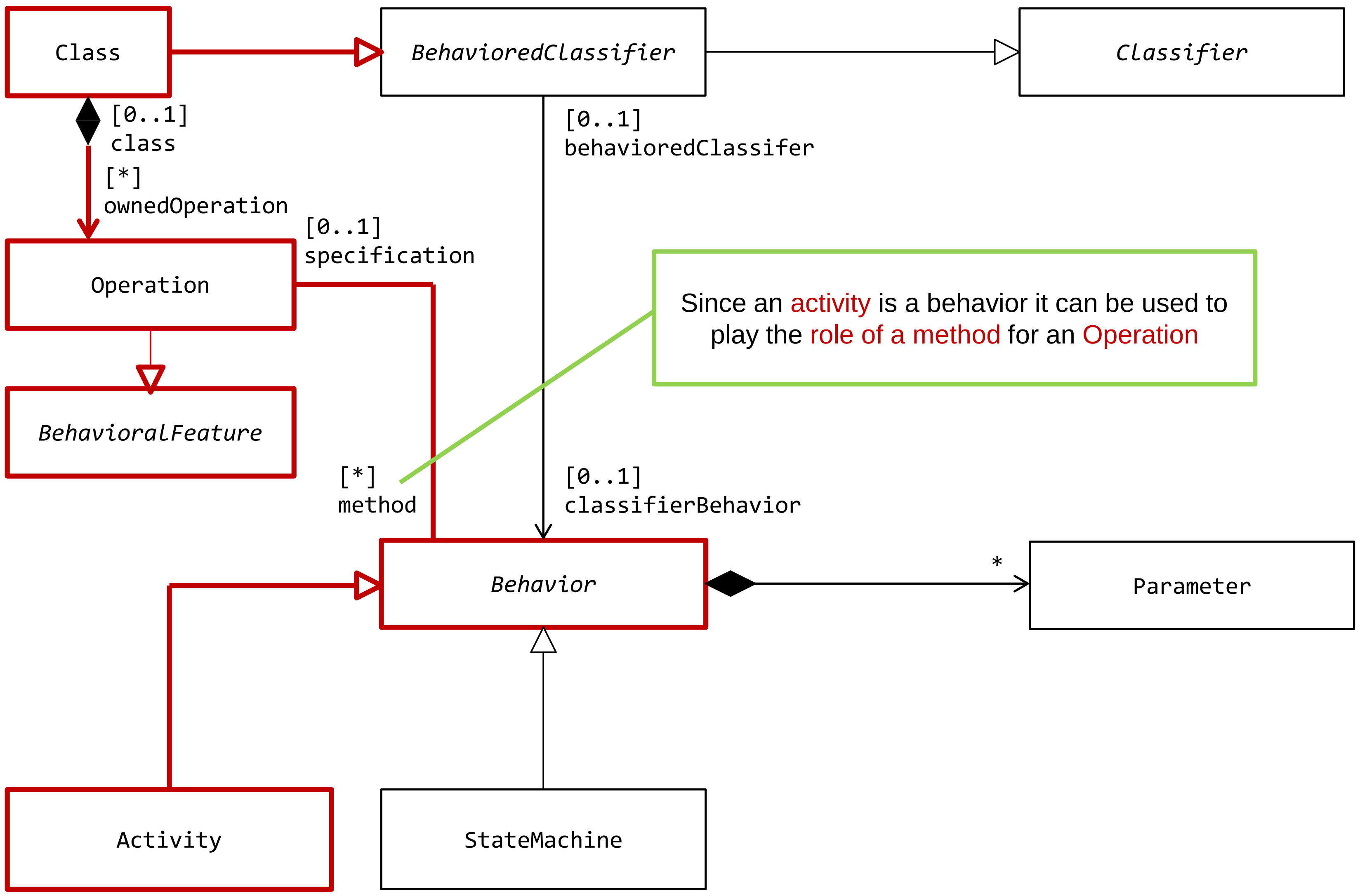
IN STANDALONE MODE



An activity is a specific kind of behavior that can be used without context classifier. When used in standalone mode, the activity is its own context.

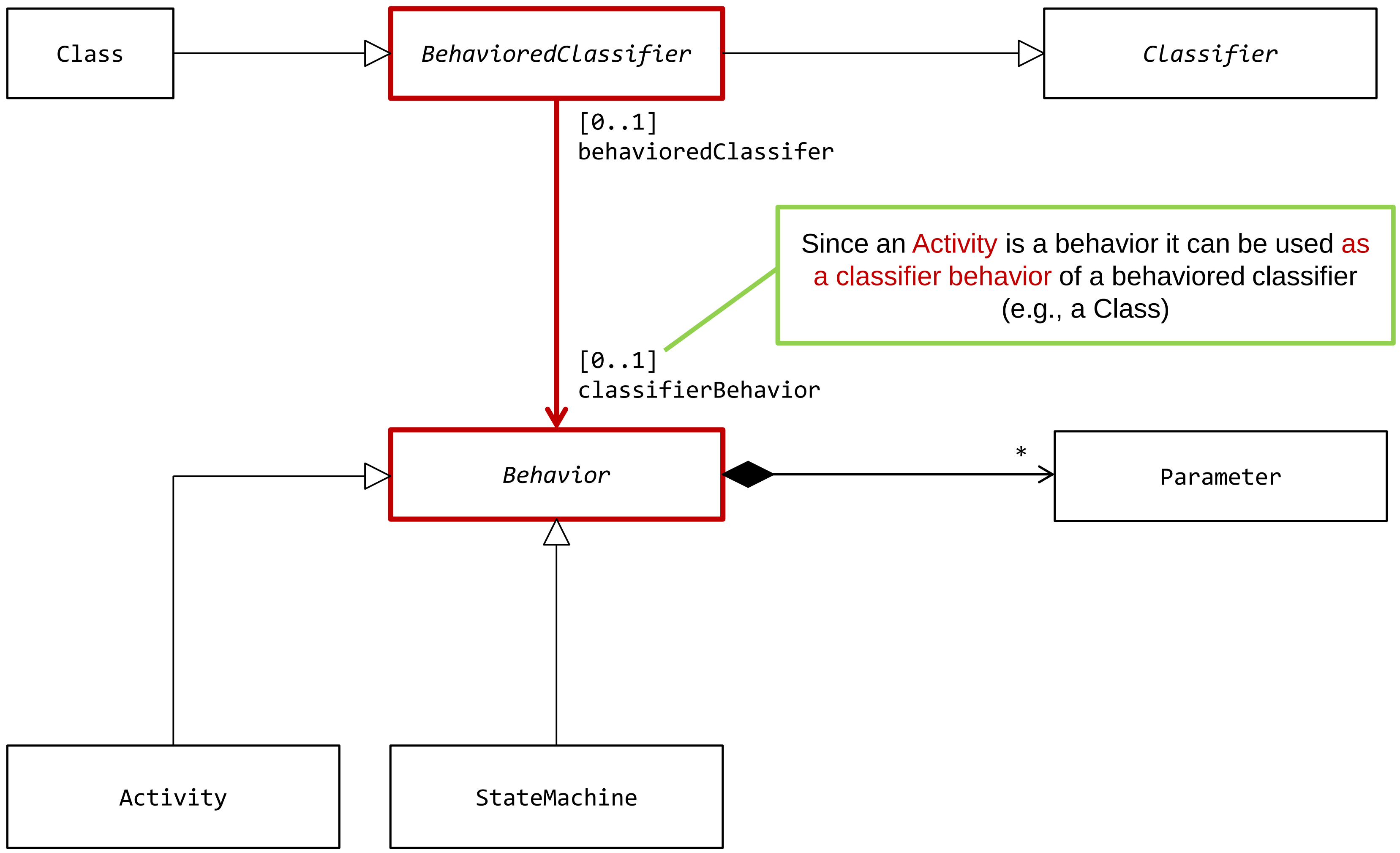
ACTVITY USAGES ALLOWED BY UML (2/3)

AS AN OPERATION IMPLEMENTATION



Since an activity is a behavior it can be used to play the role of a method for an Operation

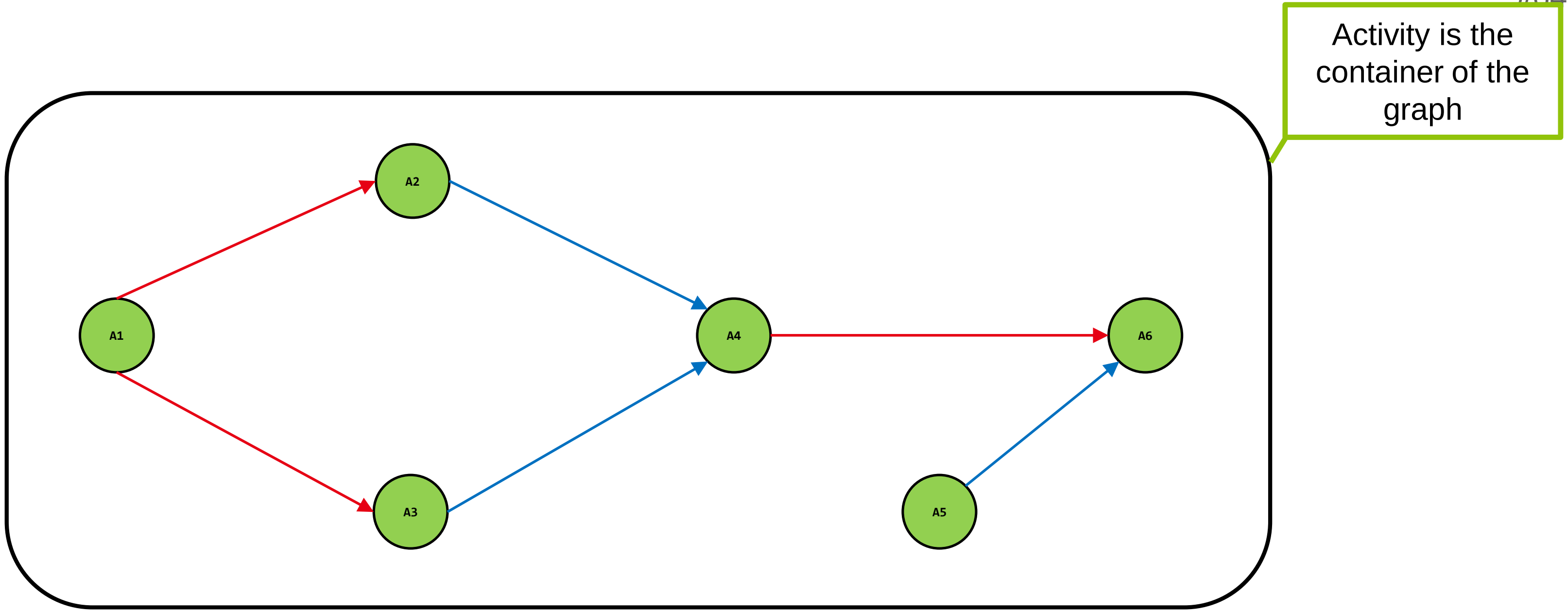
ACTVITY USAGES ALLOWED BY UML (2/3)
AS CLASSIFIER BEHAVIOR



Since an Activity is a behavior it can be used as a classifier behavior of a behaviored classifier (e.g., a Class)

CONTENT

DIRECTED GRAPH LIKE STRUCTURE (1/2)

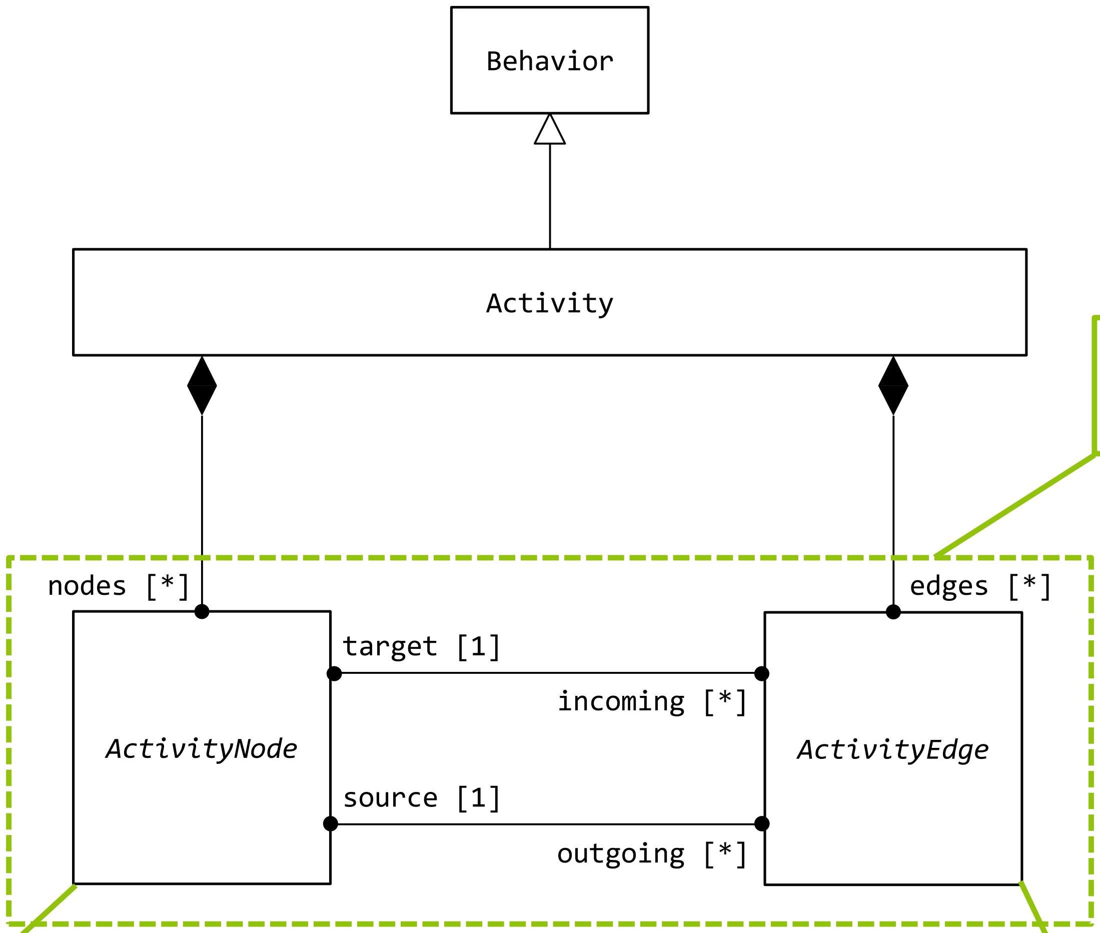


Graph composition

- The activity itself is the container of the graph
- The nodes denotes actions to realize
- The edges denote precedence relationships between action nodes
  - Edges in red denote only precedence relationships
- The edges denotes flowing of data between action nodes
  - Edges in blue denote a precedence relationship as well as data flowing from an action to another

DIRECTED GRAPH LIKE STRUCTURE (2/2)

WHAT DOES THE META-MODEL SAY?



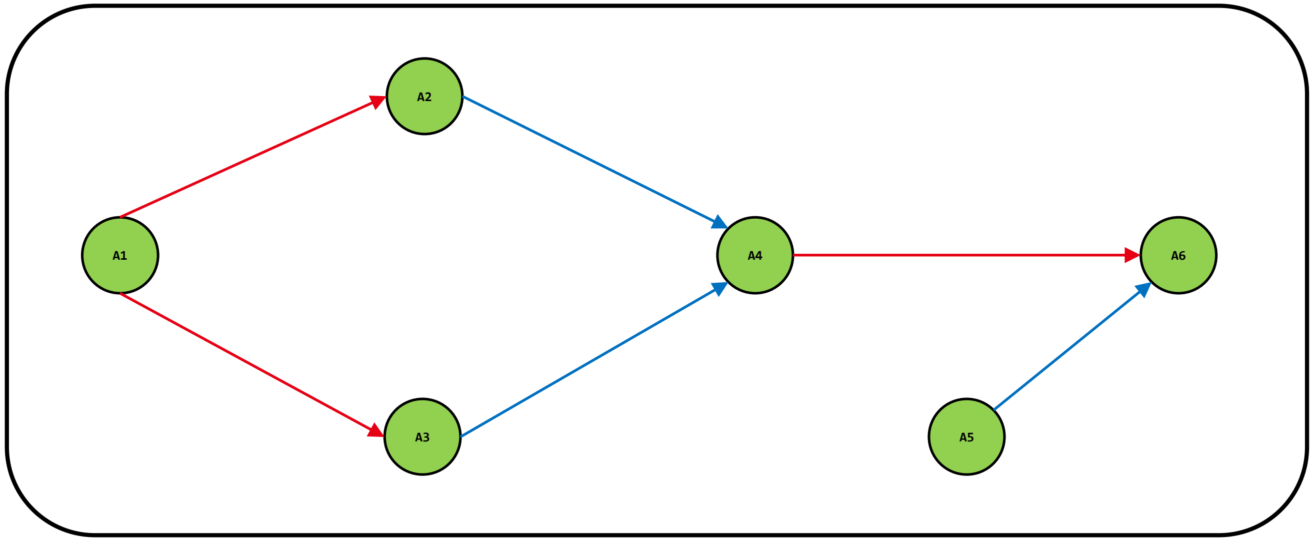
We retrieve the graph like structure

Activity nodes specialize into a large set of "action nodes" that all expose a different semantics

ActiviyEdge specializes into two kinds of edges: ControlFlow and ObjectFlow. These two kinds of edges are used to materialize precedence relationships and data passing

DATA-FLOW SEMANTICS (1/4)

THE EXECUTION RULES



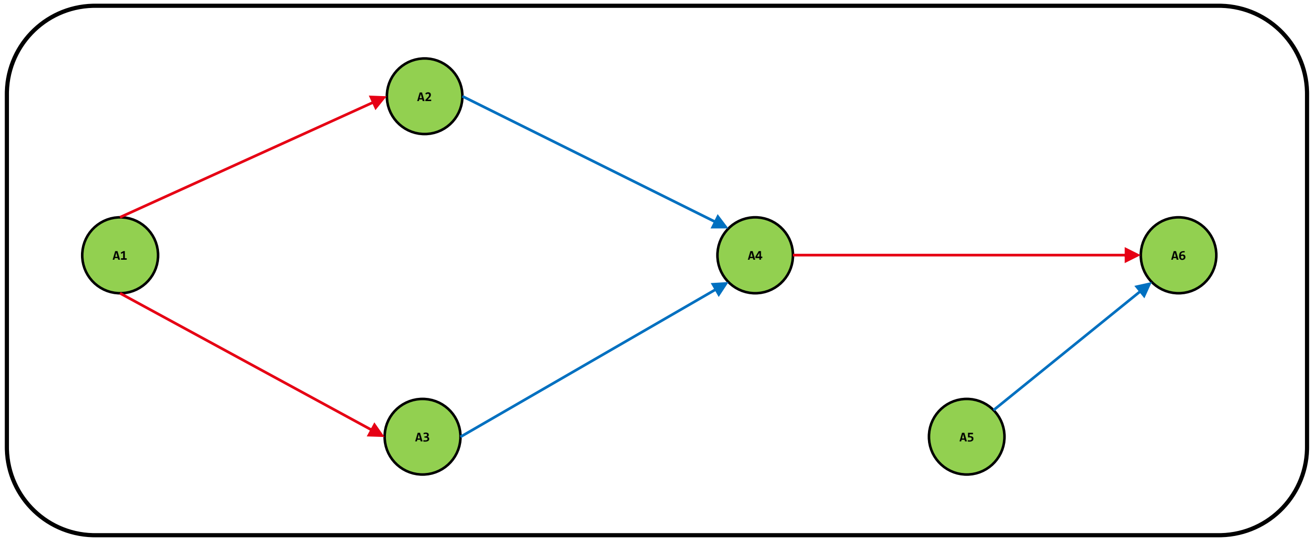
Execution rules

- A node can be executed when
  - Its control and data dependencies are satisfied
  - Example: A4 cannot be executed before A1-A2-A3 are done

- When the activity execution starts
  - Nodes that have no dependencies to satisfy are executed concurrently
  - Example: A1 and A5 executes concurrently

DATA-FLOW SEMANTICS (3/4)

PARTIAL EXECUTION ORDERS

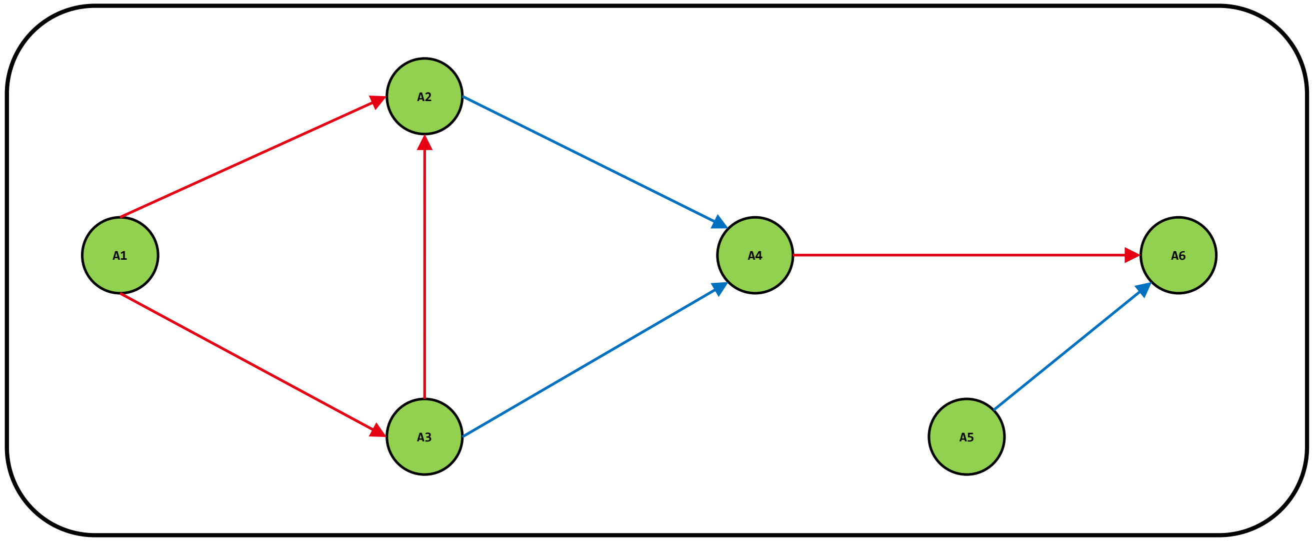


We can be sure that:	However we cannot say that:	Partial execution order:
A2 after A1	A5 after A1 (and reverse)	A1 and A5 (parallel)
A3 after A1	A3 after A2 (and reverse)	A2 and A3 (parallel or sequence)
A4 after A2, A3		A4
A6 after A4, A5		A6

Dependent on the model form (i.e., does parallelism is explicitly described with a Fork ?)

DATA-FLOW SEMANTICS (3/4)

HOW CAN WE FORCE A PARTICULAR ORDER ?

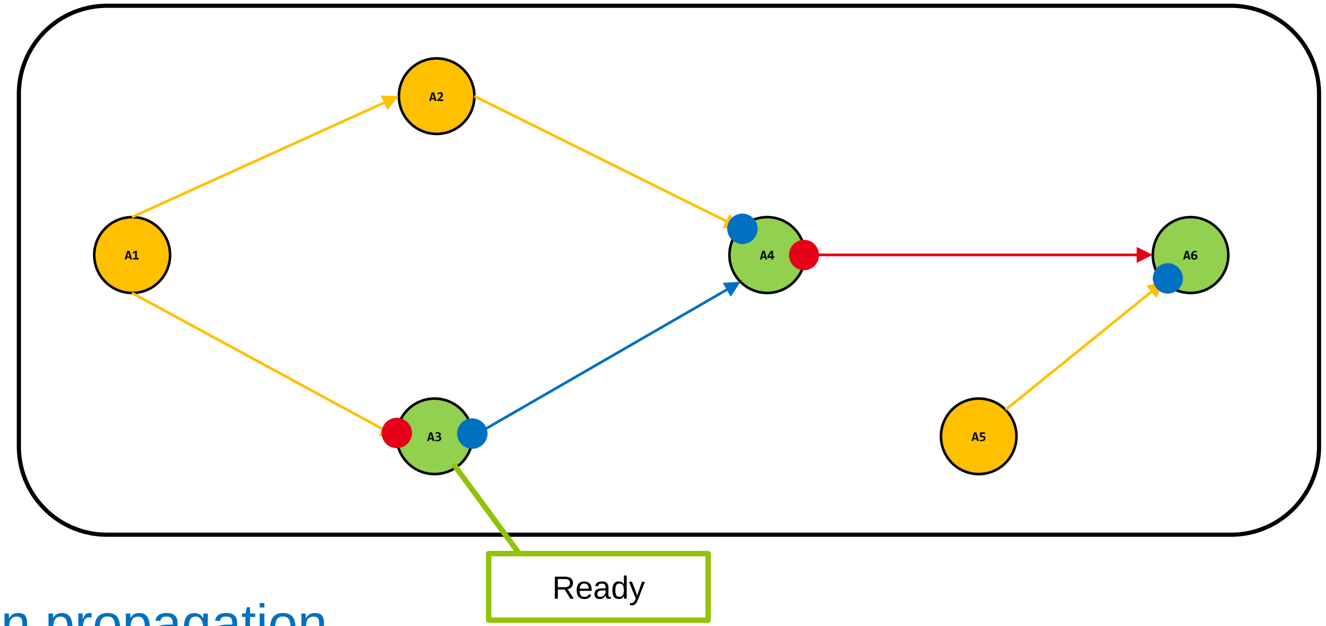


Question:

- How can we say that A2 will be executed each time after A3 ?

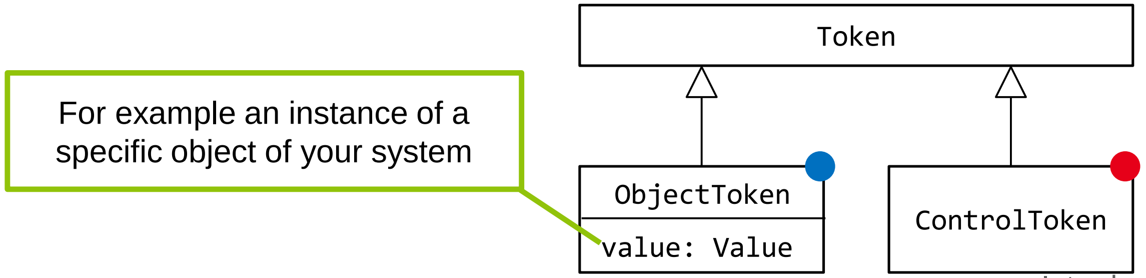
DATA-FLOW SEMANTICS (4/4)

EXECUTION PROPAGATION: TOKEN BASED SYSTEM



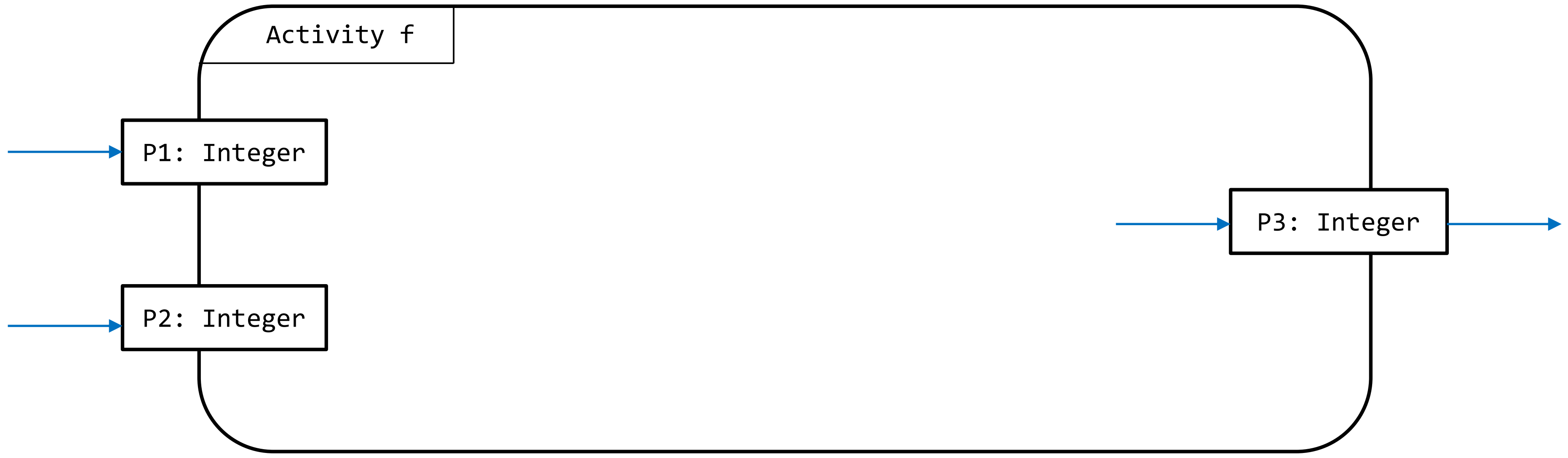
Execution propagation

- Materialized by tokens flowing through edges
  - Control token: propagate across control flows
  - Object token: propagate across the object flows and ships data



CONTENT

ACTIVITY
PARAMETERS NODES



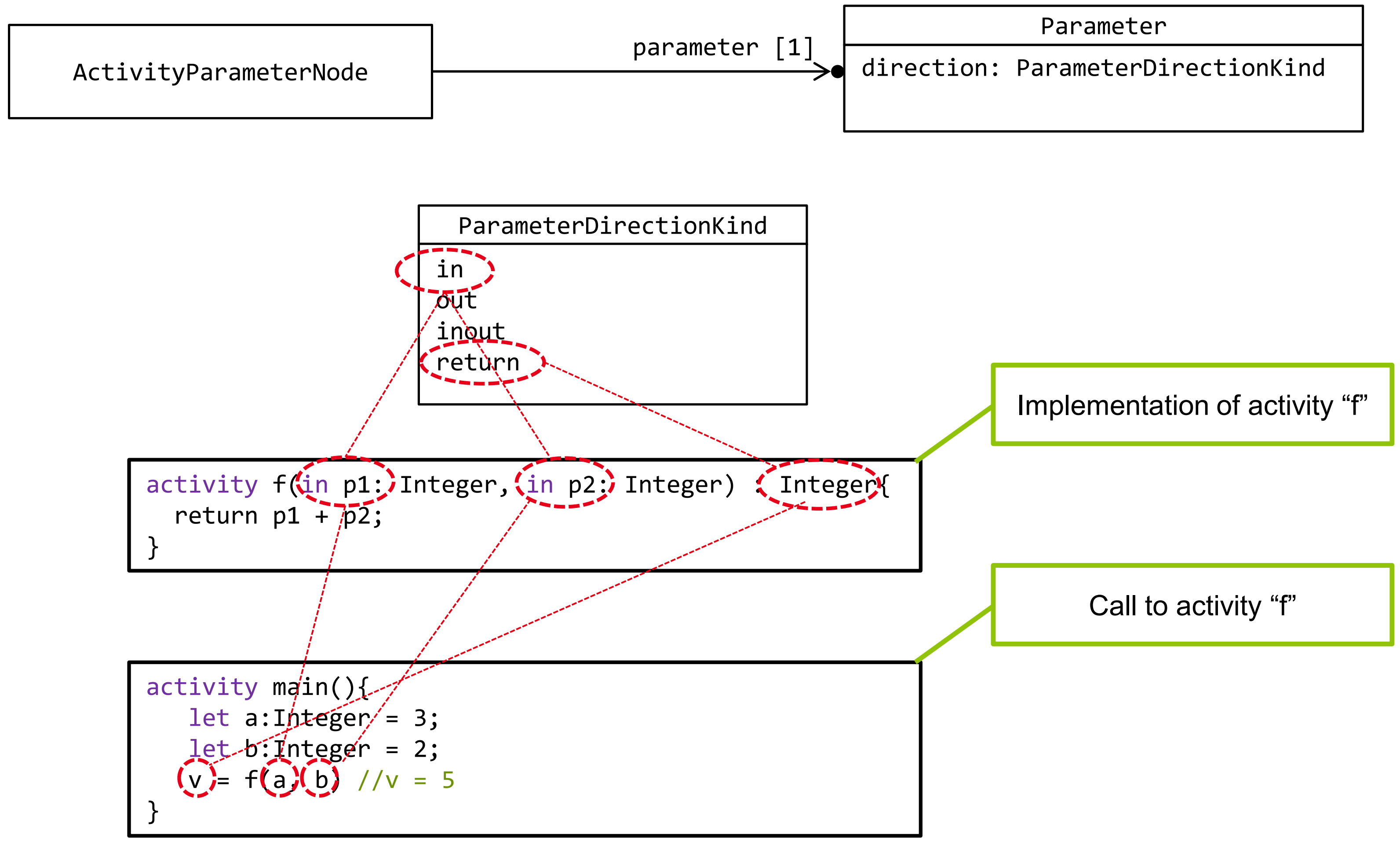
Activity parameters nodes

- Denote points used to provide an activity with input values
- Denote points used by an activity to provide output values
- They are typed (not mandatory)
  - Can be derived from the parameter attached to the activity parameter node

```
activity main(){
  v = f(5,2)
}
```

A valid call for this activity

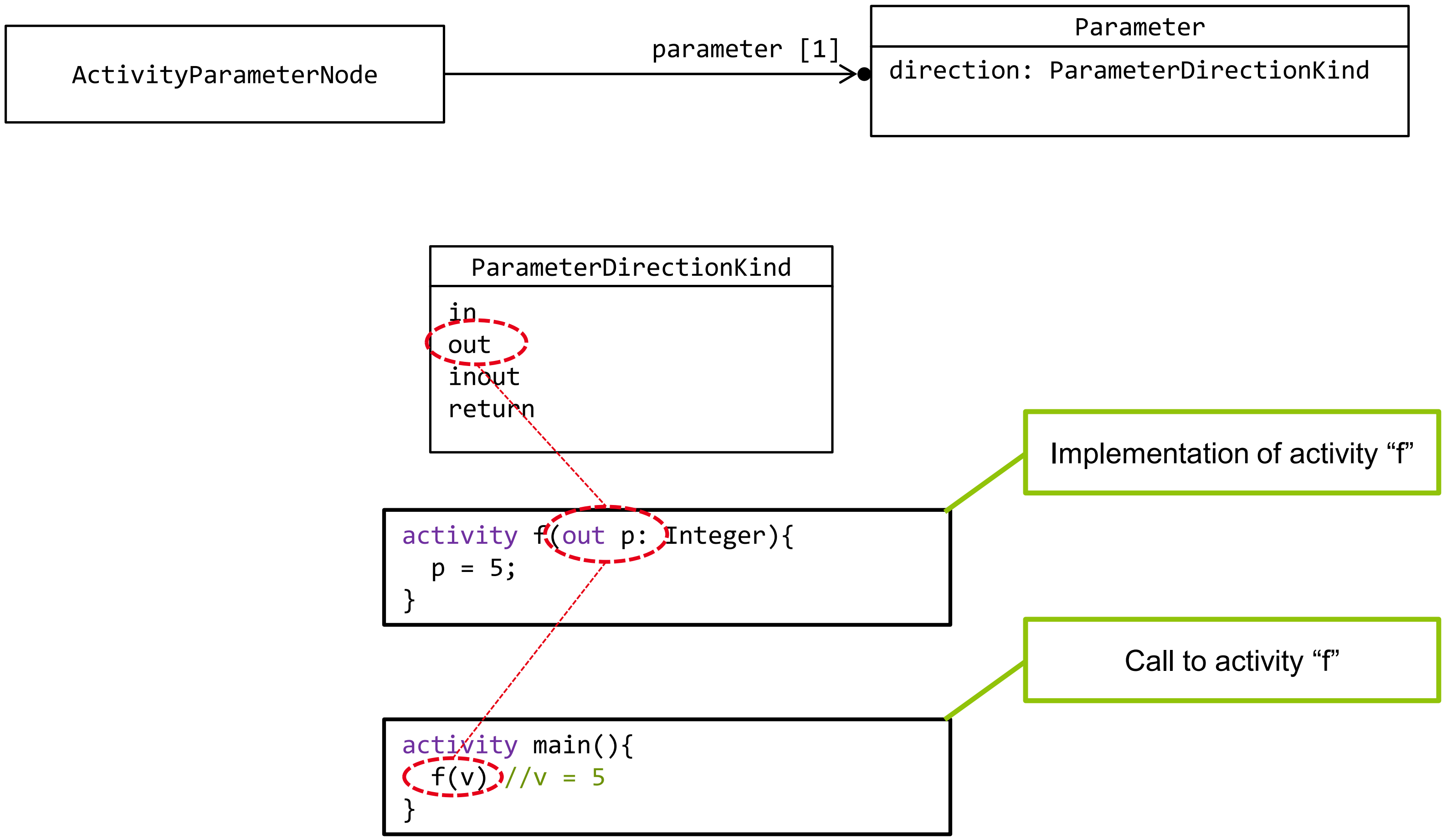
Direction of a parameter



```
activity f(in p1: Integer, in p2: Integer) : Integer{
  return p1 + p2;
}
```

```
activity main(){
   let a:Integer = 3;
   let b:Integer = 2;
   v = f(a, b) //v = 5
}
```

Direction of Parameter



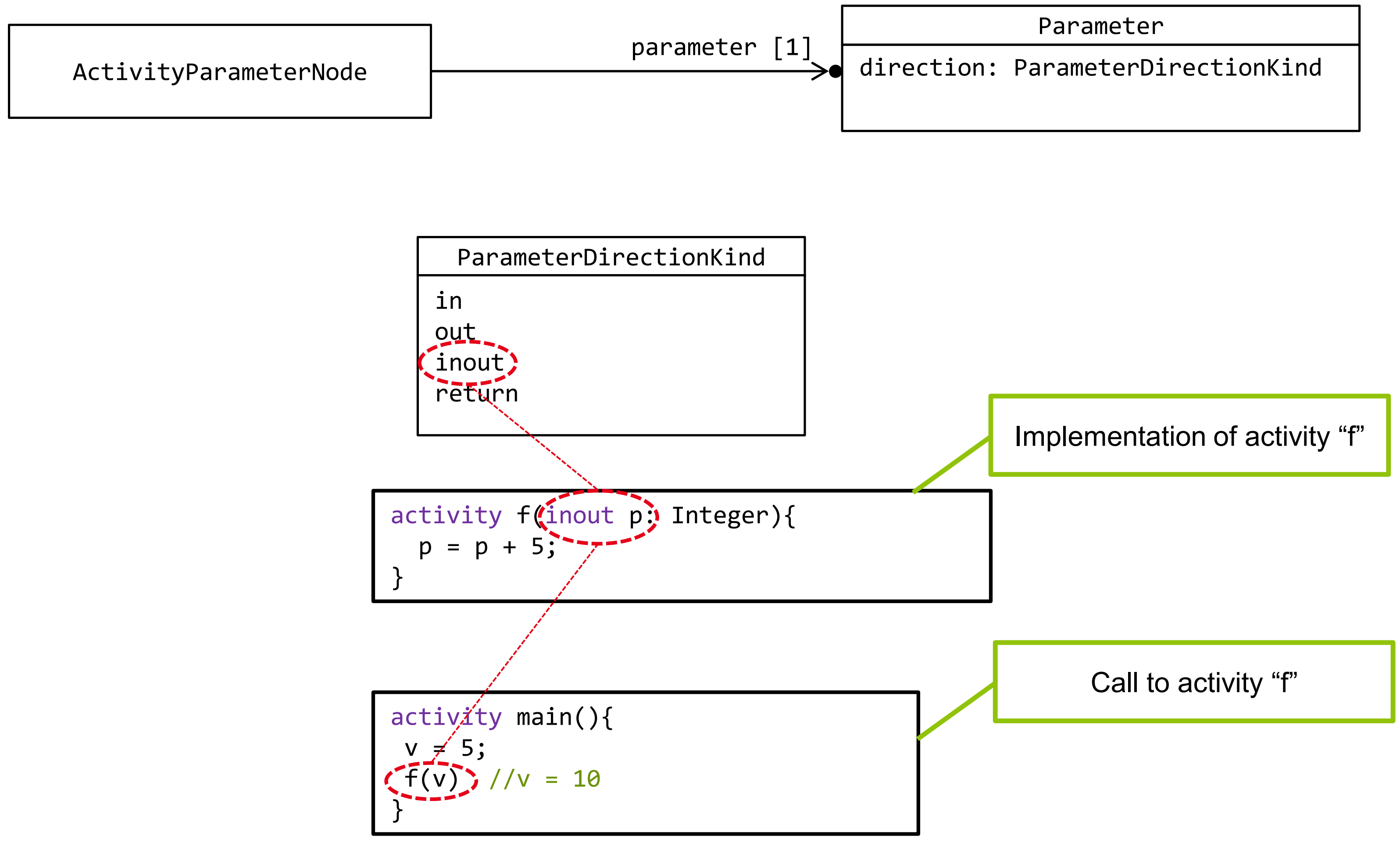
Implementation of activity “f”

```
activity f(out p: Integer){
   p = 5;
}
```

Call to activity “f”

```
activity main(){
   f(v) //v = 5
}
```

Direction of Parameter

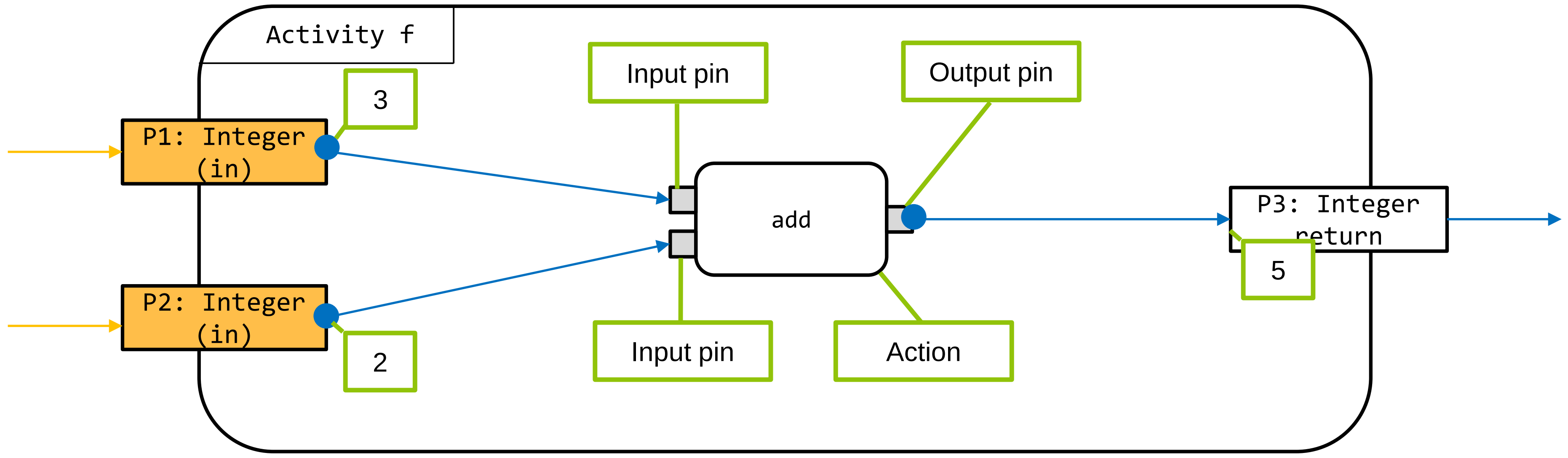


```
activity f(inout p: Integer){
  p = p + 5;
}
```

```
activity main(){
 v = 5;
 f(v) //v = 10
}
```

ACTIONS

INPUT AND OUTPUT PINS



Actions

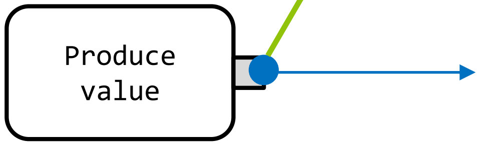
– Can consume and/or produced data
  - Data consumed by an action come from its input pin(s)
  - Data produced are placed on its output pin(s)

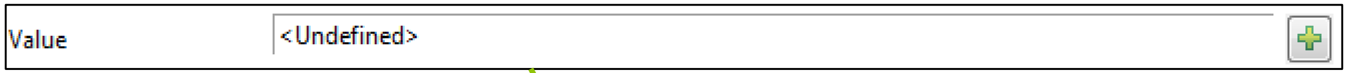
– Note: it is not mandatory for an action to consume or produce data

CONTENT

VALUE SPECIFICATION ACTION

The value placed on the result pin is typed. It depends on the interpretation of the value specification associated with the action



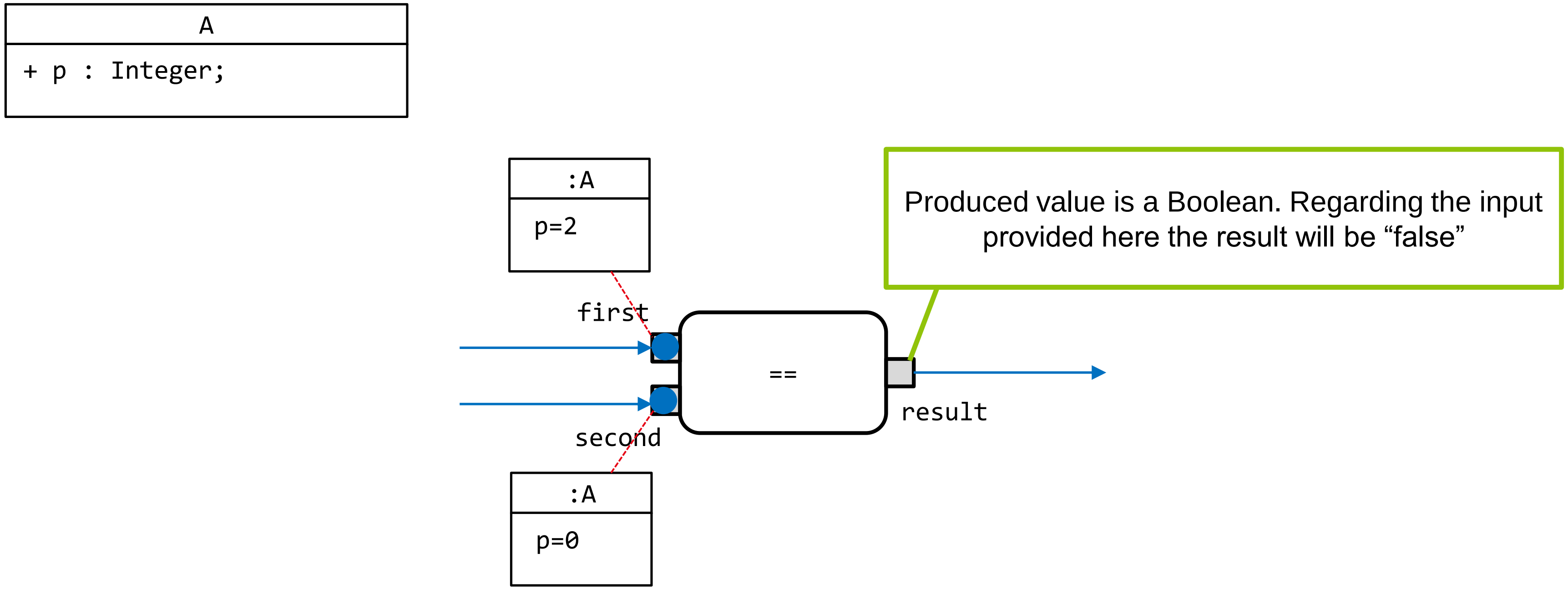


The value to be produced is specified for the action through the Papyrus property view

ValueSpecificationAction

- Produce a value on its result output pin from any kind of value specification
  - Expression (e.g. StringExpression, TimeExpression)
  - Literal value (e.g. LiteralInteger)
  - Duration
  - InstanceValue (e.g. an instance of a class)

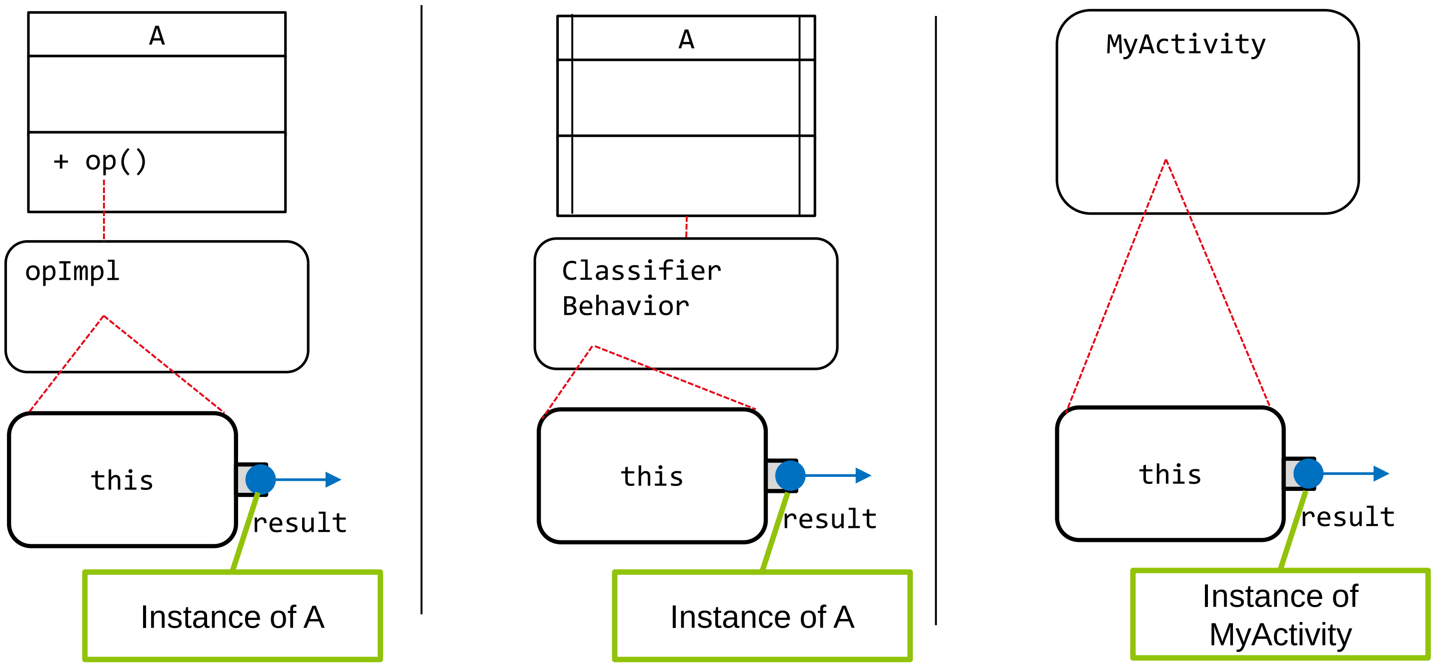
TEST IDENTITY ACTION


A
+ p : Integer;
:A
p=2
first
==
result
second
:A
p=0
Produced value is a Boolean. Regarding the input provided here the result will be “false”

TestIdentityAction

- Assess the equality between the provided values
  - Two values are equal since they have the same types as well the same values for their properties
  - Two enumeration values are equal since they use the same enumeration literal

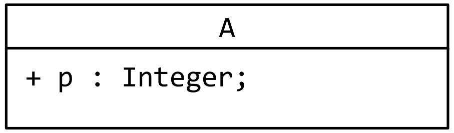
READ SELF ACTION

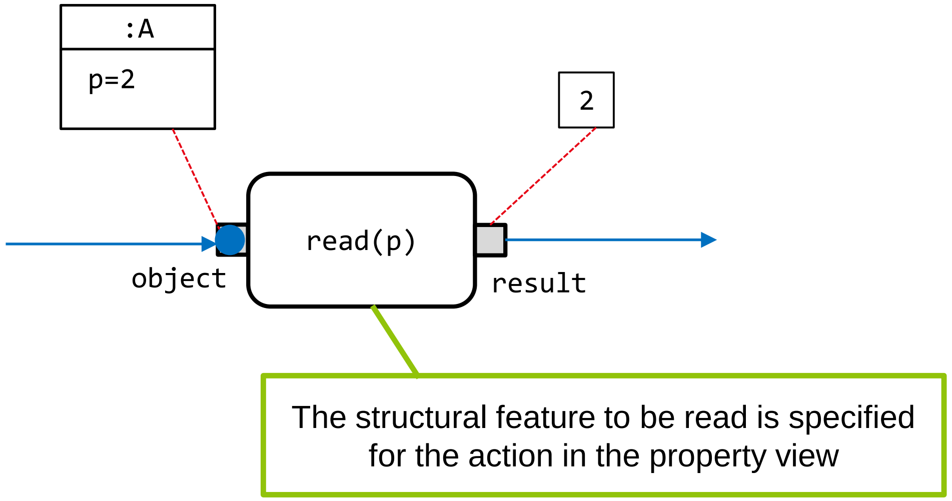


Provides a reference on the execution context

- Looks like a “this” instruction in C++ or Java
- If the activity is used in standalone mode
  - The context is the activity itself
- If the activity is used as an implementation for an operation
  - The context is the class instance that was used to make the operation call
- If the activity is used as a classifier behavior
  - The context is the instance of the active class executing the classifier behavior

READ STRUCTURAL FEATURE



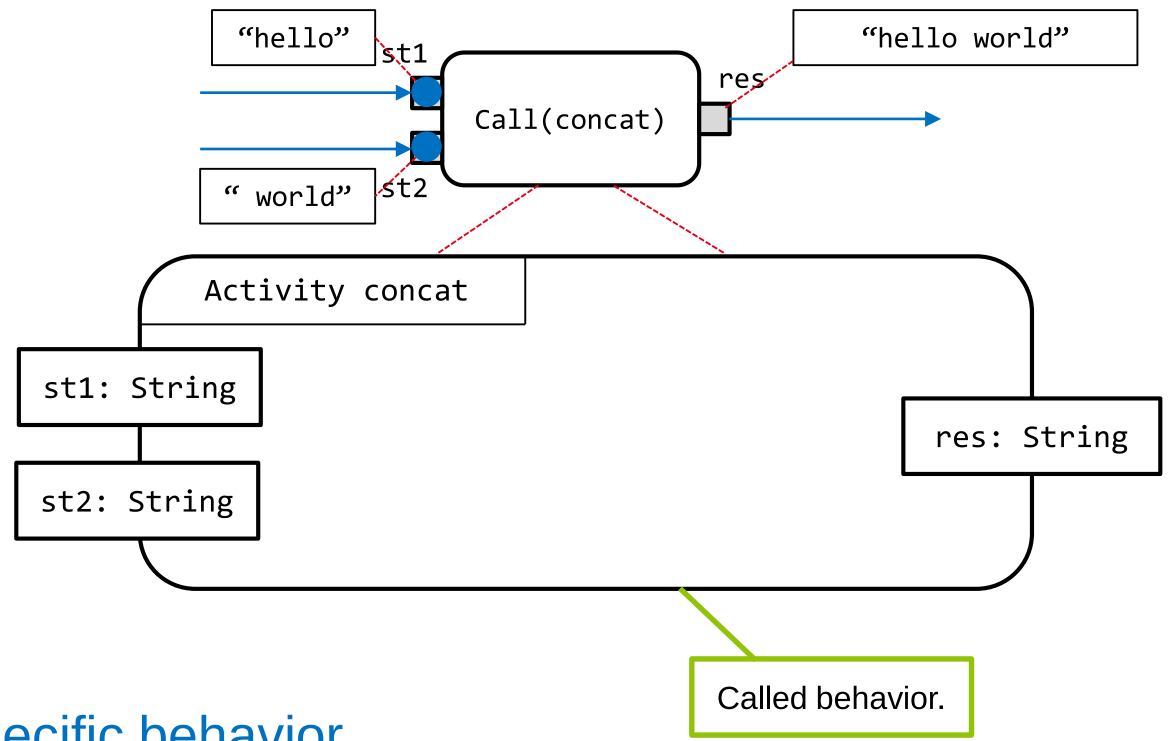

:A
p=2
2
read(p)
object
result
The structural feature to be read is specified for the action in the property view

Read a structural feature that belongs to a classifier instance

- Requires a target to read
  - Basically the object into which the feature must be read
- Provides the result of the reading on its output pin
  - Type of the value is the type of feature

CONTENT

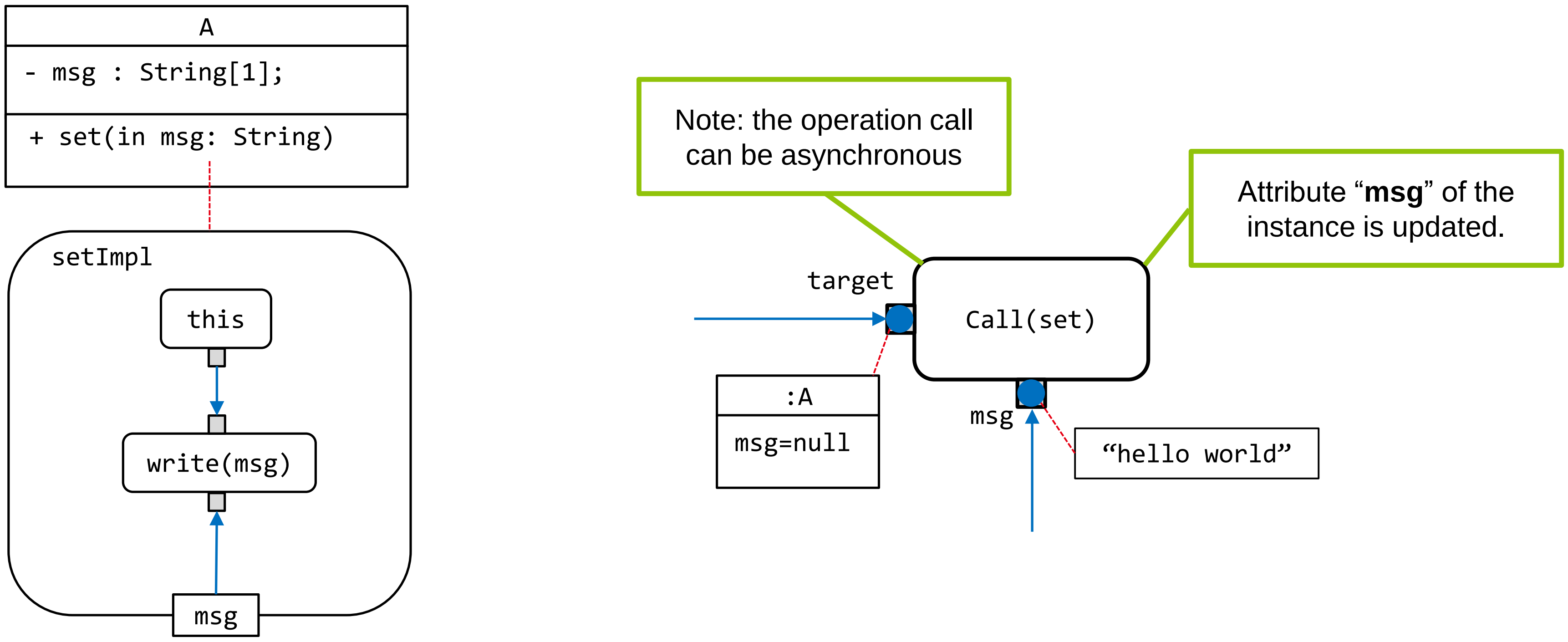
CALL BEHAVIOR ACTION



Call to a specific behavior

- Similar to a function call in C for example
- Requires the behavior to be called to be specified
  - This can be realized through the property view of Papyrus
- Eventually requires values for input pins if the called behavior as parameters

CALL OPERATION ACTION

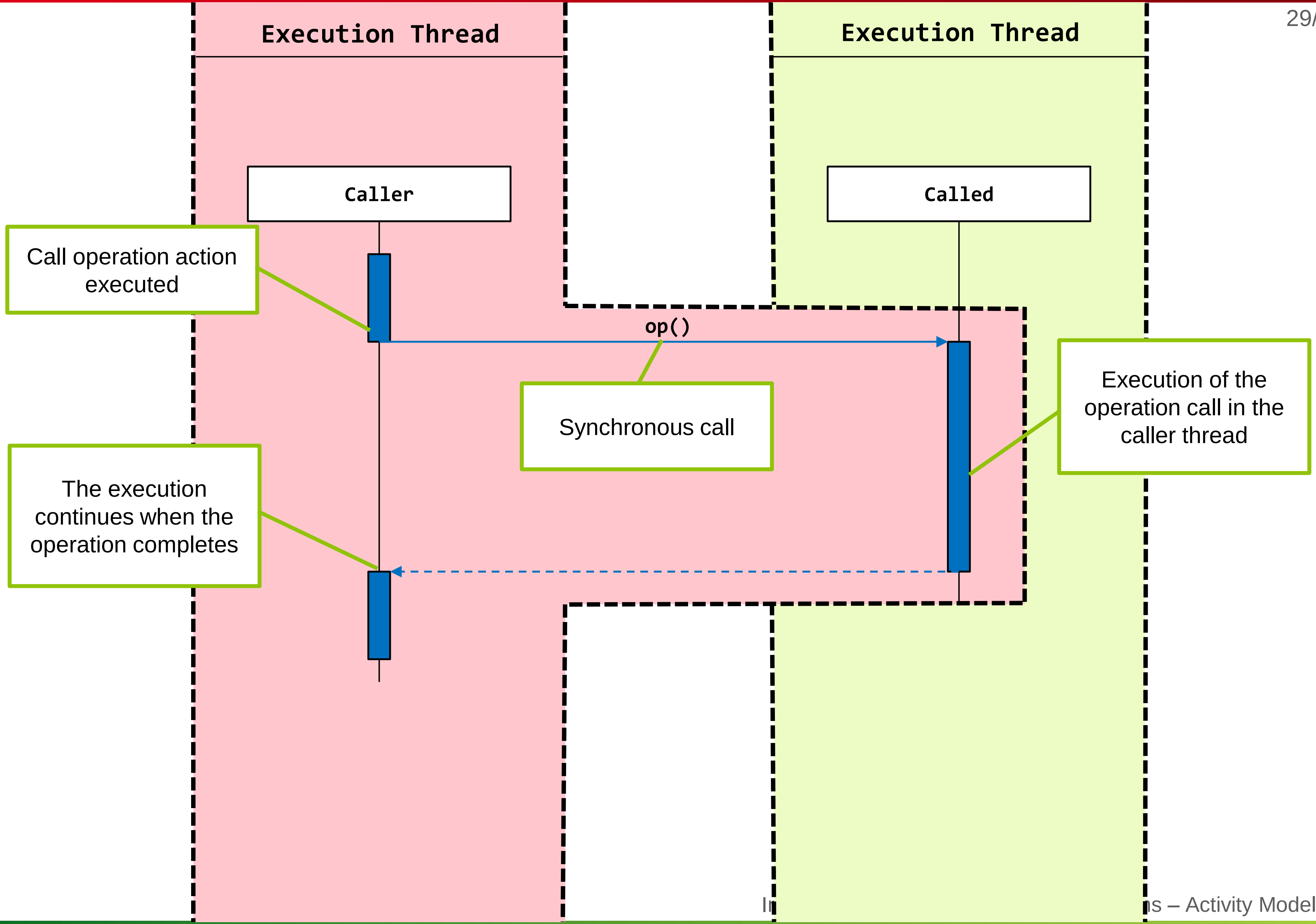


Call to a an operation of class

- Requires a target on which the operation can be called
  - The target is an instance of a classifier
- Requires the operation to be called to be specified
  - This can be realized through the property view of Papyrus
- Eventually requires values for input pins if the called operation as parameters

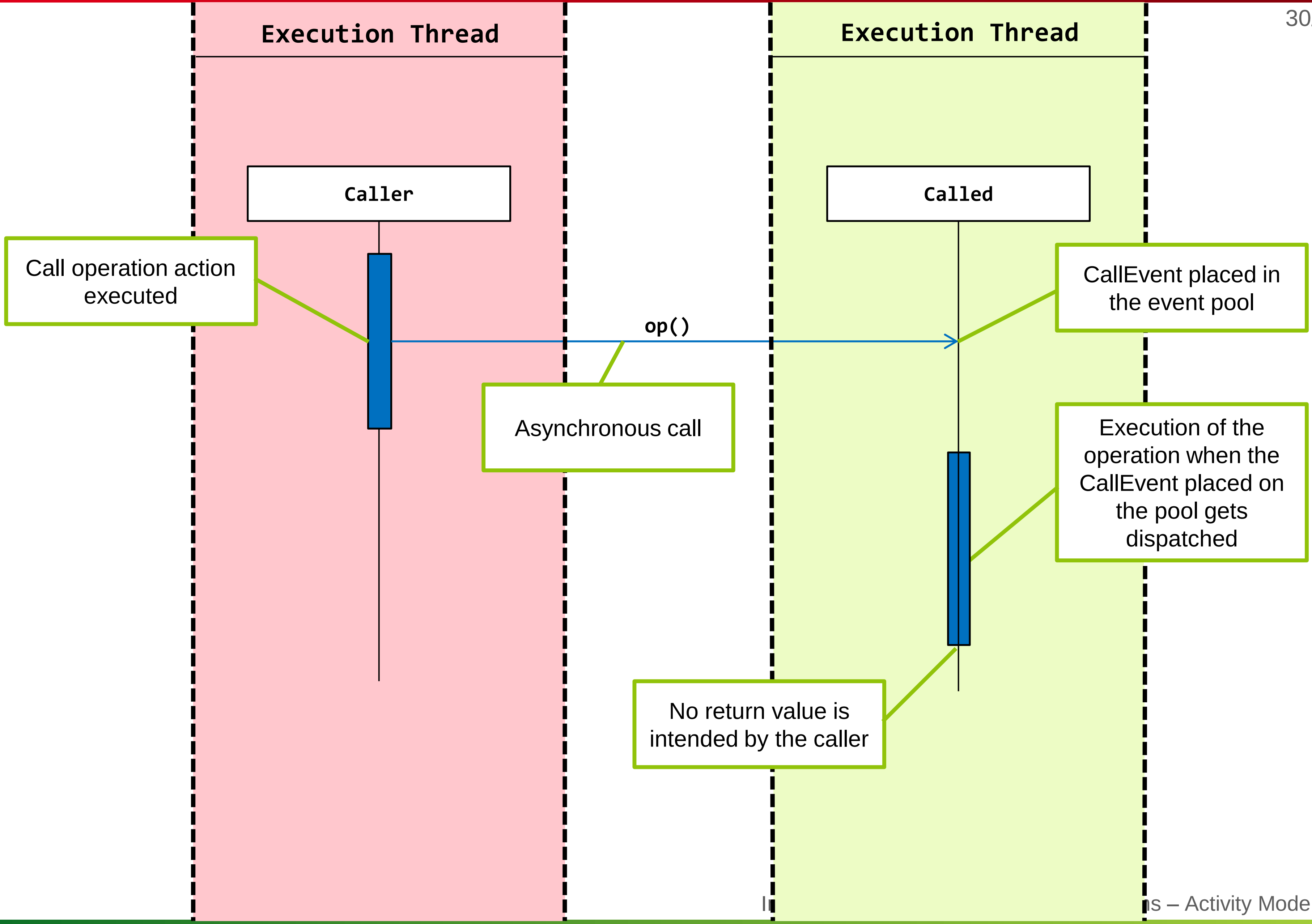
REMINDER

SYNCHRONOUS AND ASYNCHRONOUS CALLS (1/2)

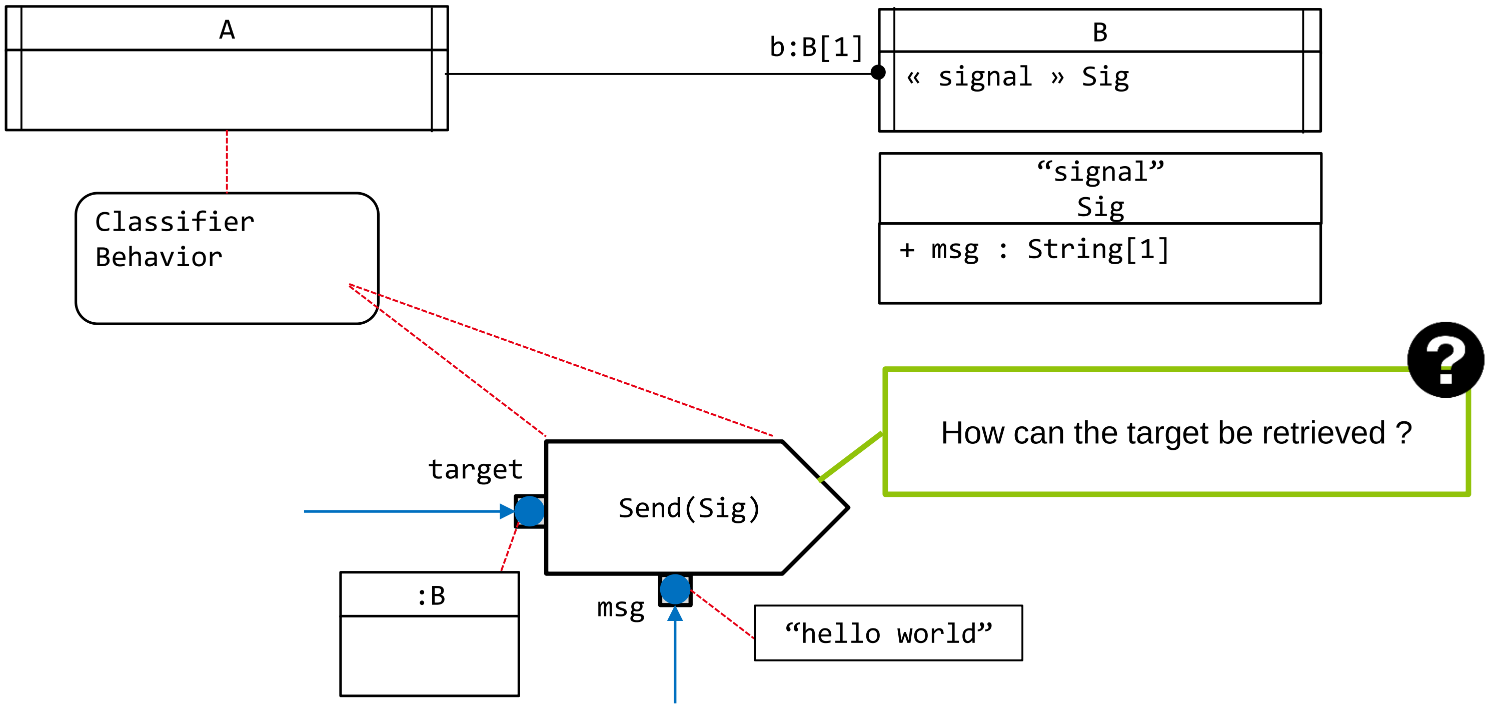

Execution Thread
Execution Thread
Caller
Called
Call operation action executed
op()
Synchronous call
Execution of the operation call in the caller thread
The execution continues when the operation completes

REMINDER

SYNCHRONOUS AND ASYNCHRONOUS CALLS (2/2)


Execution Thread
Execution Thread
Caller
Called
Call operation action executed
op()
CallEvent placed in the event pool
Asynchronous call
Execution of the operation when the CallEvent placed on the pool gets dispatched
No return value is intended by the caller

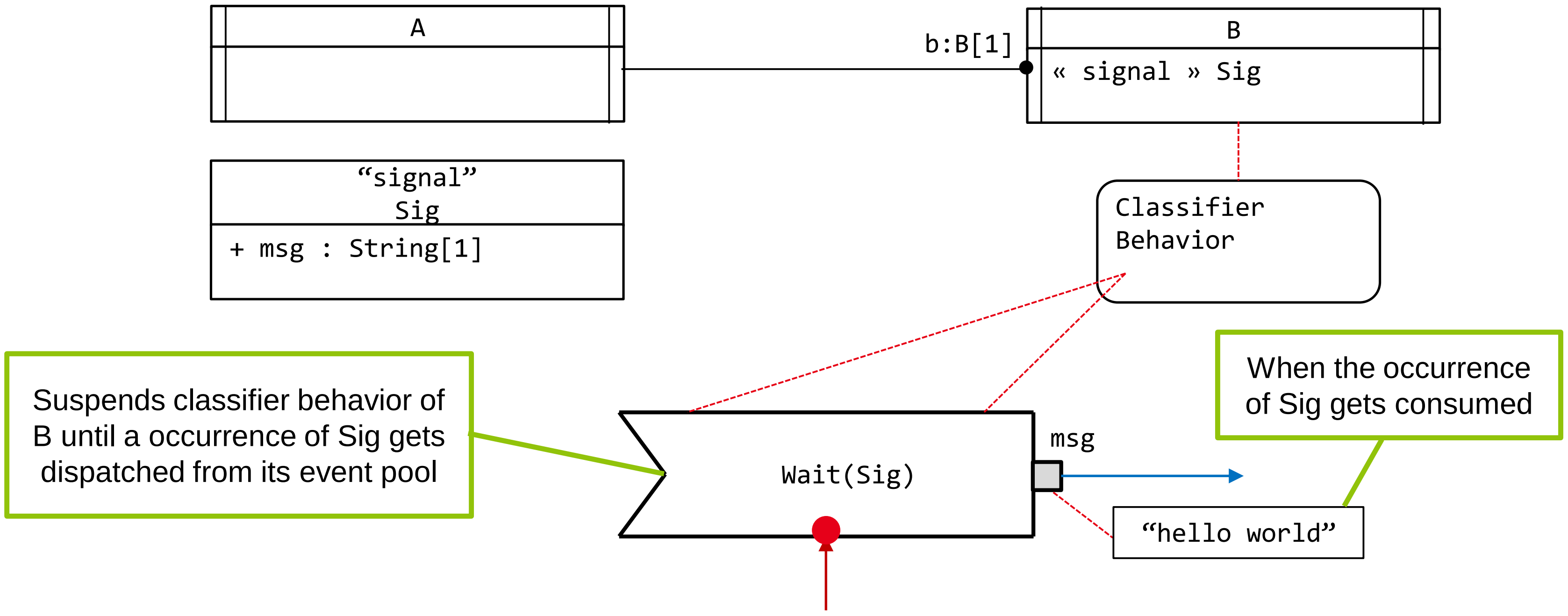
ASYNCHRONOUS COMMUNICATIONS (1/2)



Send a signal to an instance of an active object

- Requires a target to which the signal is sent
  - It is an instance of a classifier that is active (i.e., has a classifier behavior)
- Requires the signal type to be sent to be specified
  - This can be realized through the property view of Papyrus
- Eventually requires values for input pins if the type of the signal that is sent has structural features

ASYNCHRONOUS COMMUNICATIONS (2/2)


A
b:B[1]
B
« signal » Sig
"signal"
Sig
+ msg : String[1]
Classifier
Behavior
Suspends classifier behavior of B until a occurrence of Sig gets dispatched from its event pool
When the occurrence of Sig gets consumed
Wait(Sig)
msg
"hello world"

Suspends the classifier behavior that executes this action

- Register an event accepter for the event (SignalEvent, CallEvent,…)
  - The one referenced by the trigger attached to the "AcceptEventAction"
- Execution continues when an occurrence of the expected event arrives at the event pool and gets dispatched
- If the expected event is intended to ship values (e.g., for signal properties)
  - Properties values are placed on output pin(s) of the action
  - Note: this requires the action to specify "isUnmarshall" to true

EVENT DISPATCHING AND RTC STEPS

Event dispatching

- The event dispatching semantics is shared for all kind of behaviors
- Events are placed in the pool according to their order of arrival
- An event that is consumed cannot return in the pool
  - It is considered as lost
  - Exception for specific situation (e.g., deferred events for state-machines)
- Only one event is dispatched (i.e. consumed) at a time
  - This denotes the starting of Run-To-Completion step

Run to completion

- Starts with the consumption of an event
- If the event does not match any trigger the steps completes trivially
- If there is a matched trigger
  - The execution starts from the element to which the trigger is attached.
  - The execution flows propagates while it is possible to do so.
  - When the execution flow stops (e.g., unsatisfied dependencies in the context of an activity) the step completes.

CONTENT

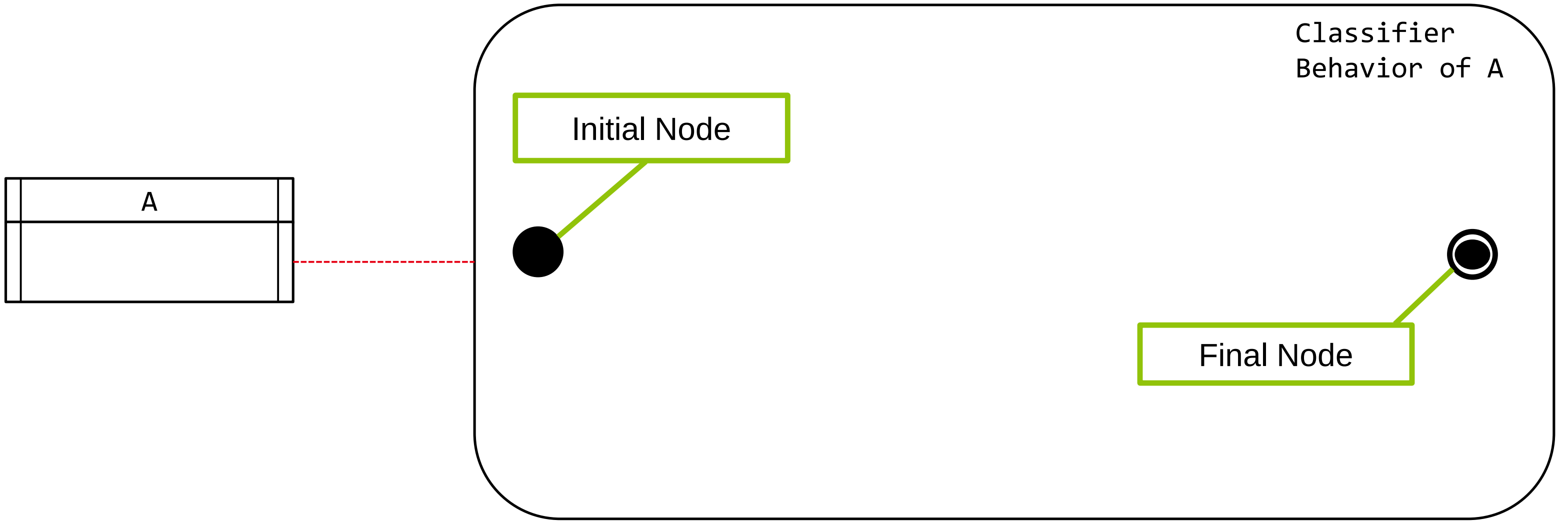
INITIAL AND FINAL NODES

Initial Node

- Denotes the starting point of an activity
  - If it is not specified then all nodes in the activity that do no have incoming edges are started concurrently

Final Node

- Denotes the end point of an activity
  - When this node is reached then the containing activity terminates



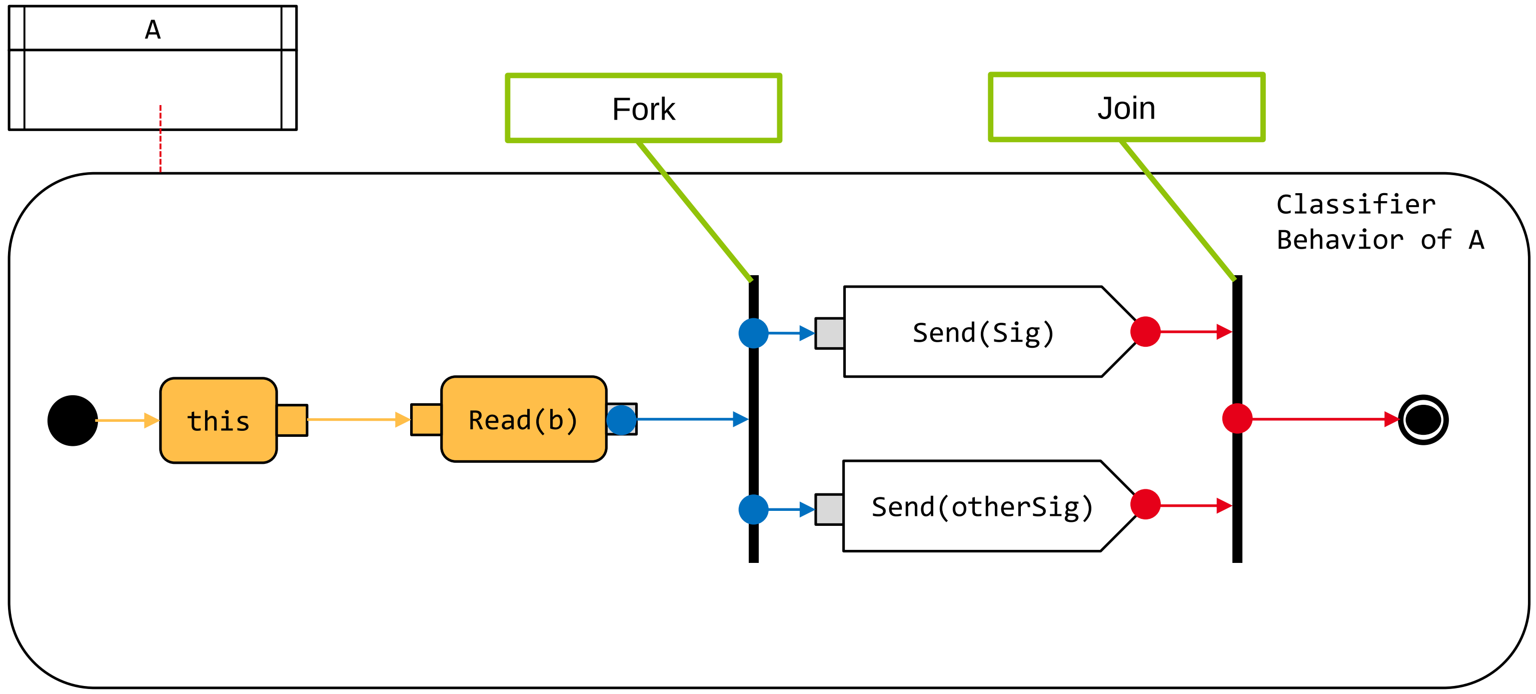
FORK AND JOIN
PARALLEL EXECUTION FLOWS

Fork

- Denotes the beginning of multiple concurrent execution flows
  - Each outgoing edges of the fork fires concurrently
  - Each branch is executed in parallel with the other

Join

- Denotes the end multiple concurrent execution flows
  - To be traversed the join must receive a token from all of its incoming edges



MERGE AND DECISION

CONDITIONAL EXECUTION AND LOOPS

Merge

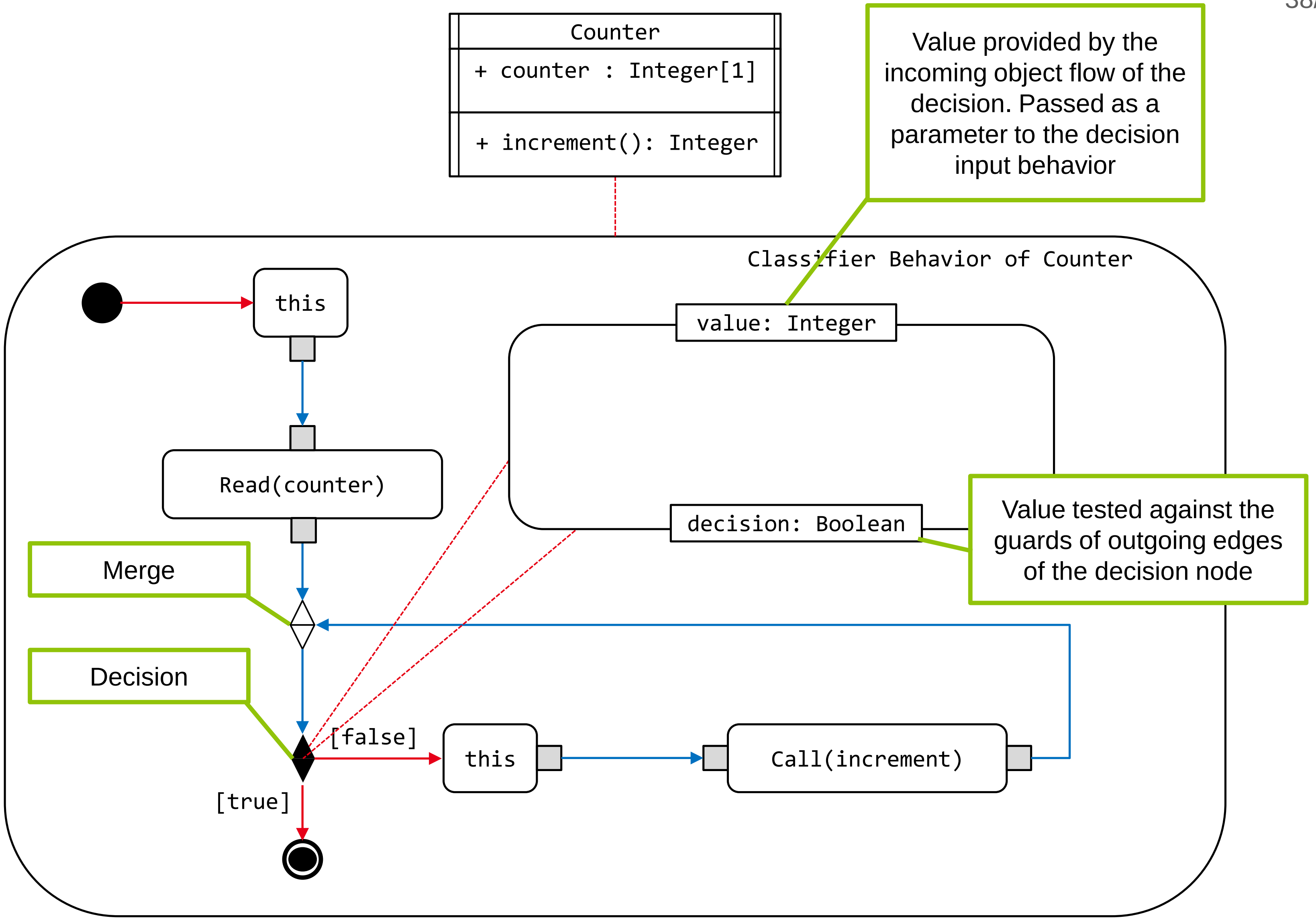
- Merge multiple incoming flows into a single one
  - If a single incoming edge is traversed then the merge is traversed
  - The execution flow continues on the outgoing edge.

Decision

- Provide the possibility to route the execution flow on a specific path
  - The choice of a specific branch is based on the evaluation of the decision value against guard specifications placed on outgoing edges.
- Decision value can be
  - The values produced by the decision input behavior
  - The values arriving by the incoming object flow if there is no decision input behavior and no input flow.
  - The value arriving by the input flow if there is one but no decision input behavior.

MERGE AND DECISION

CONDITIONAL EXECUTION AND LOOPS

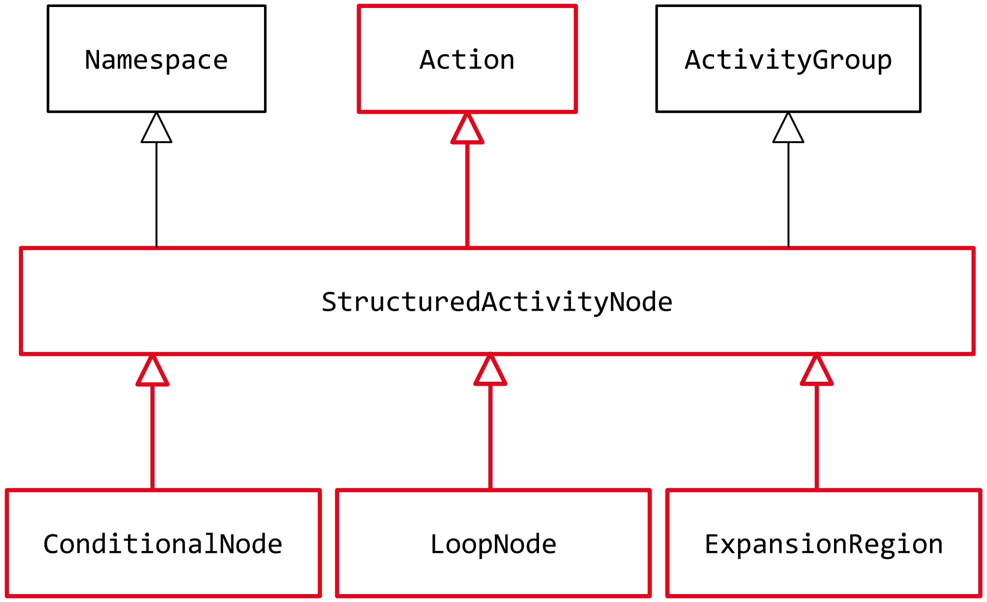

Counter
+ counter : Integer[1]
+ increment(): Integer
Value provided by the incoming object flow of the decision. Passed as a parameter to the decision input behavior
Classifier Behavior of Counter
this
value: Integer
Read(counter)
decision: Boolean
Value tested against the guards of outgoing edges of the decision node
Merge
Decision
[false]
this
Call(increment)
[true]

FROM RESEARCH TO INDUSTRY

CONTENT

MAKE ACTIVITY MORE CONCISE

THE BENEFITS OF STRUCTURED ACTIVITY NODES



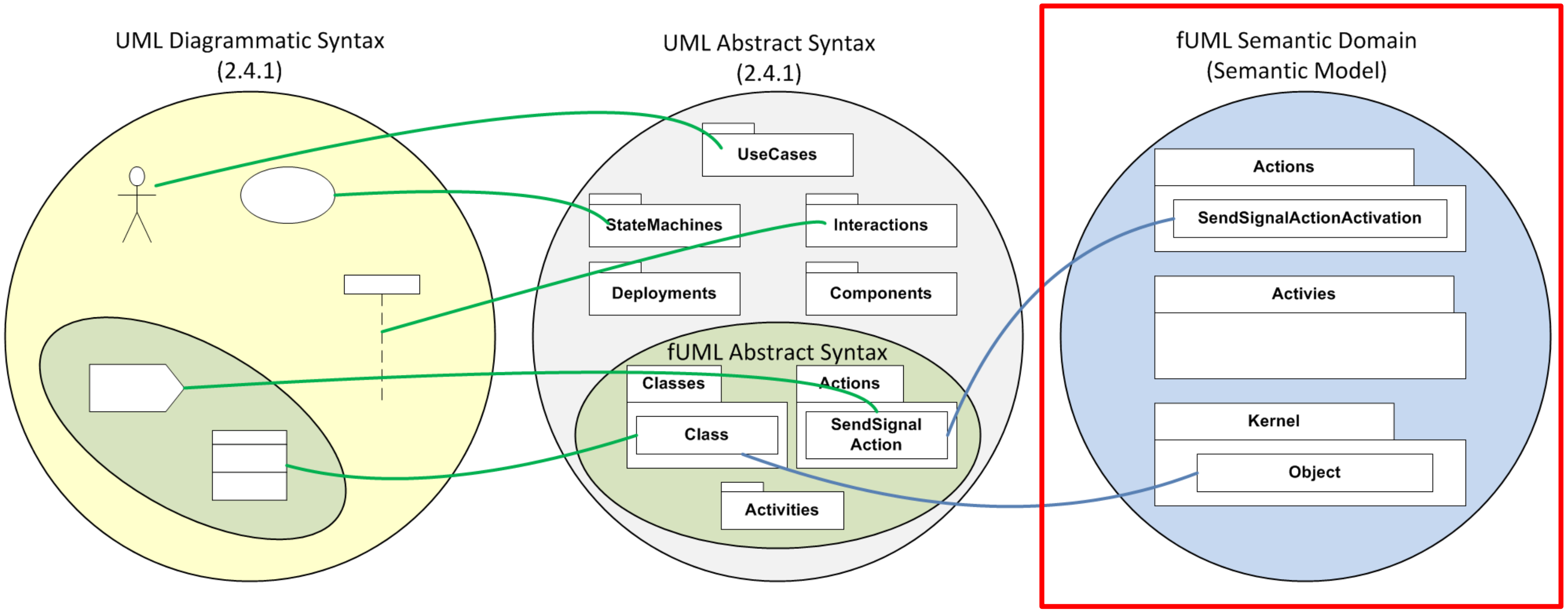
Structured Activity Nodes

- Conditional Node
  - Provide a way to structure a sequence of alternatives
- Loop Node
  - Provide a way to structure a loop
- ExpansionRegion
  - Execute the same content with different semantics

CONTENT

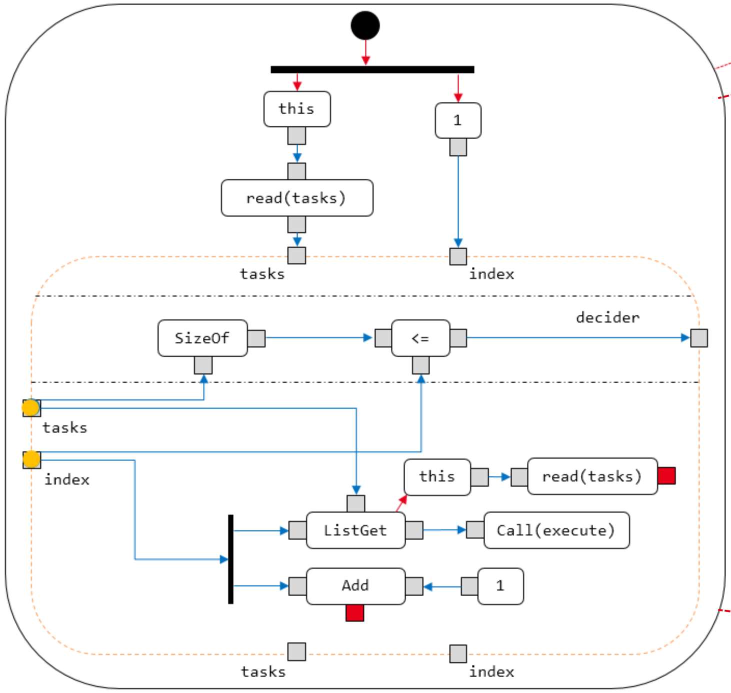
Foundational UML

- Formalizes the execution semantics of Classes and Activities
- Semantics is defined through a semantic model
  - A class model whose elements capture the intended behavior of syntactic elements (operational approach).
- Models built using Classes + Activities are executable
  - Let see a demo of the Papyrus model execution framework: Moka



Action Language for Foundational UML

- Textual surface notation to express executable models
  - Limited to the fUML scope
  - Evolutions: alignment with composite structures and state-machines
- Any model specified in Alf can be transformed into an fUML model
  - Any model specified with Alf is executable
  - The execution semantics is the one defined for fUML
- It makes a lot simpler the specification of detailed behaviors in UML



```
activity classifierBehavior(){
  this.tasks->iterate t (t.execute)
}
```

```
activity classifierBehavior(){
  for(task in this.tasks){
    task.execute();
  }
}
```

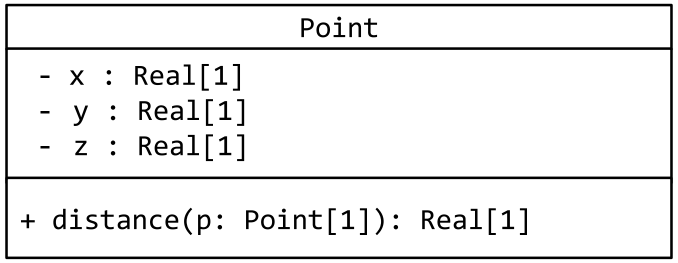
```
activity classifierBehavior(){
  let i: Integer = 1;
  while(i <= this.tasks->size()){
     this.tasks->at(i).execute();
     i++;
  }
}
```

papyrus
www.eclipse.org/papyrus

?

Moka

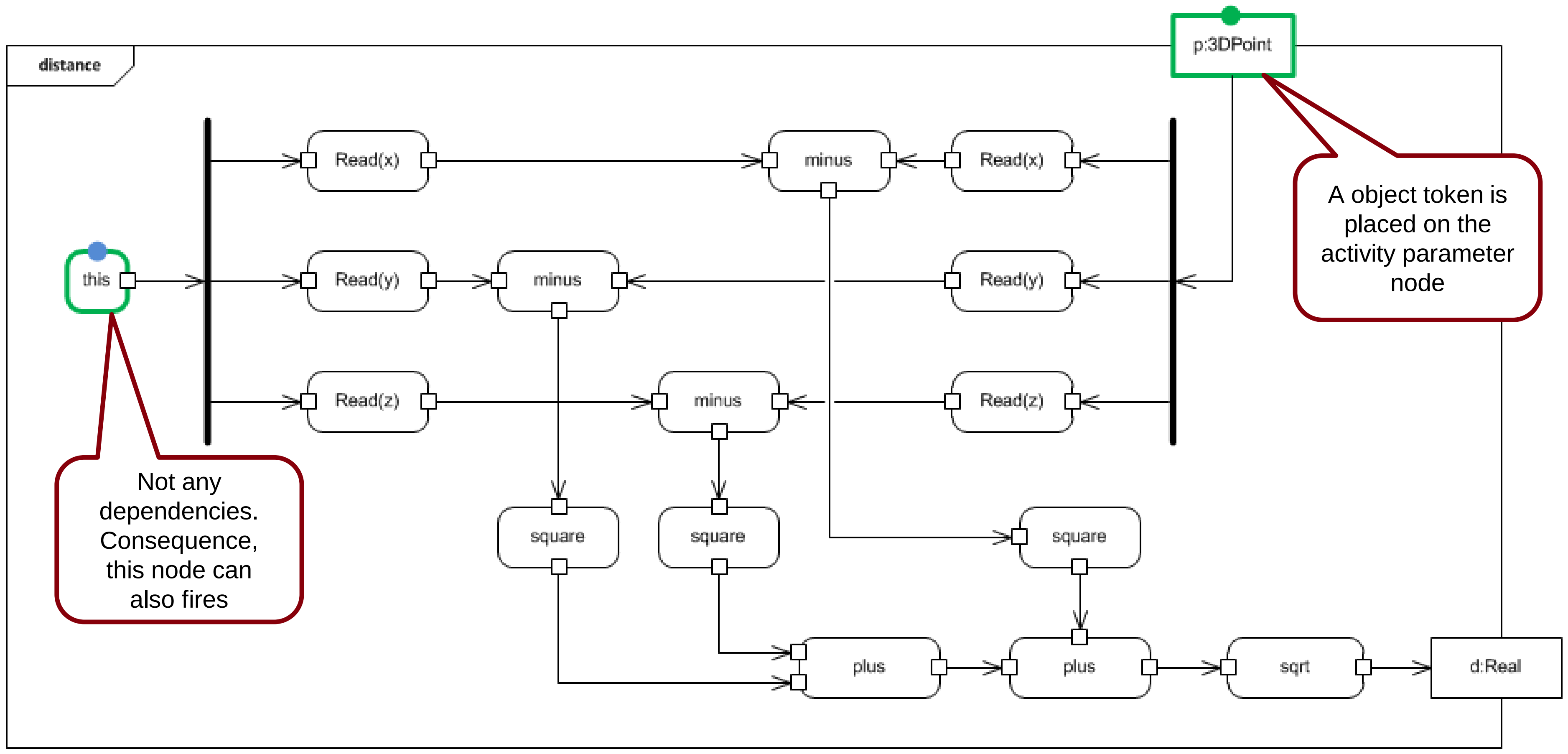
CONTENT



Objective

- Implement the “distance” operation as an activity
- Syntax elements that need to be used
  - Read Self Action
  - Read Structural Feature Action
  - Call Behavior Action
  - Activity Parameter Node
  - Input Pin
  - Output Pin
  - Control Flow
  - Object Flow
  - Fork Node

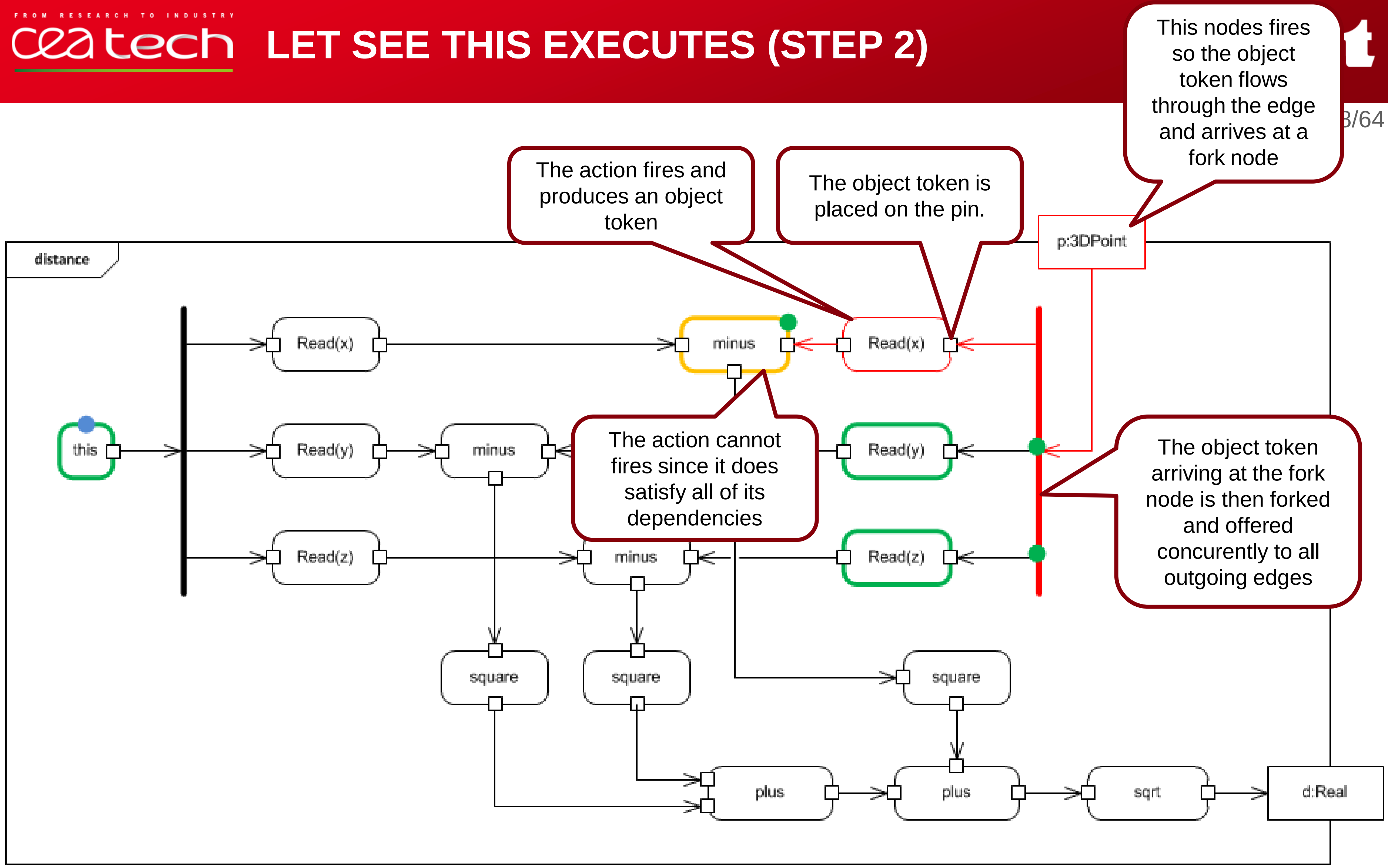
LET SEE THIS EXECUTES (STEP 1)



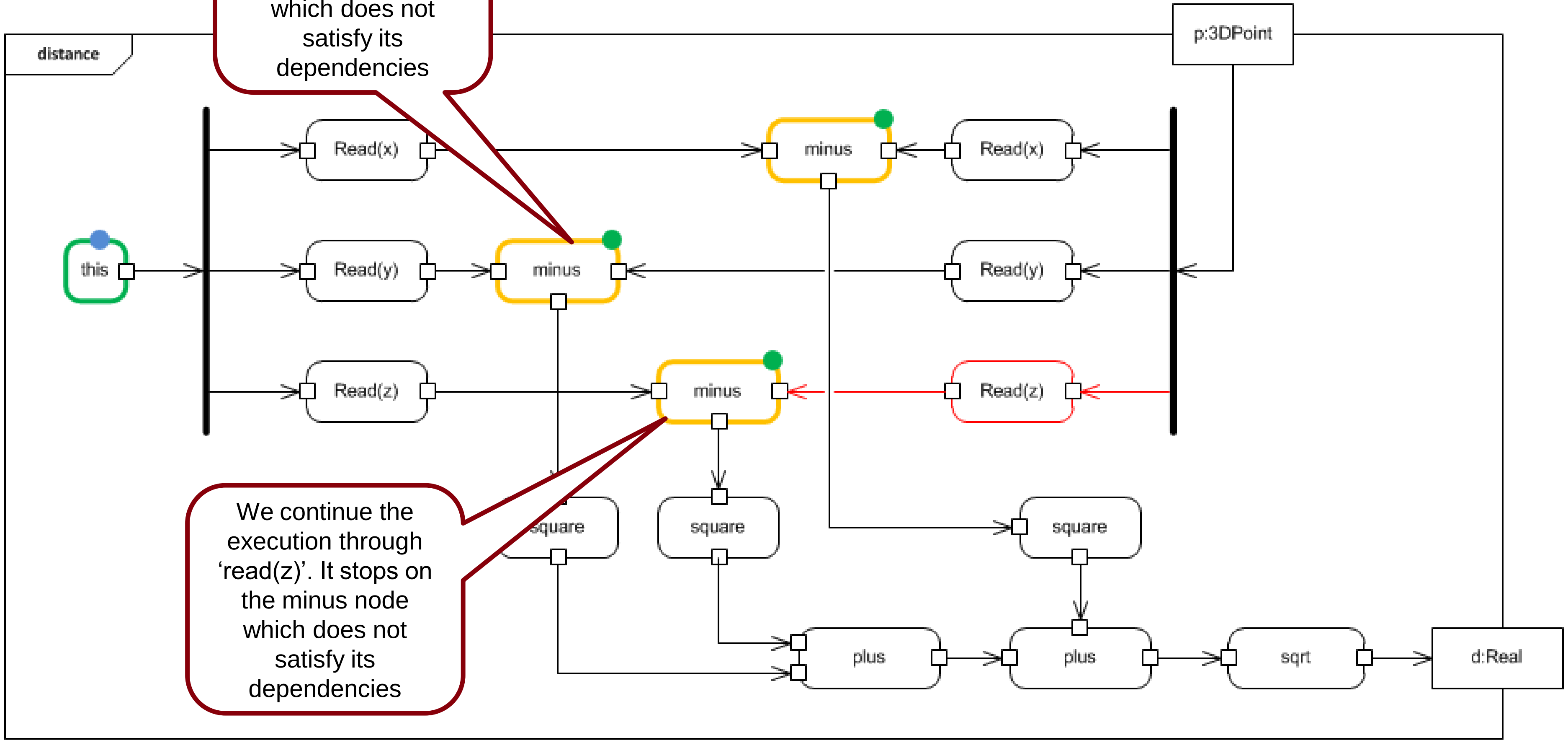
A object token is placed on the activity parameter node

Not any dependencies. Consequence, this node can also fires

LET SEE THIS EXECUTES (STEP 2)


This nodes fires so the object token flows through the edge and arrives at a fork node
The action fires and produces an object token
The object token is placed on the pin.
The action cannot fires since it does satisfy all of its dependencies
The object token arriving at the fork node is then forked and offered concurrently to all outgoing edges
distance
p:3DPoint
this
Read(x)
Read(y)
Read(z)
minus
minus
minus
Read(x)
Read(y)
Read(z)
square
square
square
plus
plus
sqrt
d:Real

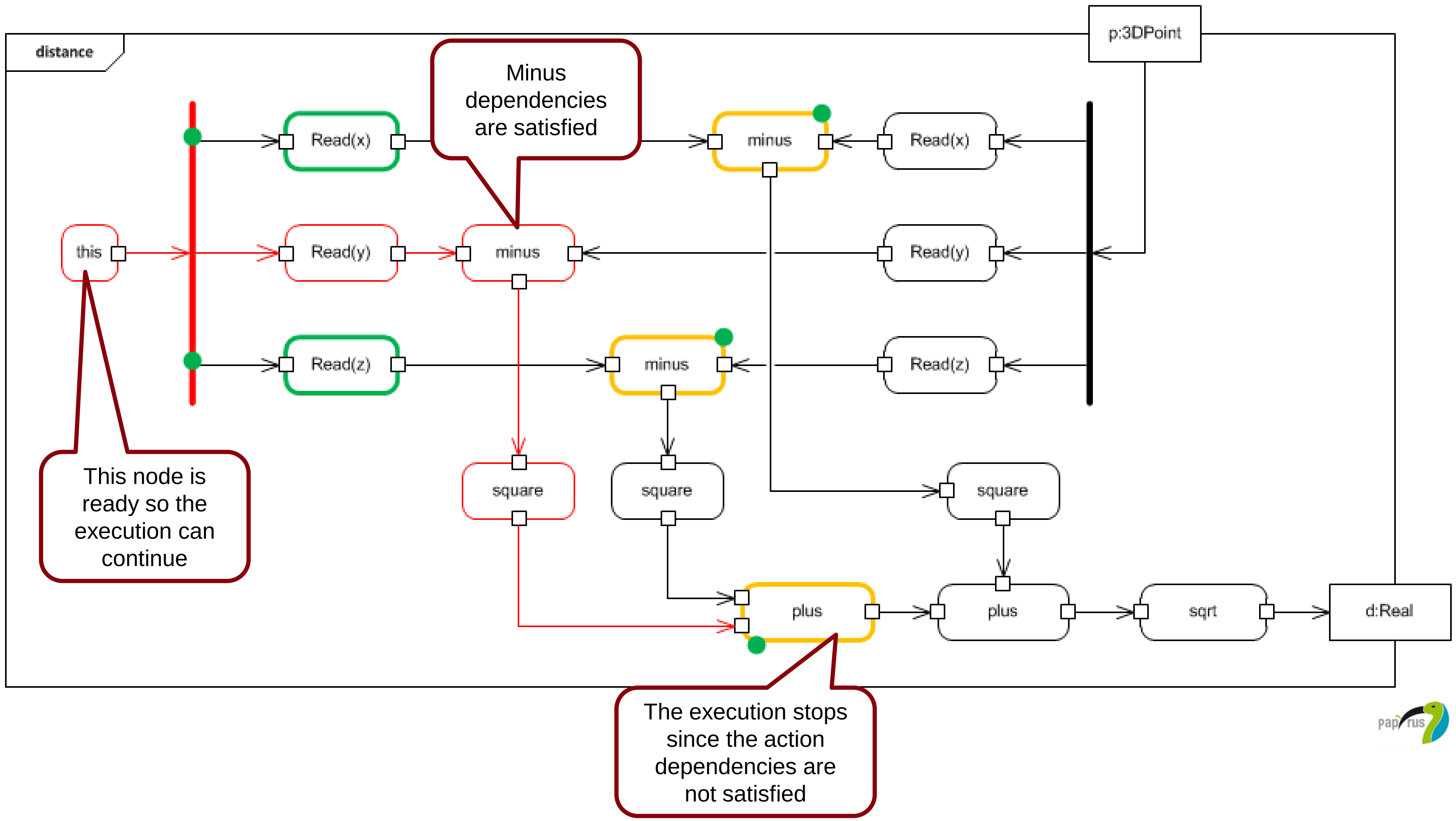
LET SEE THIS EXECUTES (STEP 3)



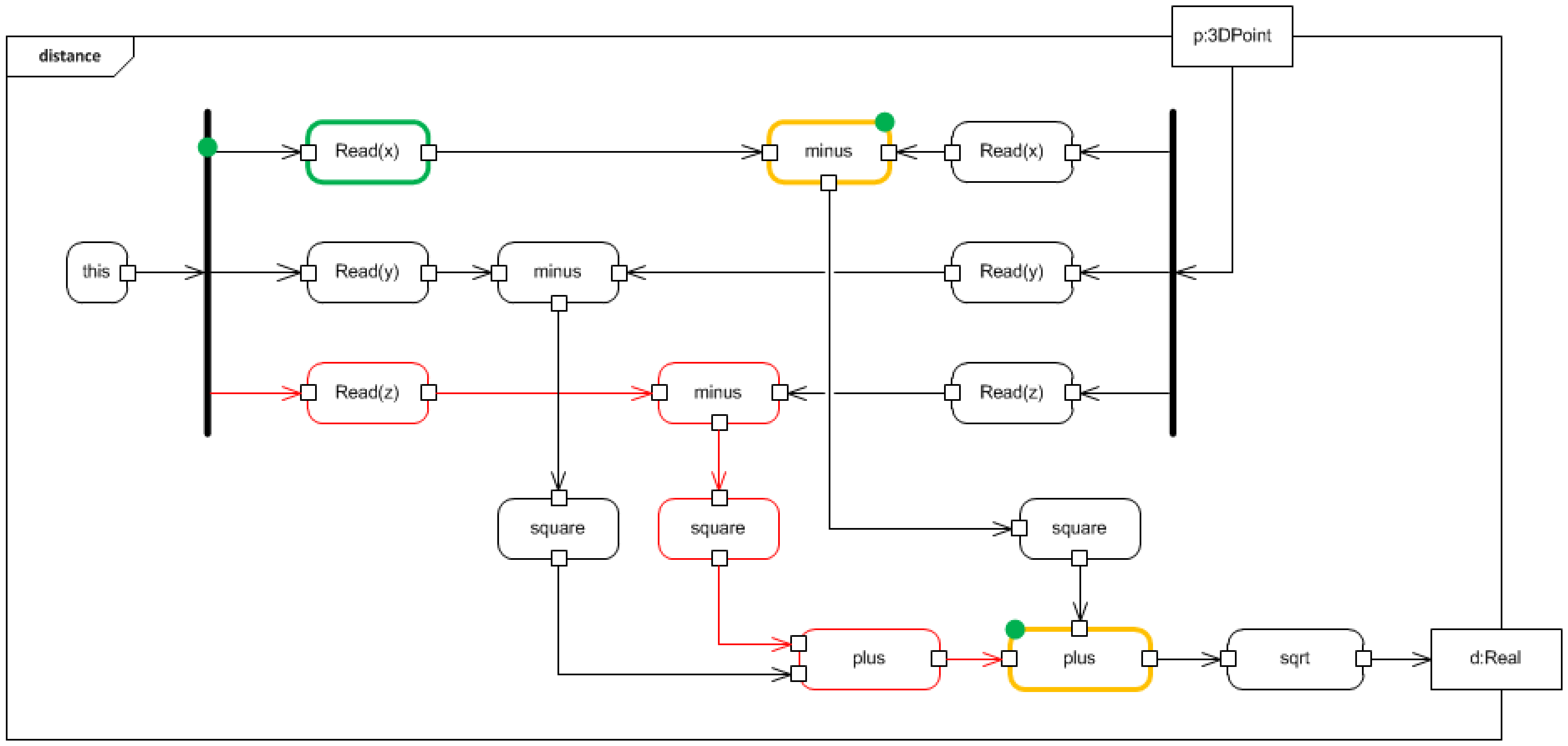
We continue the execution through 'read(y)'. It stops on the minus node which does not satisfy its dependencies

We continue the execution through 'read(z)'. It stops on the minus node which does not satisfy its dependencies

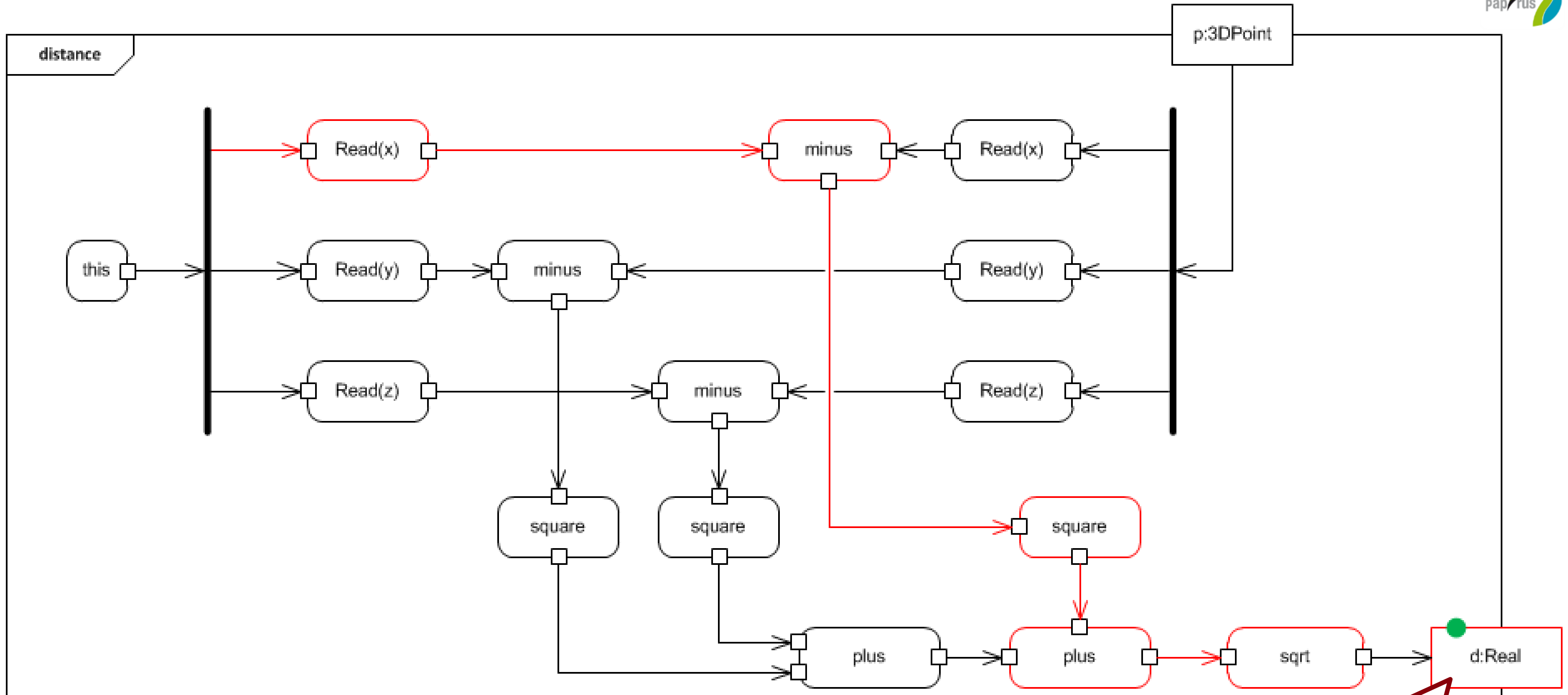
LET SEE THIS EXECUTES (STEP 4)


distance
p:3DPoint
this
Read(x)
Read(y)
Read(z)
minus
minus
minus
Read(x)
Read(y)
Read(z)
square
square
square
plus
plus
sqrt
d:Real
Minus dependencies are satisfied
This node is ready so the execution can continue
The execution stops since the action dependencies are not satisfied

LET SEE THIS EXECUTES (STEP 5)



LET SEE THIS EXECUTES (STEP 6)



The object token referencing the result of our calculation is then delivered on the activity parameter node 'd'