A GUIDE TO TEACHING ALLOY 6

WHY A TEACHING MODULE ON ALLOY 6?

A teaching module is a significant, highly homogeneous and unified part of a planned disciplinary program, having well-defined verifiable objectives based on building up skills and knowledge in discrete units.

Methodology

To create an engaging method that makes students more accountable and calls them in for self-assessment on their understanding of new concepts and for solving exercises and real problems.

Ability

To make SE students being able to develop good software that can be delivered within specific requirements using a formal specification language like Alloy.

Usability

Alloy 6 is useful, especially with the new temporal logic introduced in the sixth version.

THE PROBLEM

The fundamental aspect of the definition of a teaching module is the **identification of learning objectives** so that students understand what is required of them. In order to get students more deeply involved in the learning process, it is necessary to **adopt a strategy that allows a "balance of power"** between students and teachers, **making lessons interactive and dynamic**. A successful lesson requires the introduction of **alternative teaching strategies** that allow students to test themselves and assess their own level of understanding during the lesson.

THE SOLUTION

SCHEDULE

Several didactic forms were used to define an effective Alloy 6 teaching module, all listed here on the left with the suggested schedule.

→ **Lessons**
→ **Exercises**
→ **Challenge**

First Theoretical Lecture: **Alloy 5 vs. Alloy 6 / Alloy 6 intro, LTL, var keyword**	45min
Flipped Classroom: **Alloy 6 temporal operators**	15min
Second Theoretical Lecture: **Quiz / Alloy 6 new visualizer, time horizon**	45min
Exercise Lecture: **3 summary exercises on Alloy 6**	90min
Home Exercise	25min
Challenge	2weeks

LESSONS

45 MIN

First Theoretical Lecture

PRE-REQUISITES

First-order logic and Alloy 5

MATERIALS

Powerpoint slides

STRATEGIES

Structuring lesson, Setting goals, Traffic-light questioning, feedback

TOPICS

- Overview of learning objectives (5 min.)
- Static vs. Dynamic world (5 min.)
- How to represent dynamic models in Alloy 5 (20 min.)
- Alloy 6 (10 min.)

LEARNING OUTCOMES

- How Alloy 5 deals with dynamic modelling
- Limitations of dynamic modelling in Alloy 5 and the need of a new version

15 MIN

Flipped Lecture

PRE-REQUISITES

First Lecture

MATERIALS

Powerpoint slides and Video

STRATEGIES

Explicit-teaching, questioning and feedback.

TOPICS

- Alloy 6 temporal operators (past and future) (15 min.)

LEARNING OUTCOMES

- Usage of Alloy 6 temporal connectives with awareness
- Ability to deal with Alloy 6 new keywords

45 MIN

Second Theoretical Lecture

PRE-REQUISITES

First lecture and Flipped lecture

MATERIALS

Powerpoint slides

STRATEGIES

Structuring lesson, peer and teacher feedback (quizzes), traffic-light questioning

TOPICS

- Quizzes about the previous and flipped lecture content (20 min.)
- Alloy 6 time horizon, new visualizer and concurrency (25 min.)

LEARNING OUTCOMES

- Self evaluation of Alloy 6 through quizzes
- Knowledge of Alloy 6 concepts and possibilities

90 MIN

Exercise Lecture

PRE-REQUISITES

All lectures (first, flipped and second)
AlloyTools installed

MATERIALS

Powerpoint slides,
.als source codes

STRATEGIES

Worked-example, feedback and questioning

TOPICS

- Ex. 1: Concurrent communication in distributed systems (15 min.)
- Ex. 2: Travel (interrail) (30 min.)
- Ex. 3: Mailbox (20 min.)

LEARNING OUTCOMES

- Knowledge of Alloy 6 language and Alloy Tools
- Ability to model scenarios

2 weeks

Challenge

PRE-REQUISITES

All lectures (first, flipped and second) and exercises

MATERIALS

Powerpoint slides

STRATEGIES

Collaborative-learning, team-working, problem-solving

TOPICS

- Software-Defined Network

LEARNING OUTCOMES

- Ability to model a realistic and complex scenarios

MODULE MATERIAL

You can find all the materials by scanner this QRCode

ALLOY 6

A MATTER OF TIME

Authors:
Luca Padalino
Francesca Pia Panaccione
Francesco Santambrogio

https://github.com/AlloyTools/org.alloytools.alloy/releases

Alloy 6: there is an **implicit**, built-in notion of **(discrete) time**

1. Linear temporal logic
2. Mutable signatures and fields
3. Temporal operators
4. Time horizon
5. New visualizer
6. Cocurrency

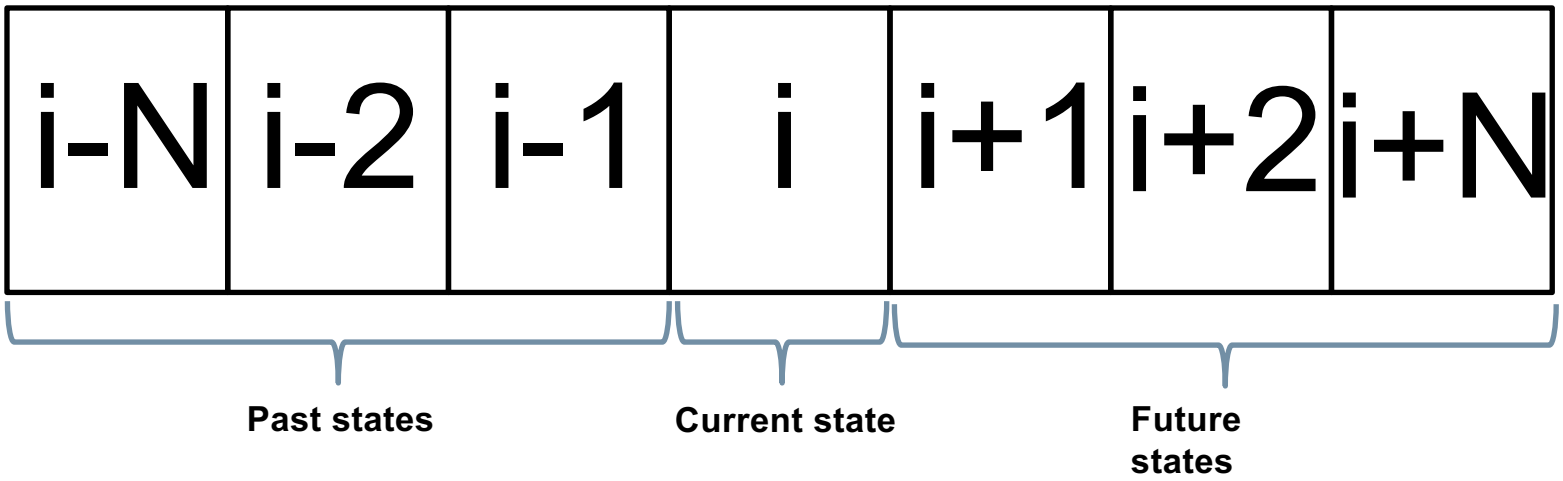
ALLOY 6-Syntactic overview

TEMPORAL CONNECTIVES

Authors:
Luca Padalino
Francesca Pia Panaccione
Francesco Santambrogio

LINEAR TEMPORAL LOGIC (LTL)
Definition

LINEAR TEMPORAL LOGIC (LTL): *«an infinite sequence of states where each point in time has a unique successor, based on a linear-time perspective»*[1]



[1] Ashari, R., & Habib, S. (n.d.). *LINEAR TEMPORAL LOGIC (LTL)*.

SIGNATURES

```
sig Person {
var liveness: Liveness
}

enum Liveness {Alive, Dead}
```

PREDICATES

```
pred Die [p: Person] {
p.liveness = Alive
p.liveness' = Dead }
```

To express constraints that hold at different instants of time or for a certain amount of time

- How can we express that a person is not immortal?
- How can we express that a person cannot come back to life?

To express constraints that hold at different instants of time or for a certain amount of time

- How can we express that a person is not immortal?
- How can we express that a person cannot come back to life?

TEMPORAL CONNECTIVES

ALWAYS

```
fact NoResurrection {
always (all p:Person|
p.liveness = Dead
implies always
p.liveness = Dead)}
```

AFTER

```
assert NoResurrection {
always (all p:Person|
p.liveness = Dead
implies after
p.liveness = Dead)}
```

EVENTUALLY

```
fact noImmortality {
always (all p:Person |
p.liveness = Alive implies
after (eventually
p.liveness = Dead))}
}
```

ALWAYS + EVENTUALLY

```
pred Mortality1 {
all p:Person |
always eventually
p.liveness = Dead
}
pred Mortality2 {
all p:Person |
eventually always
p.liveness = Dead
}
```

HISTORICALLY

```
fact NoDeadThenAlive {
always (all p:Person |
p.liveness = Alive
implies historically
p.liveness = Alive)}
```

ONCE

```
fact DeadSinceDeath {
always (all p:Person |
p.liveness = Dead implies
once Die [p])}
```

BEFORE

```
assert IfAliveBeforeAlive {
after
(always (all p:Person |
p.liveness = Alive implies
before p.liveness = Alive)
)}
```

UNTIL

```
fact AliveUntilDeath {
always (all p:Person |
p.liveness = Alive implies
(p.liveness=Alive until
p.liveness = Dead))}
```

RELEASES

```
assert AliveUntilDeath2 {
always (all p:Person |
p.liveness = Alive implies
(Die[p] releases
p.liveness = Alive))}
```

TRIGGERED

```
assert DeadSinceDeath2 {
always (all p: Person |
p.liveness = Dead implies
(Die[p] triggered
p.liveness = Dead))}
```

SINCE

```
assert DeadSinceDeath2 {
always (all p: Person |
p.liveness = Dead implies
(p.liveness=Dead since
Die[p]))}
```

TEMPORAL CONNECTIVES

;

;

```
run{#Person = 4 and
some p: Person |
(p.liveness = Alive ;
p.liveness = Alive ;
p.liveness = Dead)}
for 5
```

ALLOY 6

EXERCISE LECTURE

Authors:
Luca Padalino
Francesca Pia Panaccione
Francesco Santambrogio

CONCURRENCY: a property of a system in which multiple processes or threads can run simultaneously or appear to be running simultaneously.

Scenarios that deal with **CONCURRENCY**:

- **distributed systems**
- multi-threading/multi-tasking applications
- data migrations…

CONCURRENCY: a property of a system in which multiple processes or threads can run simultaneously or appear to be running simultaneously.

Scenarios that deal with **CONCURRENCY**:

- **distributed systems:** a collection of independent components located on different systems, sharing messages in order to operate as a single unit.

EXERCISE 1

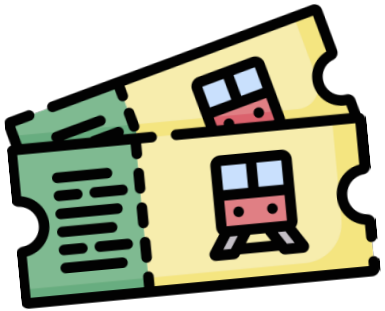
Concurrent communication in a distributed system


```
sig Node {
    edges: some Node
}

fact ConnectedGraph {
    all n1, n2: Node |
    n1 in n2.edges iff n2 in n1.edges and
    !(n1 = n2) and n1.*edges =Node
}

sig Message {
    var loc: Node
}

fact {
    all disj m1,m2: Message |
    always !(m1.loc = m2.loc)
}
```

```
pred send[m: Message, n: Node] {
    n in m.loc.edges
    no m2: Message | m2.loc = n
    m.loc' = n
}

pred sent[m: Message] {
    some n: Node | move[m, n]
}

pred unsent[m: Message] {
    m.loc = m.loc'
}

fact SendingOrNot {
    all m: Message | always moved [m]
}
```

EXERCISE 2
Interrail: general context

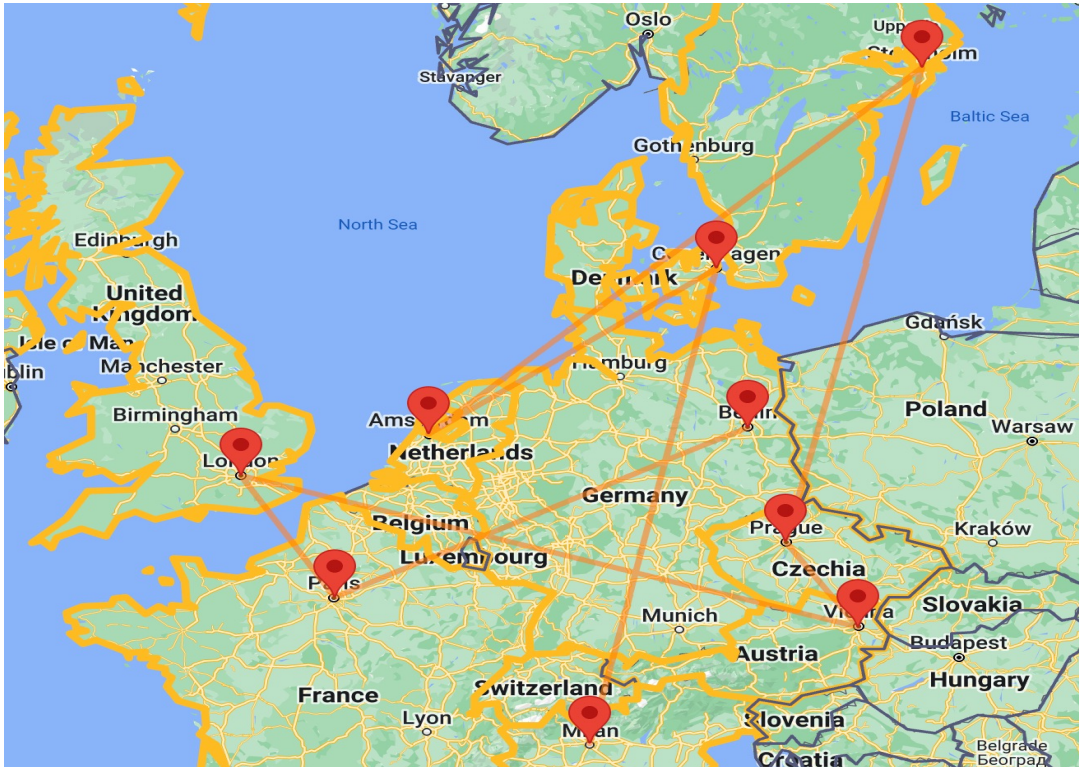
- A Person wants to organize an **interrail**.
- An interrail is a type of travel where the **traveler** can **move** to different **cities** by train.
- The person has a **pass** where all the cities he can visit are specified.
- The travel includes **European cities**.



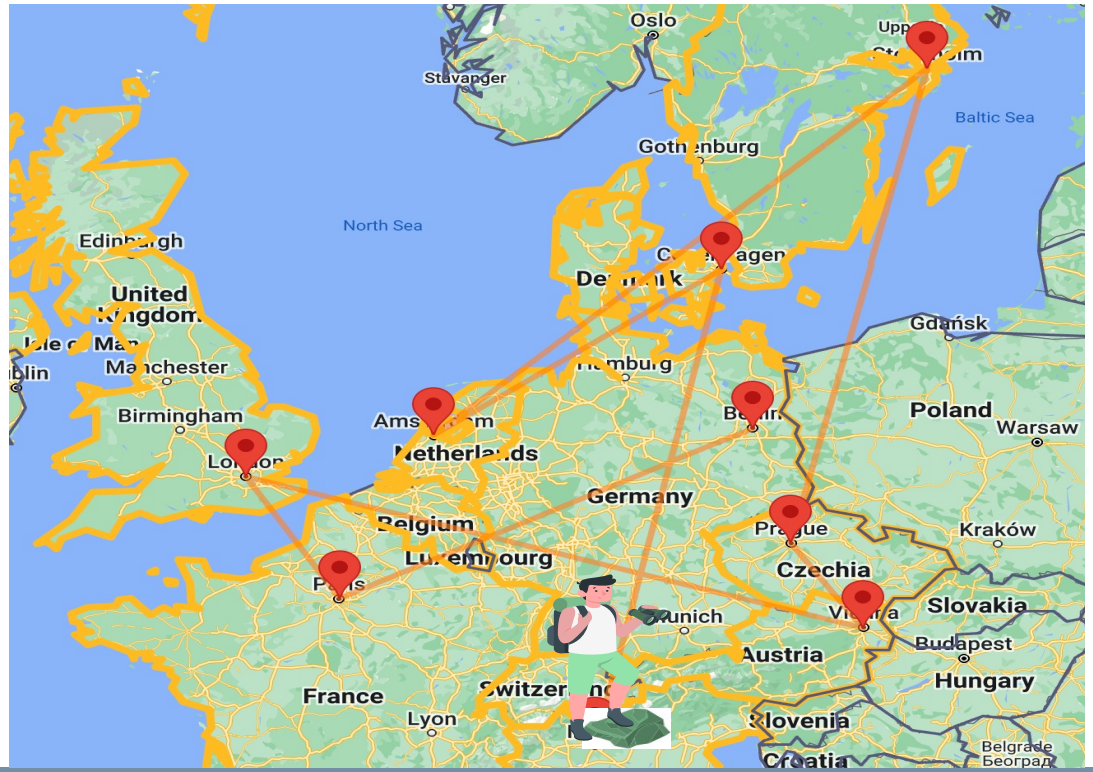
EXERCISE 2

Interrail: general context

- Cities must be **interconnected** through railways in order to create an itinerary.



- We assume that the Person can move only to the **cities** that **he has not visited yet**.



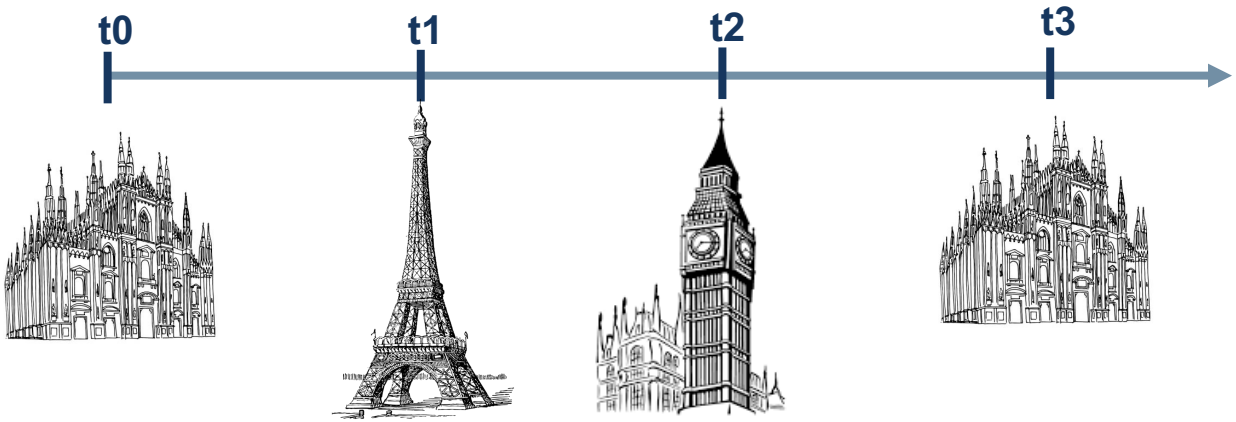
EXERCISE 2
Interrail: general context

- We want to keep track of whether the person **is traveling** or **has arrived** at the final destination.

EXERCISE 2
Interrail: signatures


```
sig City {
    linkedWith: some City,
    name: disj one EuropeanCities
}

enum EuropeanCities {
    Milan, Berlin, Paris, London,
    Vienna, Prague, Stockholm,
    Amsterdam, Copenaghen
}
```

EXERCISE 2
Interrail: signatures



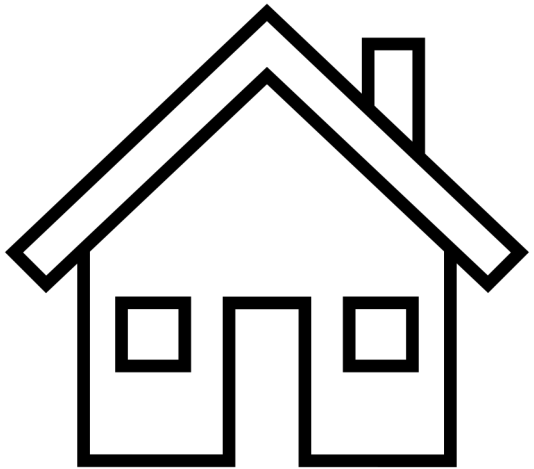

```
sig Person {
    pass: some City,
    var loc: City,
    var visited_cities: set City
    var status: Status
}

enum Status {
    Travelling, AtDestination
}
```

EXERCISE 2
Interrail: contraints

```
//Cities have connections with each other
fact BidirectionalConnection {
    all c1, c2: City | c1 in c2.linkedWith
    iff c2 in c1.linkedWith and !(c1 = c2)
    and c1.*linkedWith = City
}
```


```
//At the start of the interrail the person has not yet visited any cities
fact NoVisitedCitiesAtTheBeginning {
    all p:Person | no p.visited_cities
}
```

EXERCISE 3
Message Deletion from Mailbox



Model a scenario in which messages can be deleted from a mailbox and later restored.

Hint: the Message ends up in a Trash and can be restored from there.

EXERCISE 3
Message Deletion from Mailbox


```
var sig Message {}
var sig Trash in Message {}

pred delete[m: Message] {
    m not in Trash
    Trash' = Trash + m
    Message' = Message
}

pred restore[m: Message] {
    m in Trash
    Trash' = Trash - m
    Message' = Message
}
```

```
pred empty {
    #Trash > 0
    after #Trash = 0
}

pred doNothing {
    Message' = Message
    Trash' = Trash
}

pred restoreEnabled[m:Message] {
    m in Trash
}
```

EXERCISE 3
Message Deletion from Mailbox


```
fact Behaviour {
    no Trash
    always {
        (some m: Message | delete[m] or restore[m]) or empty or doNothing
    }
}

run {}
```

EXERCISE 3
Message Deletion from Mailbox


```
assert restoreAfterDelete {
    always (all m : Message | restore[m] implies once delete[m])
}

check restoreAfterDelete for 10 steps
check restoreAfterDelete for 1.. steps

assert deleteAll {
    always ((Message in Trash and empty) implies after always no Message)
}

check deleteAll
```

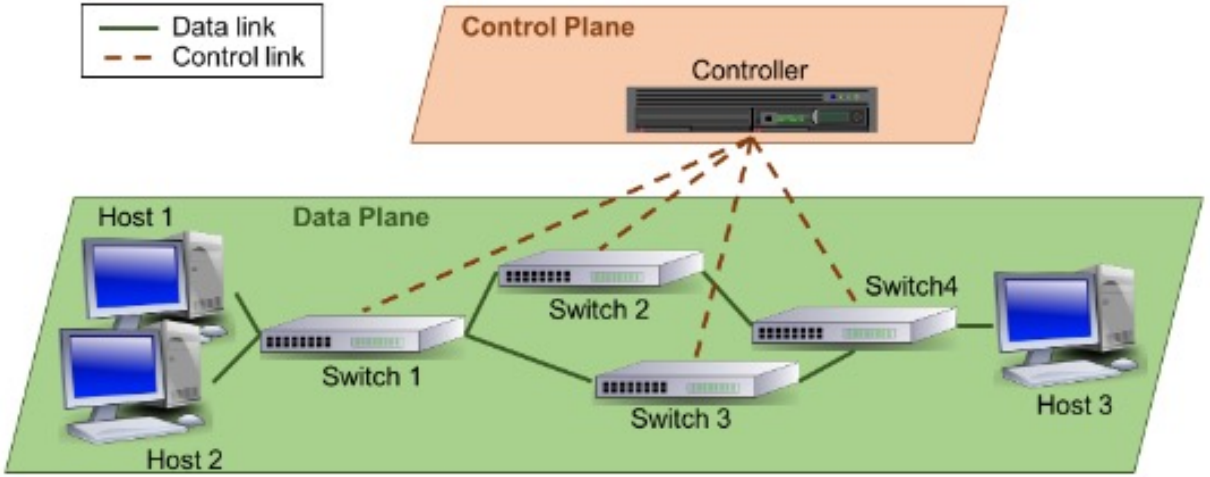
EXERCISE 3
Message Deletion from Mailbox

```
check MessagesNeverIncreases {
    always (Message' in Message and #Message' = #Message)
}

check IfNotDeletedMessagesNotEmpty {
    (always all m : Message | not delete[m]) implies always not empty
}

// a deleted message can still be restored if the trash is not empty
assert restoreIsPossibleBeforeEmpty {
    always (all m:Message | delete[m]
            implies after ((empty or restore[m])
            releases restoreEnabled[m]))
}

check restoreIsPossibleBeforeEmpty for 3 but 1.. steps
```

ALLOY 6
CHALLENGE

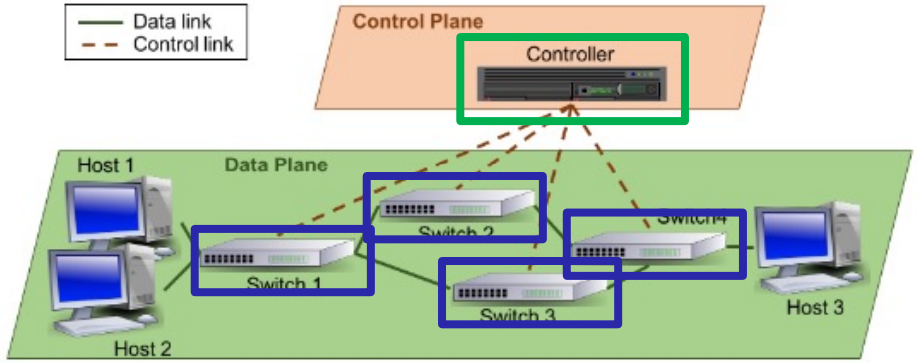
Authors:
Luca Padalino
Francesca Pia Panaccione
Francesco Santambrogio

Software-Defined Network (SDN): A is a modern networking paradigm that explicitly separates the data and control planes to include intelligence in the network.



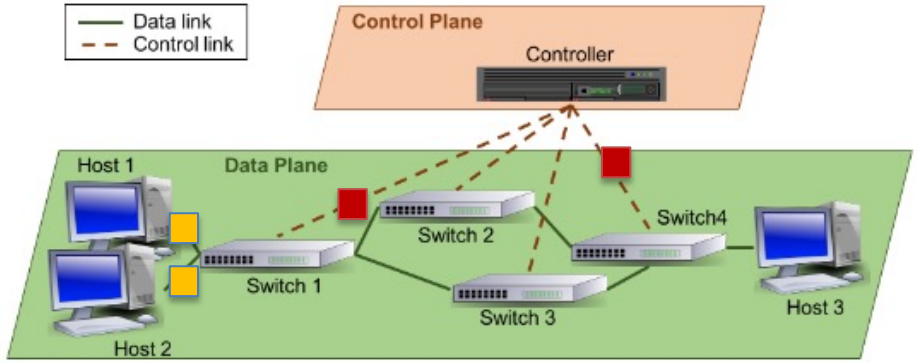
It is composed by these elements:

- **Controller**: core entity of the SDN control plane. The network intelligence is logically centralized in the controller, which is able to dynamically configure the forwarding devices of the data plane in order to achieve a specific goal;
- **Switches**: data plane components in charge of forwarding the **data packets** from its source to the destination. In SDNs, each switch has a **routing table** that contains a set of **rules** defining how the different incoming packets must be processed (forwarded, discarded, etc.). Each switch has also always a specific **link** (and thus a **port**) that connects it with the **controller**. This connection is mainly used to configure the switches. The data transmitted between nodes are called **packets**.



Packets that can be of two types:

- **control packets** include control plane information, such as new rules that must be installed in an specific switch, or a request to know how to process a data packet;
- **data packets** encapsulate information that must be transmitted from one host to another. In this case, the relevant information are the source and destination hosts, the type of data packets, and the current position in the network.



Switches contain **tables** with **rules** that specify how to route **data packets**. Simplifying, a **rule** is a structure with a field denoting the *type of* ***data packet*** *(e.g. HTTP or FTP)* and the **input** and **output ports**. The meaning of the rule is as follows:

- If a data packet of a particular type arrives at port *PortI*, it must be forwarded through port *PortO*;
- It is also possible to define rules that discard incoming data packets;
- When a switch has no rule to deal with a data packet, it sends a request to the controller in order to know how to process the packet;

Finally, the controller can also send new rules to switches to update the routing tables.

packetType	action	iPort	oPort
…	…	…	…
…	…	…	…

Hosts are the endpoints of the network, they are the source and destination of the data plane traffic. Although hosts are not specific elements of the SDN architecture, in our case study they are part of the model.