UNIVERSITÀ DEGLI STUDI
DI MILANO

Rule-Based Systems

A.I. for Video Games

Davie Gadia (gadia@di.unimi.it) and Dario Maggiorini (dario@di.unimi.it)

Rule-Based Systems

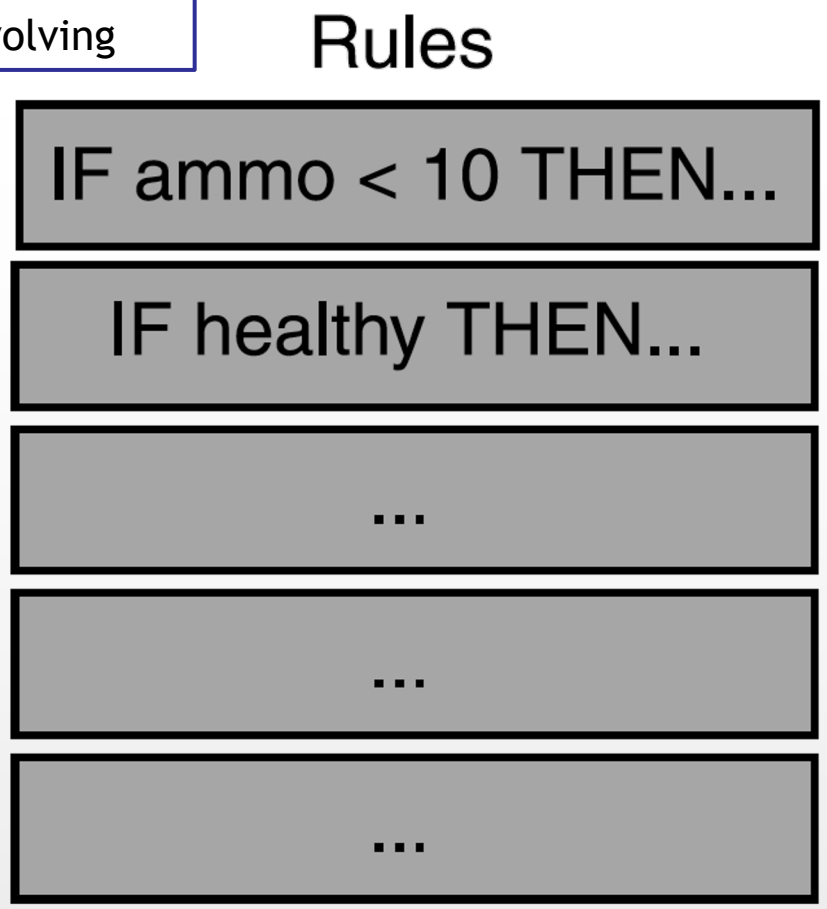
- Used to be a star in the AI firmament during ’70s and ’80s
  - Basic concept: we define the evolution of an agent by means of rules, bringing it from one state to another

- PRO
  - They come handy when an NPC must react to the environment in a way not foreseen by the game designer
    - When the environment is very complex, we cannot forecast all possible combinations

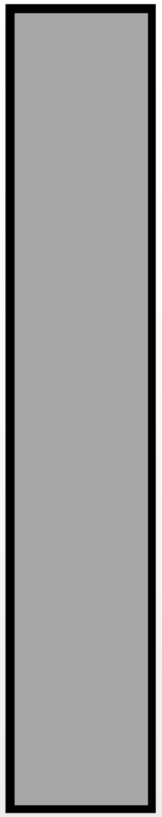
- CONS
  - Rules are difficult to implement
    - Some kind of high-level language must be used to describe them.
      A parser and an interpreter must be implemented in the engine
  - Same results can be achieved with DTs and FSMs
    - Of course, but using a much more complex schemes

Rule-Based System Architecture

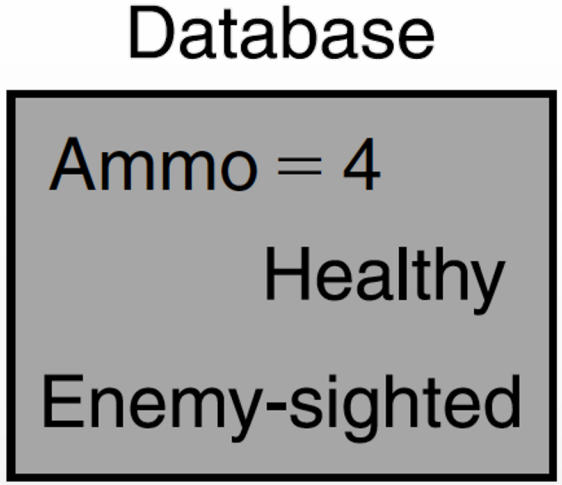
Rules define how the system is evolving



(Arbiter)



The database represents the internal and external knowledge



The arbiter is in charge to evaluate rules and perform actions

A Sample Implementation

This is a dummy implementation! You will not find any RBS folders in the source repository

```
public delegate bool RBSCondition();
public delegate void RBSAction();

public class RBSRule {

    private RBSCondition Condition;
    private RBSAction Action;

    public RBSRule(RBSCondition c, RBSAction a) {

        Condition = c;
        Action = a;
    }

    public bool Fire() {
        if (Condition ()) {
            Action();
            return true;
        }
        return false;
    }
}
```

Every rule has a condition and an action associated to it. The usual approach based on delegates is adopted here

When a rule is fired, if the condition is verified, then the action is executed

Maybe Too Simple?

```
public class RBS {

    public List<RBSRule> rules;

    public RBS() {
        rules = new List<RBSRule> ();
    }

    public void Run() {
        foreach (RBSRule r in rules) {
            if (r.Fire()) return;
        }
    }
}
```

Implementation of the arbiter is kind of straightforward

Where is This Difficult Implementation?

- Defining a syntax for rules is the actual problem
  - At least, it is language-related problem and not an AI problem
- In Unity, we can use delegates to tap on the C# language
  - Nevertheless, the code will not be accessible to an AI designer
- Using C# code is a problem if we want to put rules in a DBMS
  - We want to change the AI details without recompiling all the game
  - We are going to need a “pre-compiled” representation
  - We may want to change a rule on the fly during gameplay

About the Database

- Must contain all knowledge of the NPC
  - Both internal and external
- Must be able to contain any kind of game-related data
  - Remember, we are not talking about a DBMS
  - A set of couples (identifier, value) can be an acceptable solution

$$\text{Captain's-weapon} = \text{rifle}$$
$$\text{Johnson's-weapon} = \text{machine-gun}$$
$$\text{Captain's-rifle-ammo} = 36$$
$$\text{Johnson's-machine-gun-ammo} = 229$$

Structured Information

- Basic information is the Datum object

```
(identifier content)
```

Rediscovering "old" technologies!

- In a Datum, content can be either
  - A literal
    `(Ammo 36)`
  - Another Datum
    `(Rifle (Ammo 36))`
  - A list of Datum
    `(Rifle (Ammo 36) (Clips 2))`

This is LISP

- Nesting Datum, very complex information can be represented

```
( Captain
        (Weapon (Rifle (Ammo 36) (Clips 2)))
        (Health 65)
        (Position [21, 46, 92])
)
```

This is MONGO-DB

Datum in C#

```
public class RBSDatum {

    public string id;
    public object value; // a value or another datum

    public RBSDatum(string s) {
        id = s;
    }

    public override string ToString() {
        return "( " + id + " " + value.ToString() + " )";
    }

    public static bool Match(IRBSQuery q) {
        throw new Exception("unimplemented");

    }
}
```

Using object type we can store everything, including other Datum

Sorry guys, this is not a class about languages

A List of Datum to Create a Database

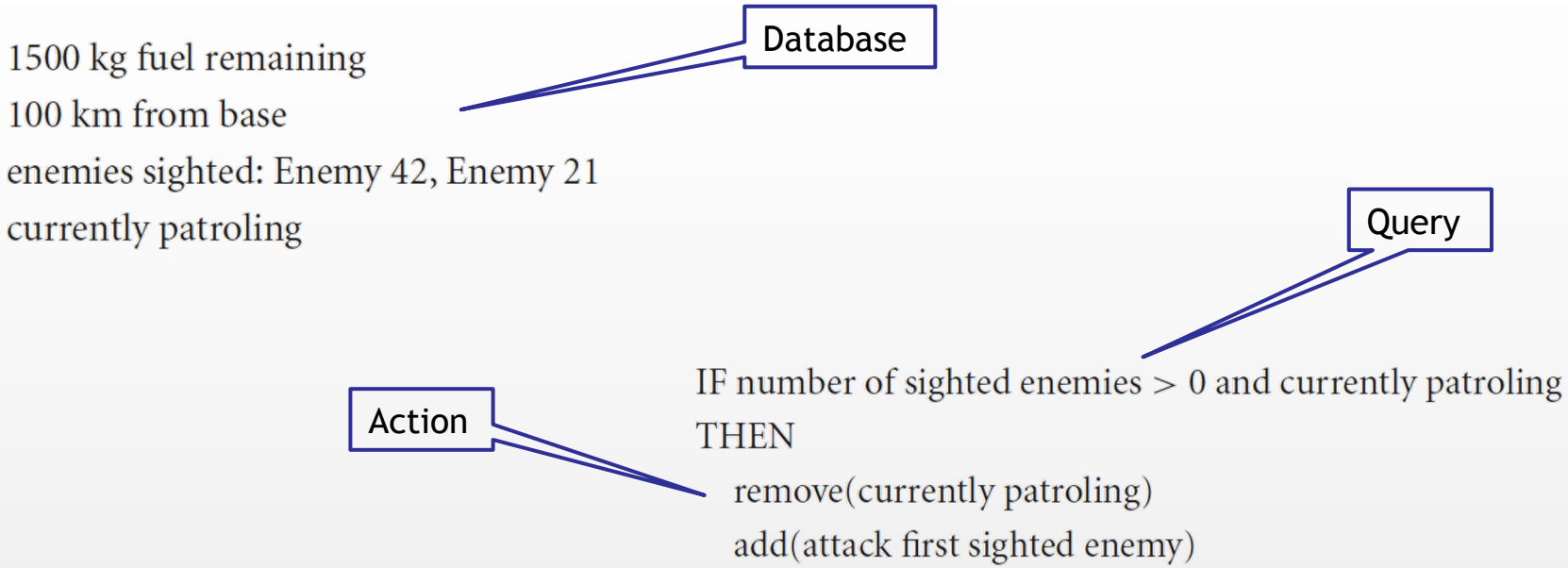
```
public class RBSDB
{

    public List<RBSDatum> data;

    public RBSDB() { ; }

    override public string ToString() {
        string r = "";
        foreach (RBSDatum d in data)
            r += d.ToString() + "\n";
        return r.Substring(0, r.Length - 1);
    }

    public static bool Match(IRBSQuery q) {
        foreach (RBSDatum d in data) {
            if (d.Match(q)) return true;
        }
        return false;
    }
}
```

A database is a list of Datum

Finding a match in the database means iterating on all the Datum and find a match to a query

Database Rewriting Rules

- An action associated to a rule can do anything, including changing the database



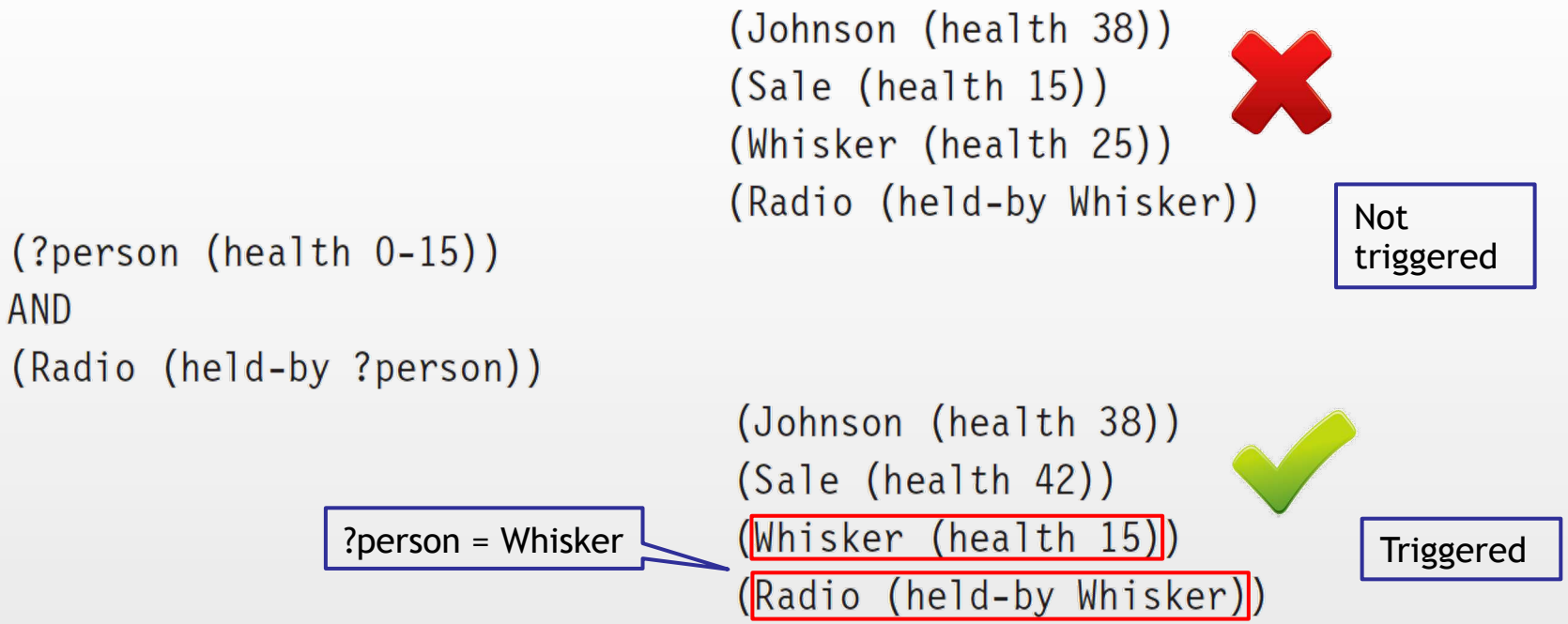
Rules Arbitration

- When more than one rule triggers, how do we pick the one to fire?

1. First applicable
   - The first match fires
2. Least Recently Used
   - Same as above, but evaluation order is based on timestamp of last firing
3. Random Rule
   - Pick a random one
4. Most Specific Conditions
   - Most specific first, based on number of verified clauses
5. Dynamic Priority Arbitration
   - A priority for each rule is assigned based on context.
     If the context changes, the priority will vary between evaluations

Rules Unification

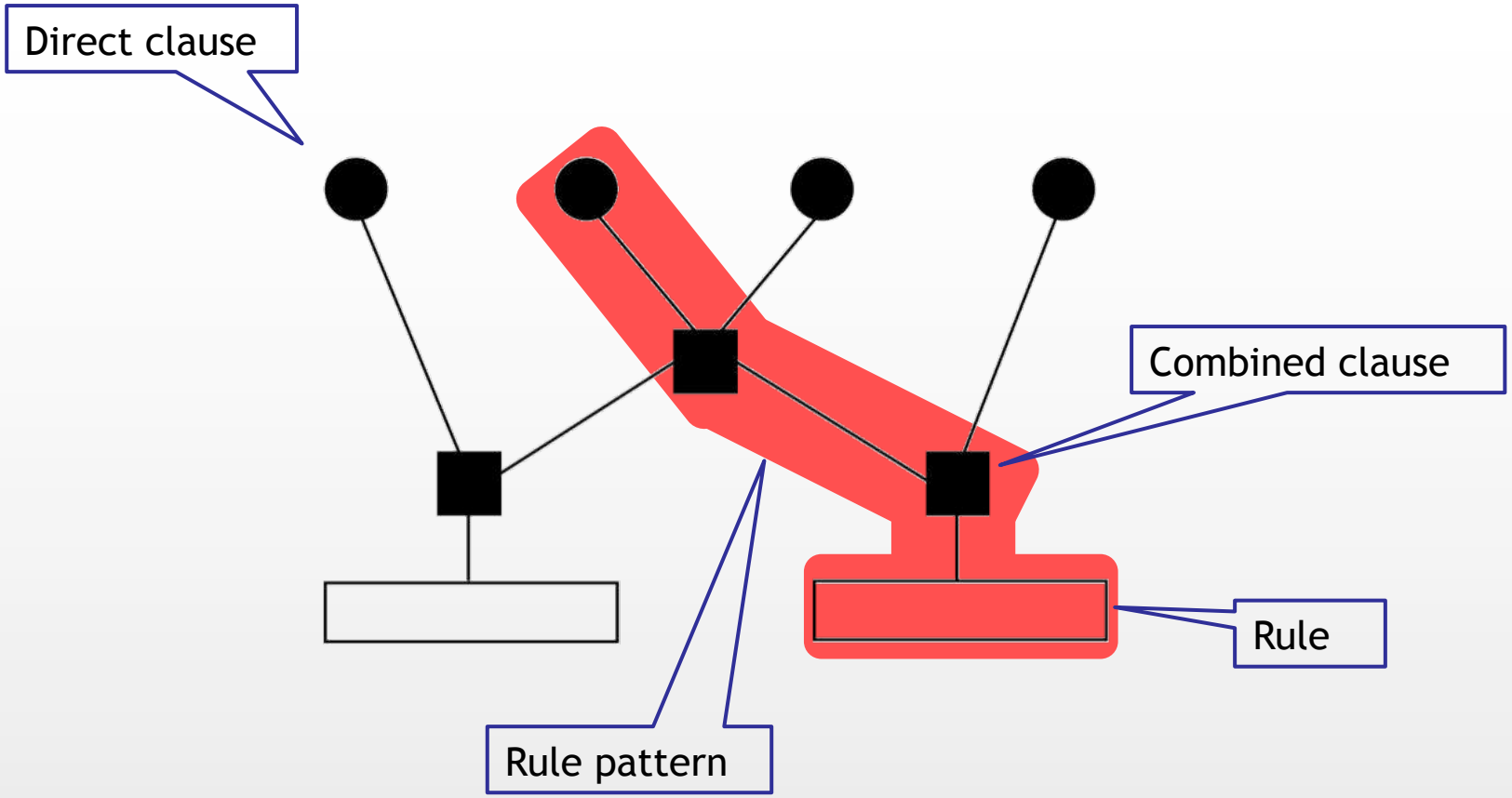
- It is useful to define a rule as a recurring pattern
  - We have no idea about who is holding the radio in the platoon, but if that soldier is low on health, we want to undertake an action



Rete

- Rete is an algorithm for matching rules against a database
  - It is a de-facto industry standard
- All rules are represented in a directed acyclic graph
  - Each node in the graph represents a pattern
  - Each path through the graph represents the complete set of patterns for one rule
- We identify three kinds of nodes
  - Direct clauses or pattern nodes (at top)
  - Joint or combined nodes (in the middle)
  - Rules (at bottom)

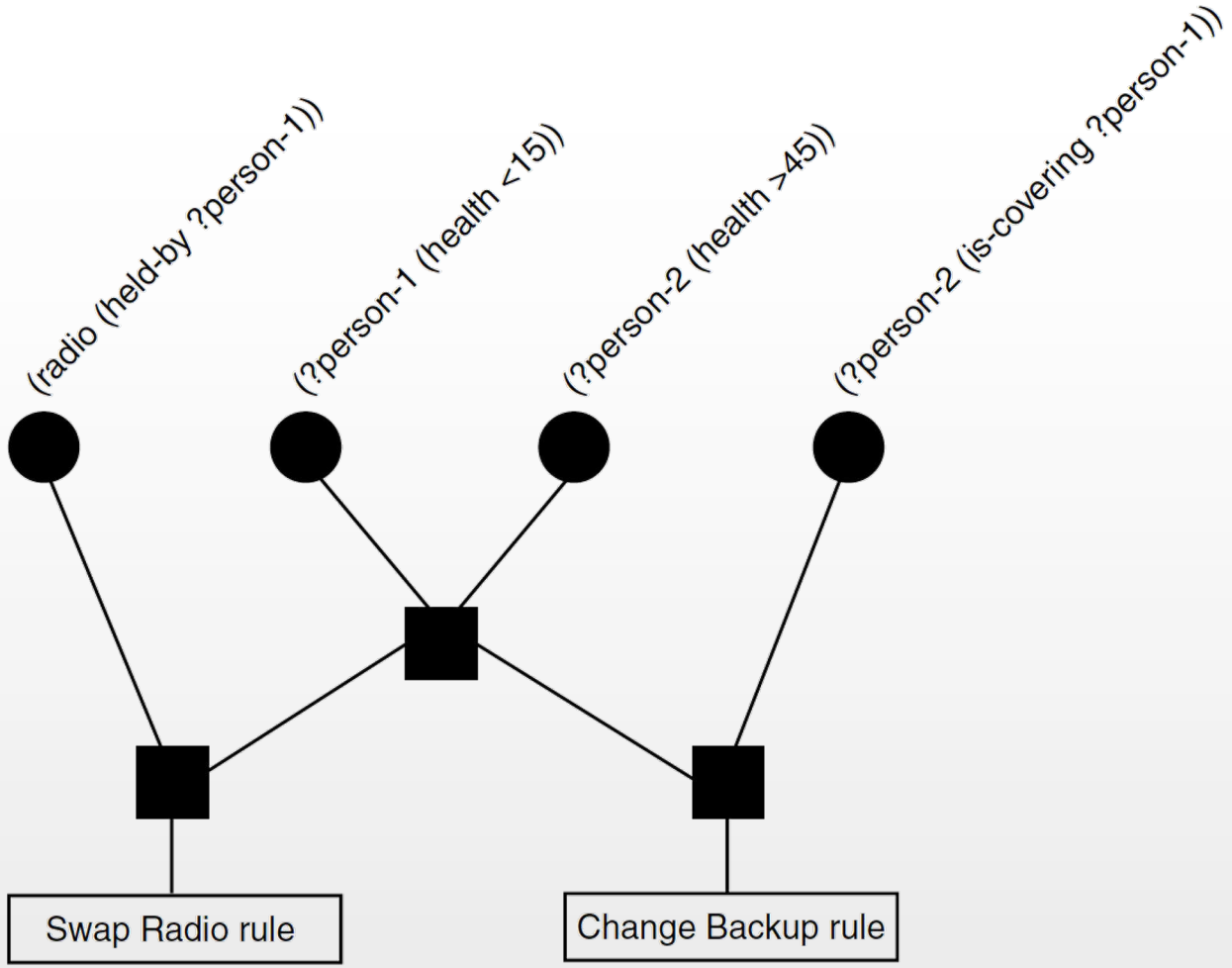
Graph



Rules

```
Swap Radio Rule:
IF
   (?person-1 (health < 15))
    AND
   (radio (held-by ?person-1))
    AND
   (?person-2 (health > 45))
THEN
   remove(radio (held-by ?person-1))
   add(radio (held-by ?person-2))

Change Backup Rule:
IF
   (?person-1 (health < 15))
    AND
   (?person-2 (health > 45))
    AND
   (?person-2 (is-covering ?person-1))
THEN
   remove(?person-2 (is-covering ?person-1))
   add(?person-1 (is-covering ?person-2))
```

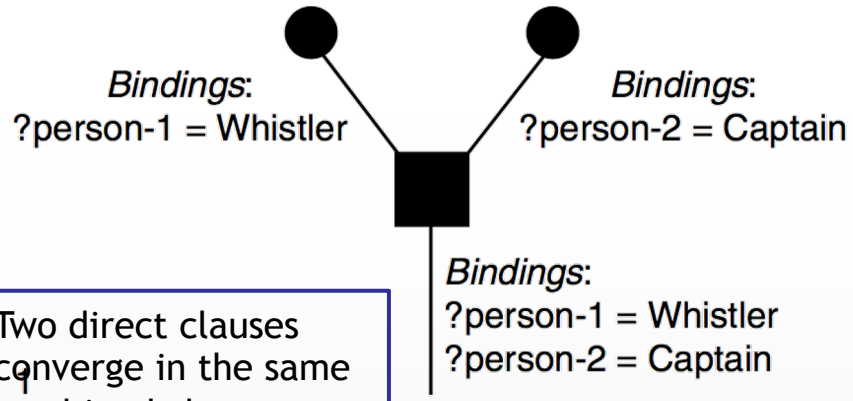
Graph


(radio (held-by ?person-1))
(?person-1 (health <15))
(?person-2 (health >45))
(?person-2 (is-covering ?person-1))
Swap Radio rule
Change Backup rule

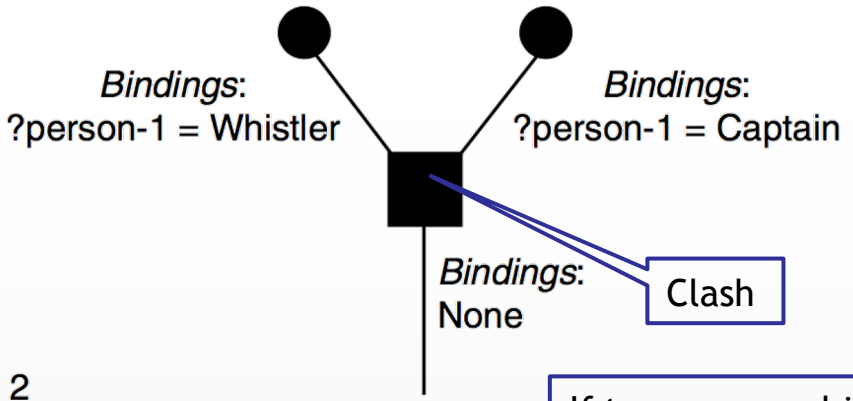
Matching a Rule

- The database is applied at the top of the graph
- Matching pattern nodes will propagate below
  - This includes variables binding
- Joint nodes will verify their input and propagate further below
  - Bindings must be consistent
- If we can reach an action, such action must be triggered

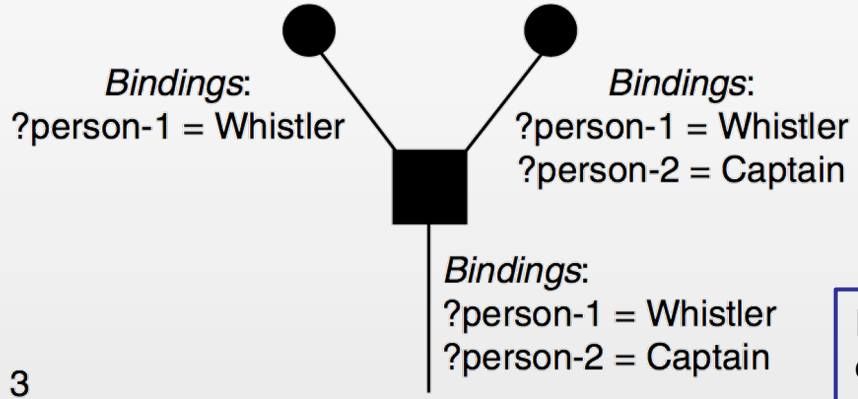
Example Using Rete



Two direct clauses converge in the same combined clause. The union of the bindings is propagated

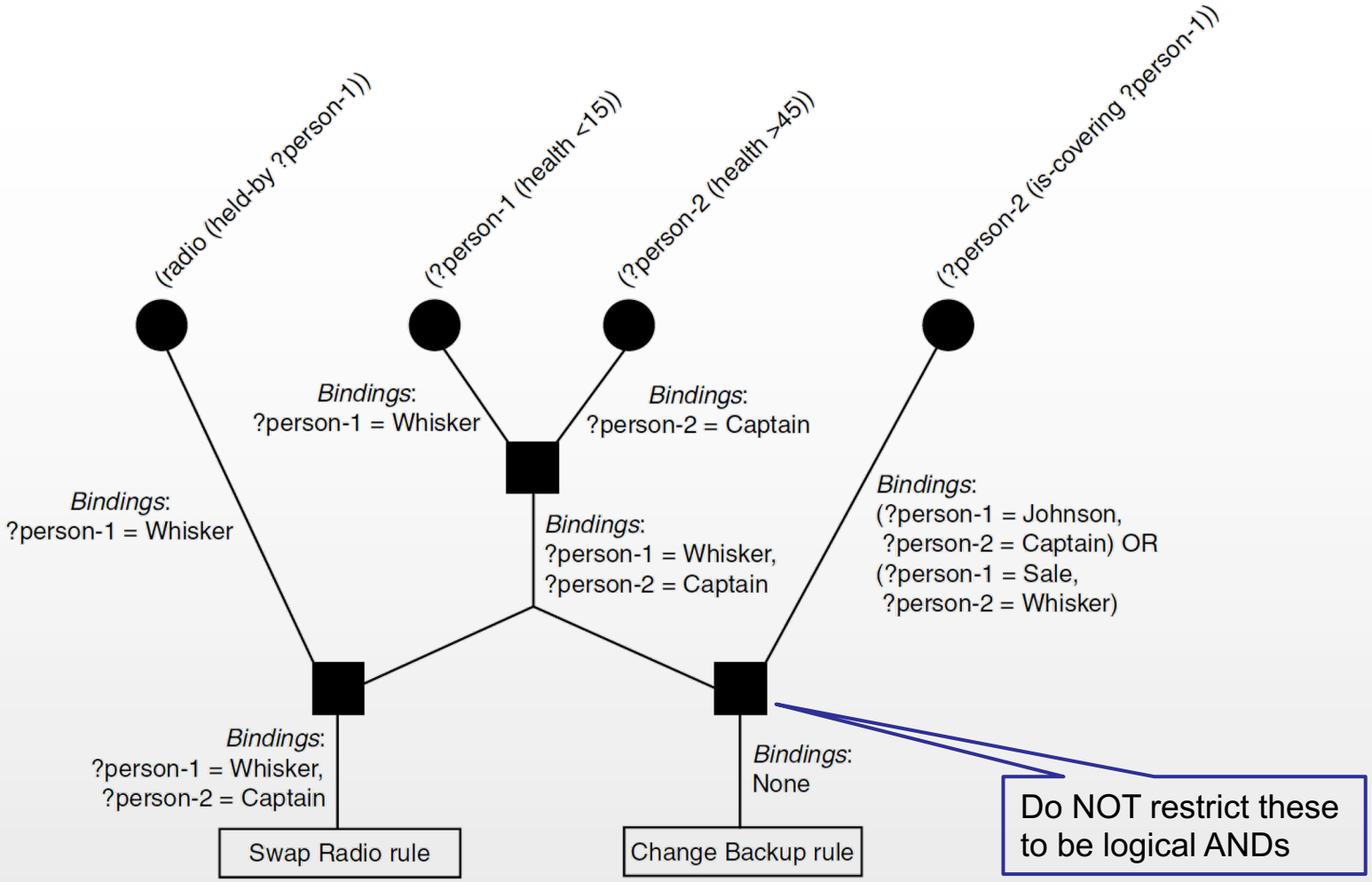


If two or more bindings are clashing, nothing is propagated and the rules traversing the combined clause will not trigger



If bindings are not clashing, the union is propagated

Example Using Rete



Why Should We Use Rete?

- Rete is designed to keep a memory of the current situation and update the data structure only if the database is changing
  - Think about an RBS where you have thousands of rules

- When adding or removing a fact from the database, only pattern nodes evaluating that fact will update and propagate
  - This is going to save a huge amount of computation

References

- On the book
  - § 5.8 (excluded 5.8.3, 5.8.4, 5.8.8 and 5.8.9)