neuroinformatics group

Behavior-Based Approach

Sascha Lange, Christian Müller, Stefan Welker

Brainstormers Tribots

Neuroinformatics Group
University of Osnabrück
Albrechstr. 28
49069 Osnabrück

Tel: +49 541 969-2390
Fax: +49 541 969-2246

http://www.tribots.uos.de
Email: tribots@informatik.uni-osnabrueck.de

brainstormers TRIBOTS
University of Osnabrück

PHILIPS
DUTCH OPEN ROBOCUP 2006
8
5
7
6
9
RST
fun
KIWI.
brainstormers
TRIBOTS
University of Osnabrück

Approach

- architectural requirements
  - behavior-based
  - modular
  - hierarchically
- feasibility requirements
  - purely reactive behavior should possible (right now 90% still is purely reactive)
  - other approaches should not be excluded (e.g. planning)
- more a collection of (unrelated) ideas than a complete theory

"this is not a big theory of behavior specification, but a framework to practically support the implementation. There is no abstract behavior specification language. it is a collection of classes, you make use of or derive your classes from and some „coding guidelines“ you should respect."

Skills / Behavior

- getCmd
- Gain / Loose Control
- callbacks

Skill / Behavior

+getCmd(Time&): DriveVector
+gainControl(Time&): void
+loseControl(Time&): void
+cycleCallback(Time&): void

- Skill: needs parameters, e.g. target position (DribbleToPos)
- Behavior: no parameters (DribbleToGoal)

Example behavior

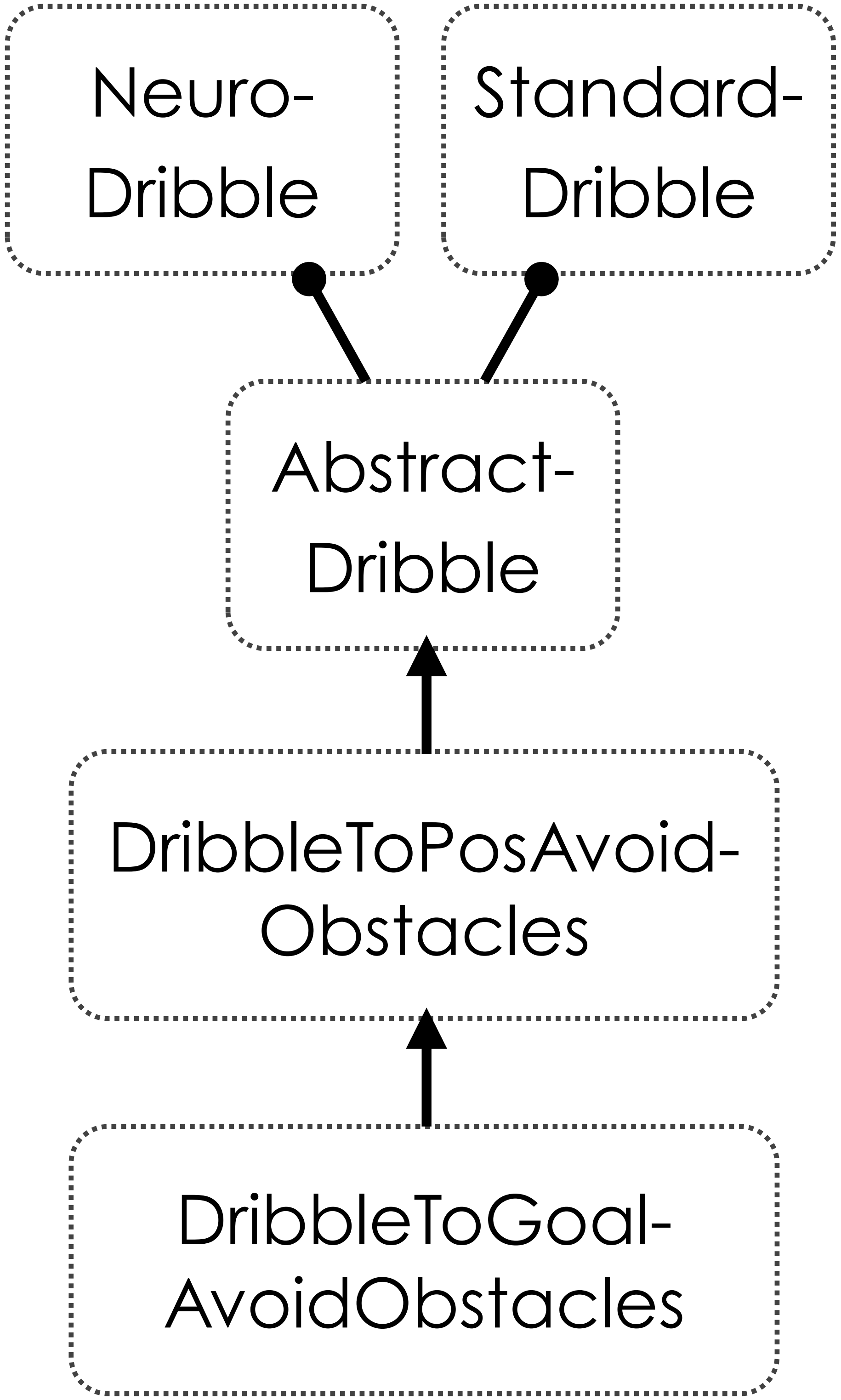
BDribbleBallToGoal

```
BDribbleBallToGoal::BDribbleBallToGoal()
  : Behavior("BDribbleBallToGoal"),
    skill(new SDribbleBallToPosRL())
{}

DriveVector BDribbleBallToGoal::getCmd(const Time& t)
throw(TribotsException) {

  //use information about the world to calculate
  //target position (in goal)
  FieldGeometry const& fgeom= MWM.get_field_geometry();
  Vec targetPos = Vec(0., fgeom.field_length / 2.);

  //use skill to produce drive commands
  skill->setParameters(targetPos, transVel);
  return skill->getCmd(t);
}
```

- behavior uses information about world to determine target position (e.g. avoid obstacles)
- skill always needs parameters to calculate drive command

Specialization by Inheritance

- Inheritance is used intensively, build functionality layer by layer (Matryoshka)
- Example 1:
  - Dribble - Just the handling of the ball, learned with NFQ
  - DribbleToGoal - adds obstacle avoidance and sets target to goal
- Example 2:
  - General defense behavior (cover ball, drive to it if possible)
  - Field player behavior derived, position and location to cover is adapted to overall strategy

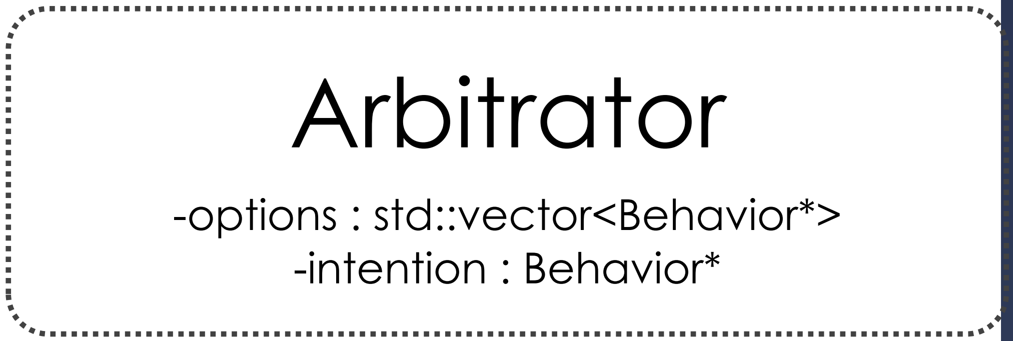


Arbitration

- Idea borrowed from BDI-architectures (MAS air-traffic controller)
- Interface extension to behaviors (conditions)
    - Invocation Condition (IC)
        - A behavior can take over control for the first time, if IC is fulfilled
        - Example EigenMove: ball possession close to side line
    - Commitment Condition (CC)
        - A behavior can keep control and does not have reached its goal, if this condition is met
        - Example Eigenmove: ball possession

Arbitration

- Use IC and CC for the generic arbitration of behaviors
- BDI-like Arbitrator
  - Belief: world model
  - Desire: drive command
  - Intention: active behavior



Arbitrator

-options : std::vector<Behavior*>
-intention : Behavior*

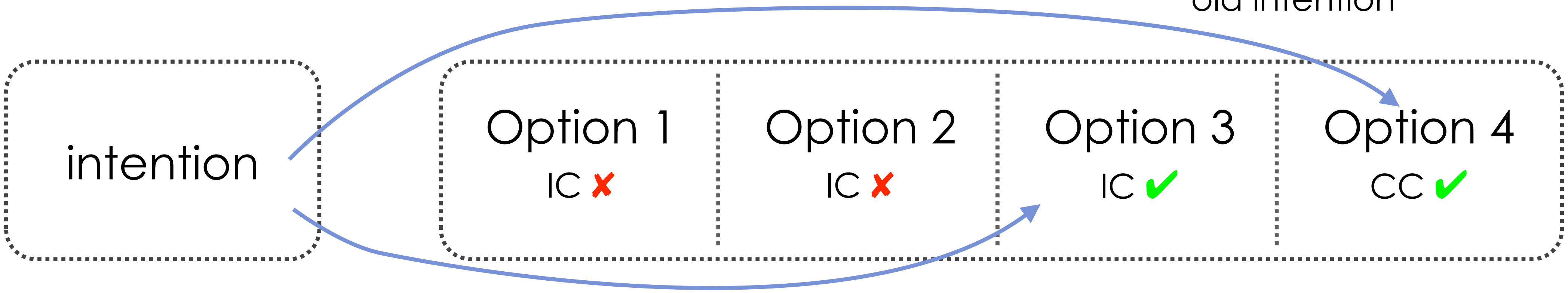
Types of Arbitration

- Highest Priority First (purely reactive, most used)
  - Check CC of the active intention (possibly remove intention and signal loseControl)
  - Run through list of options up to the active option (iff intention still active, otherwise up to the end) and check IC's:
    - If IC is true make the currently inspected option to the intention (signal gainControl)
  - Active intention is then called by getCmd()

Algorithm 1 The ”highest priority first” arbitration scheme.

```
Require: intention $\neq 0$
  if not intention.commitment_condition(t) then
    intention $\Leftarrow$ emergency_stop
  end if
  for $i = 0$ to options.length() do
    if options[$i$] $=$ intention then
      break
    end if
    if options[$i$].invocation_condition(t) then
      intention $\Leftarrow$ options[$i$]
      break
    end if
  end for
Ensure: intention $\neq 0$
```

old intention

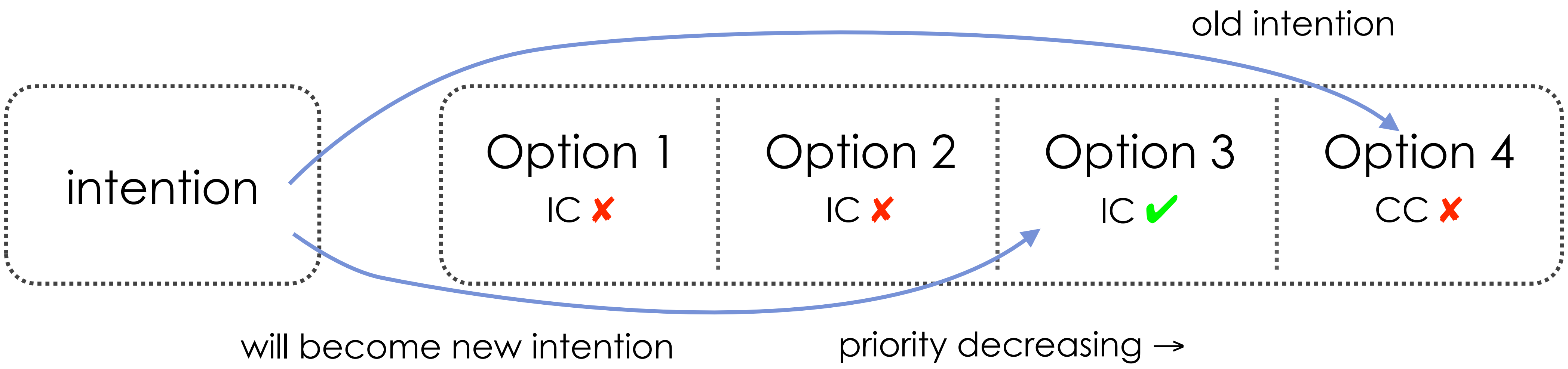


will become new intention

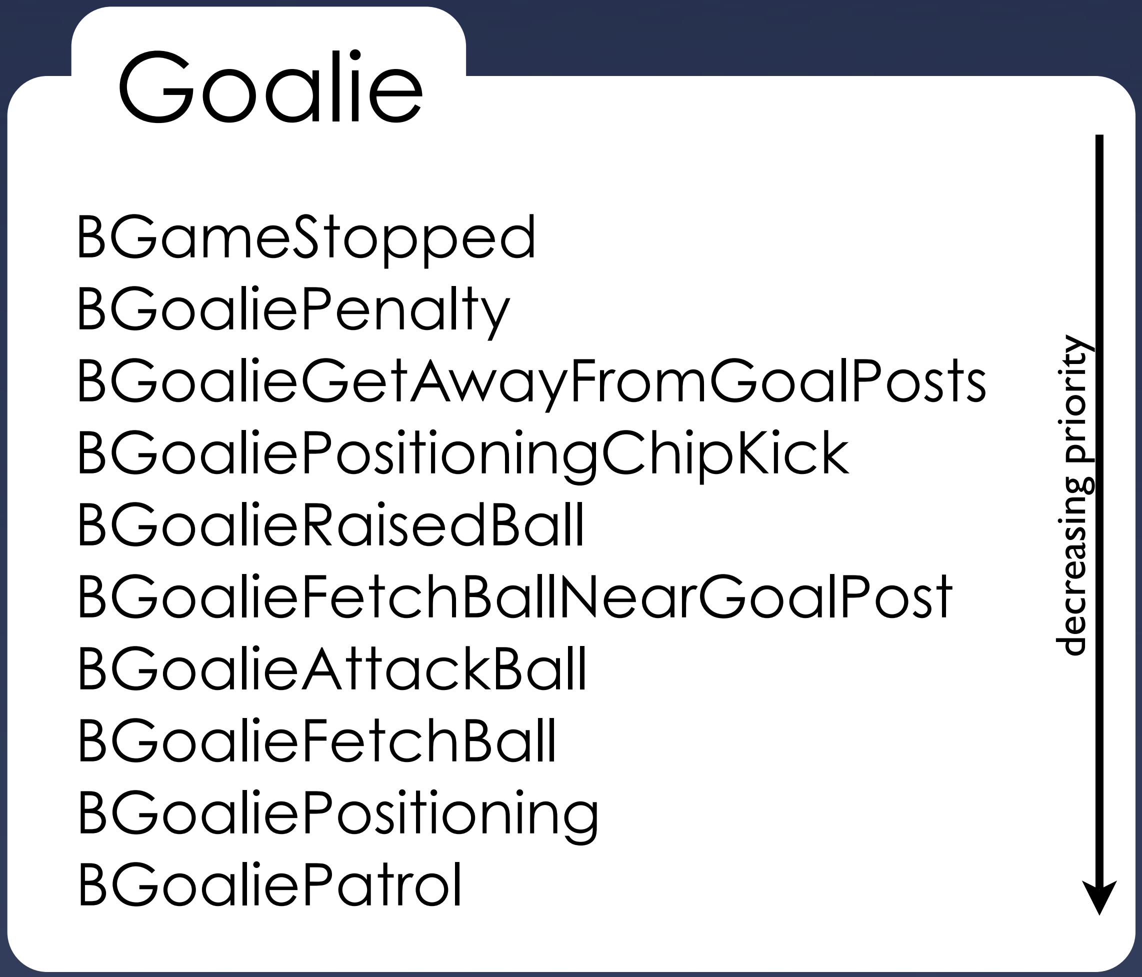
priority decreasing →

Types of Arbitration

```
Algorithm 2 The ”finish plan first” arbitration scheme.
Require: intention $\neq 0$
  if not intention.commitment_condition(t) then
    intention $\Leftarrow$ emergency_stop
    for $i = 0$ to options.length() do
      if options[$i$].invocation_condition(t) then
        intention $\Leftarrow$ options[$i$]
        break
      end if
    end for
  end if
Ensure: intention $\neq 0$
```



Example: Goalie



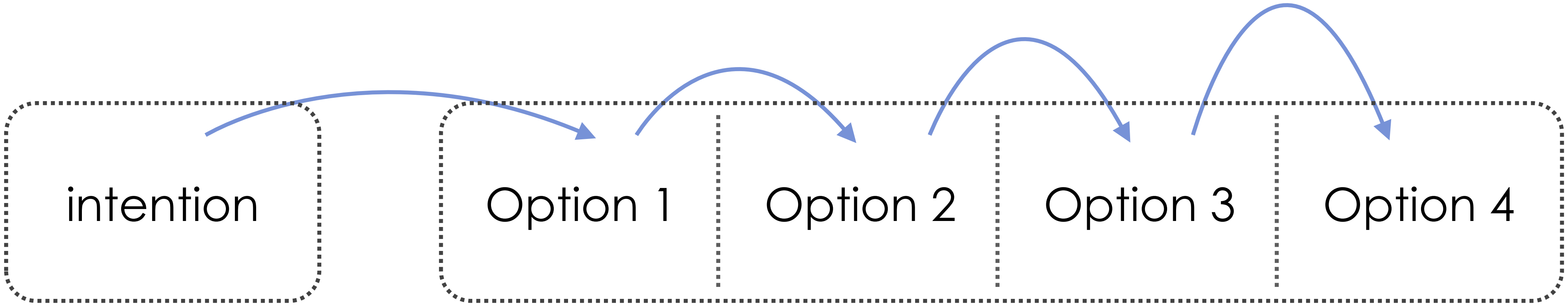
Goalie

BGameStopped
BGoaliePenalty
BGoalieGetAwayFromGoalPosts
BGoaliePositioningChipKick
BGoalieRaisedBall
BGoalieFetchBallNearGoalPost
BGoalieAttackBall
BGoalieFetchBall
BGoaliePositioning
BGoaliePatrol

- Goalie, plain list (highest priority first)

Types of Arbitration

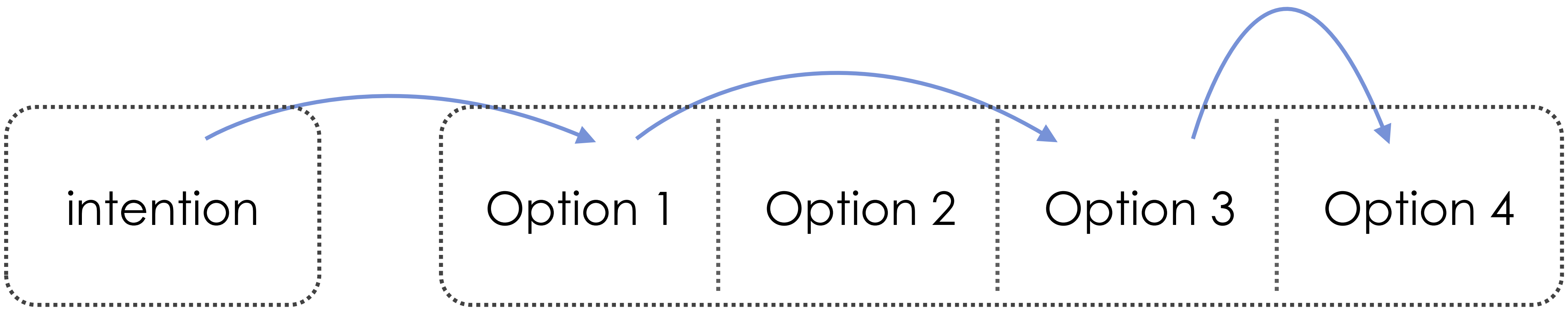
- Sequence (used for complex behaviors)



only check IC of next option in list

Types of Arbitration

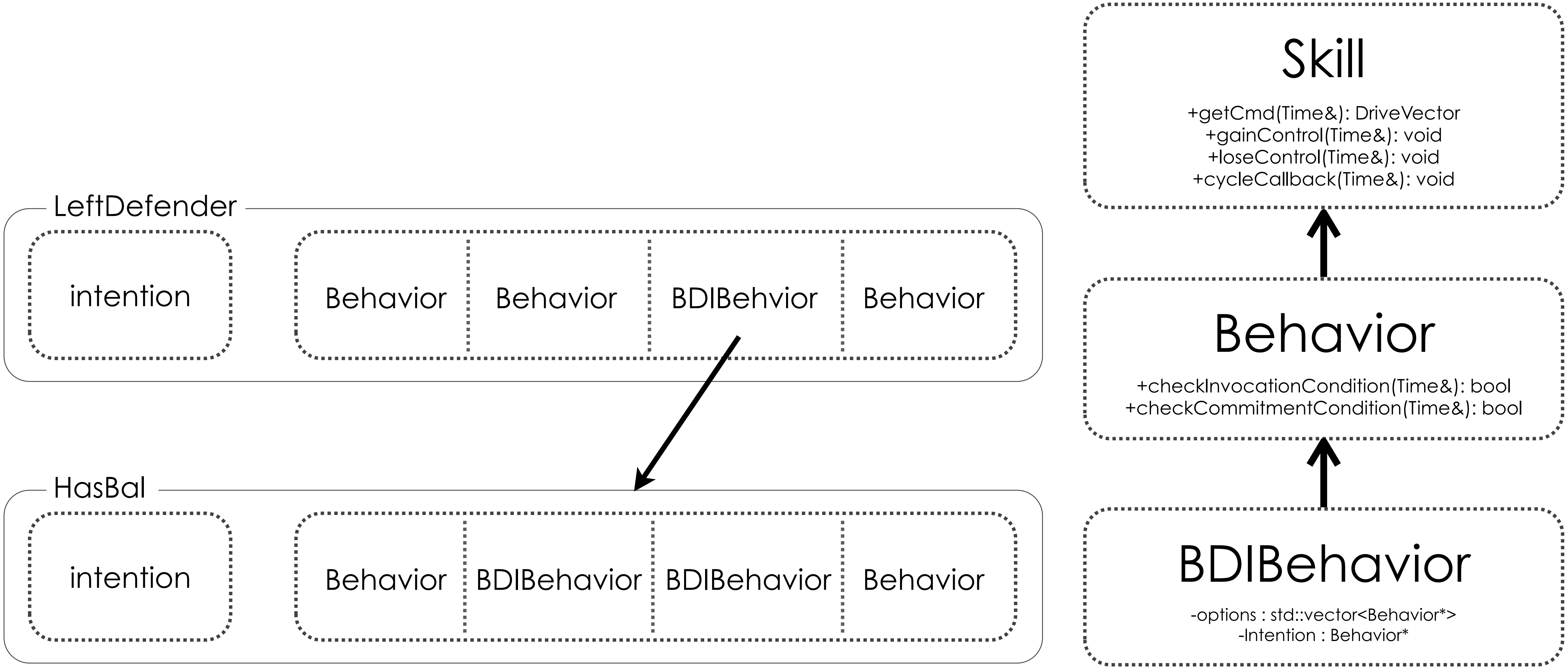
- Sequence (used for complex behaviors)
  - Generalized Sequence
    - Node has to be activated / can be skipped
    - Present node cedes control / subsequent node grabs control



Nesting

- Making this whole thing interesting: Nesting
- Arbitrators (BDIBehavior) are Behaviors themselves

➡ Behavior Hierarchy



LeftDefender

intention

Behavior | Behavior | BDIBehvior | Behavior

HasBal

intention

Behavior | BDIBehavior | BDIBehavior | Behavior

Skill

+getCmd(Time&): DriveVector
+gainControl(Time&): void
+loseControl(Time&): void
+cycleCallback(Time&): void

Behavior

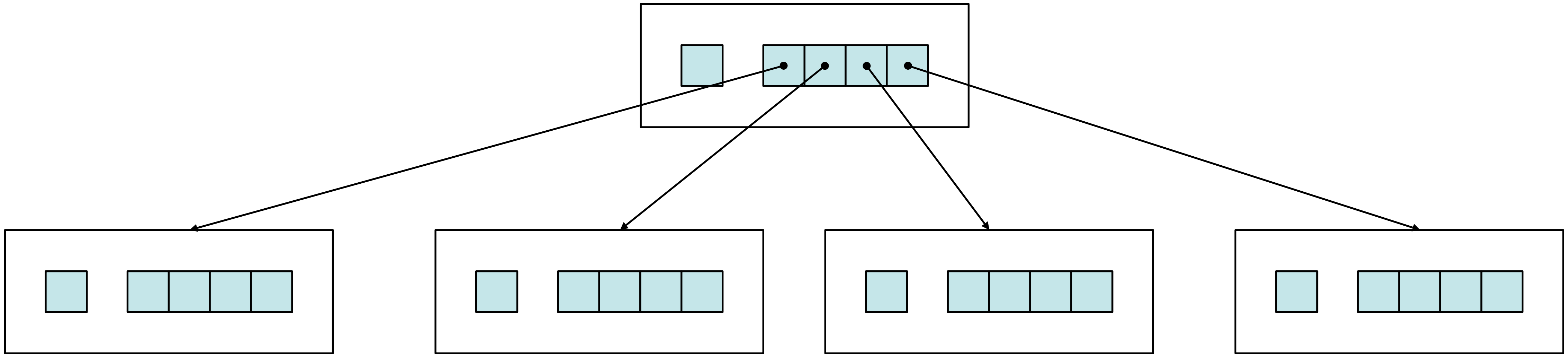
+checkInvocationCondition(Time&): bool
+checkCommitmentCondition(Time&): bool

BDIBehavior

-options : std::vector<Behavior*>
-Intention : Behavior*

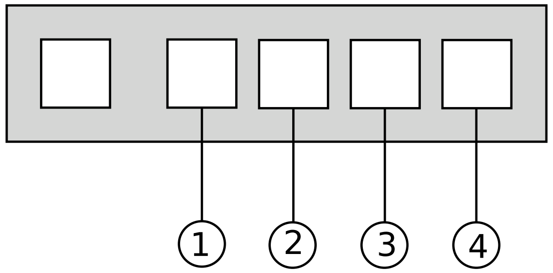
Example: Decision Tree

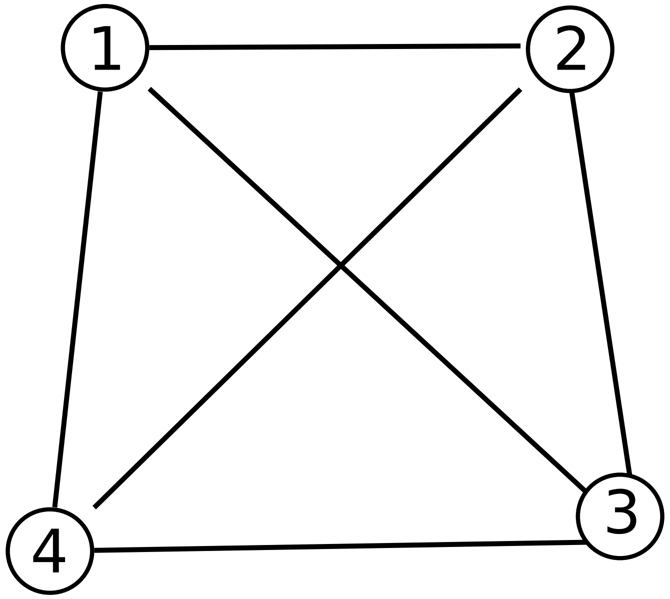
- Stack Arbitrators (binary, n-ary, whatever)
- Highest Priority first Arbitration
- nodes are arbitrators
- leaves are behaviors



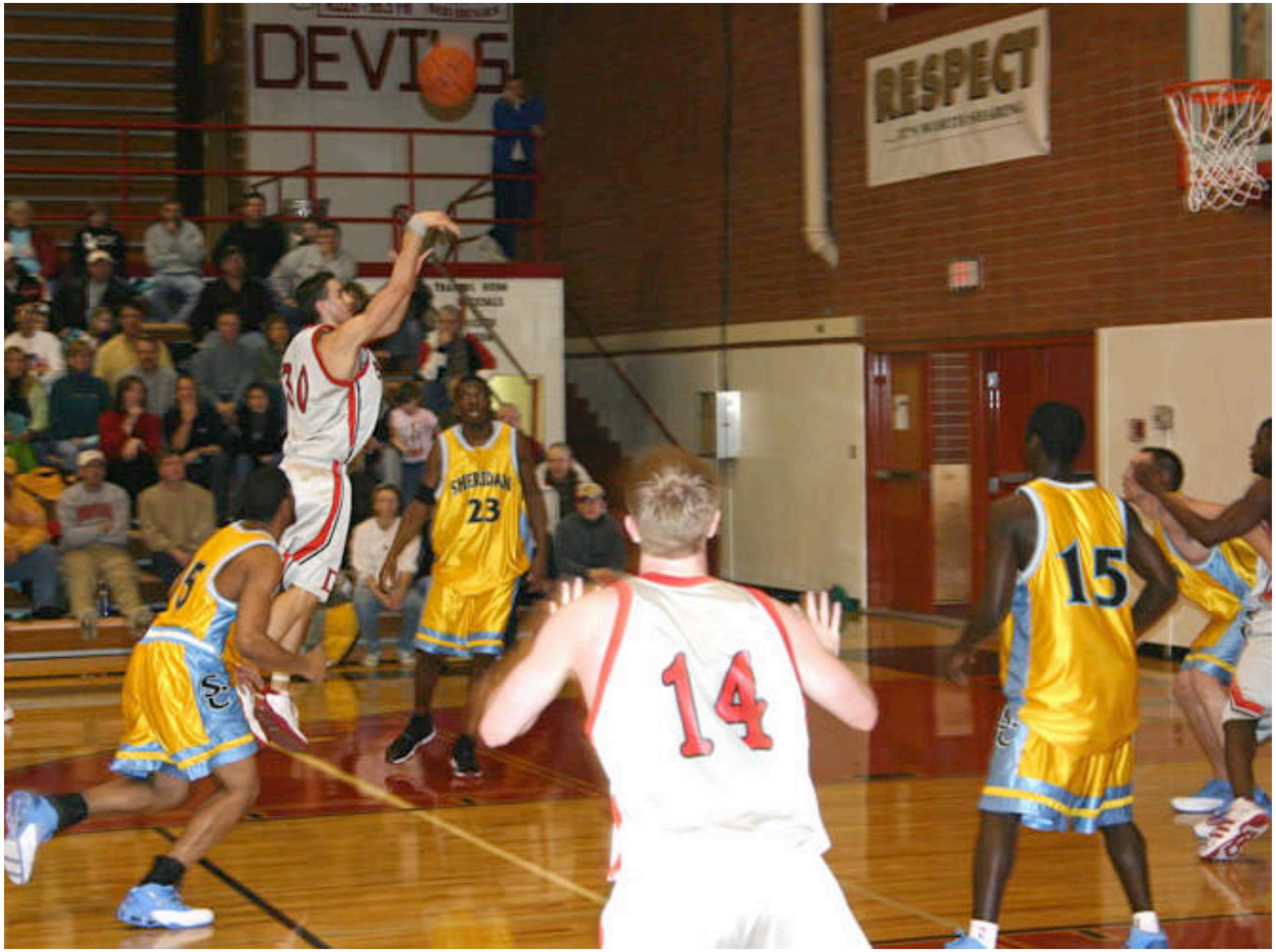
Analogy to Finite State Machines

- All our hierarchies can in principle be translated into an equivalent FSM
  - you would have to spread / C&P IC and CC among the transitions
- However it's a different way of thinking
  - we assume a situation
  - history is not important





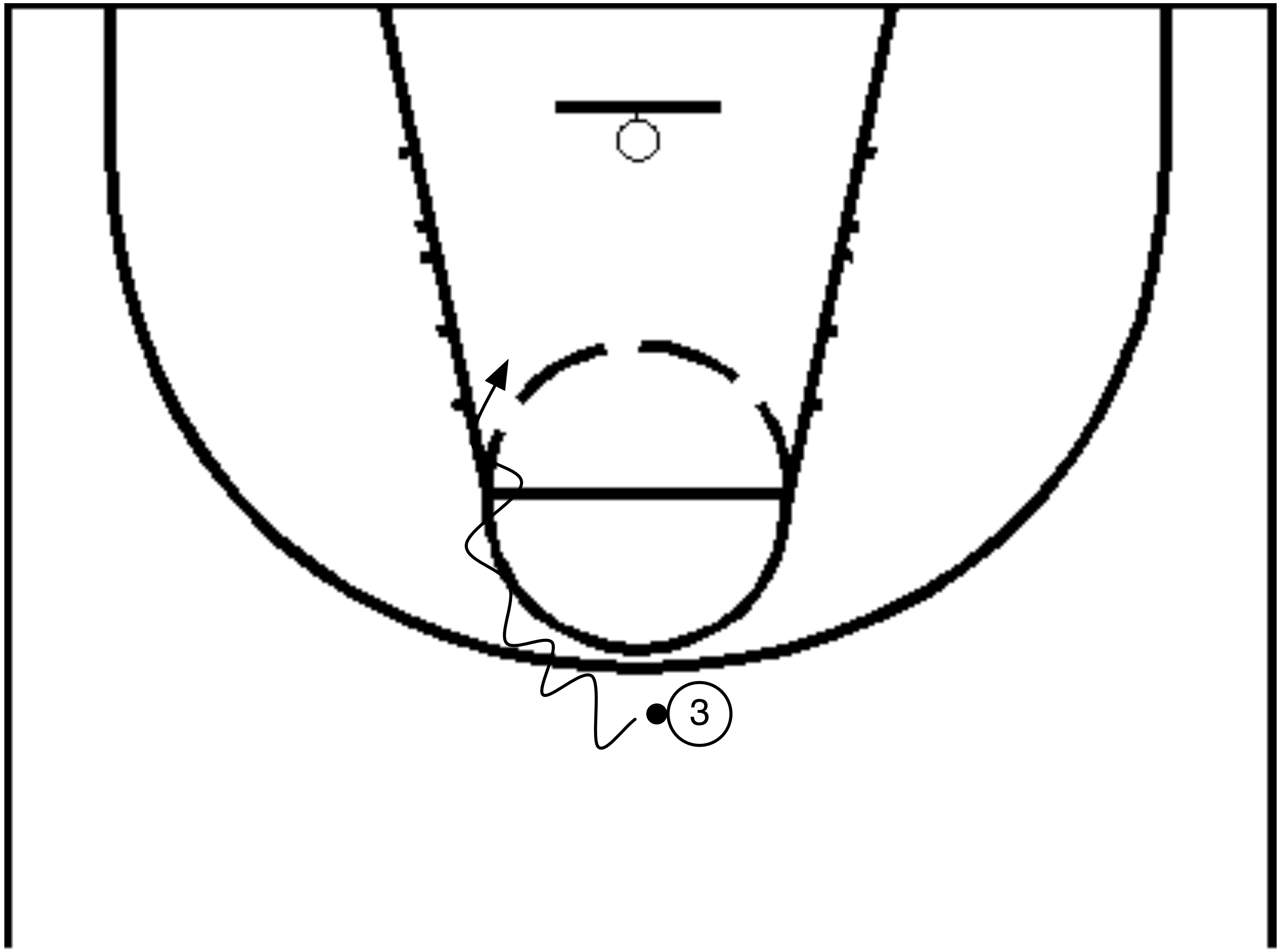
Think in „Situations“



In this situation, the only correct decision is to shoot the ball

(ok, obviously, I would have tried to dunk it, but trying to score is the right decision ;-)

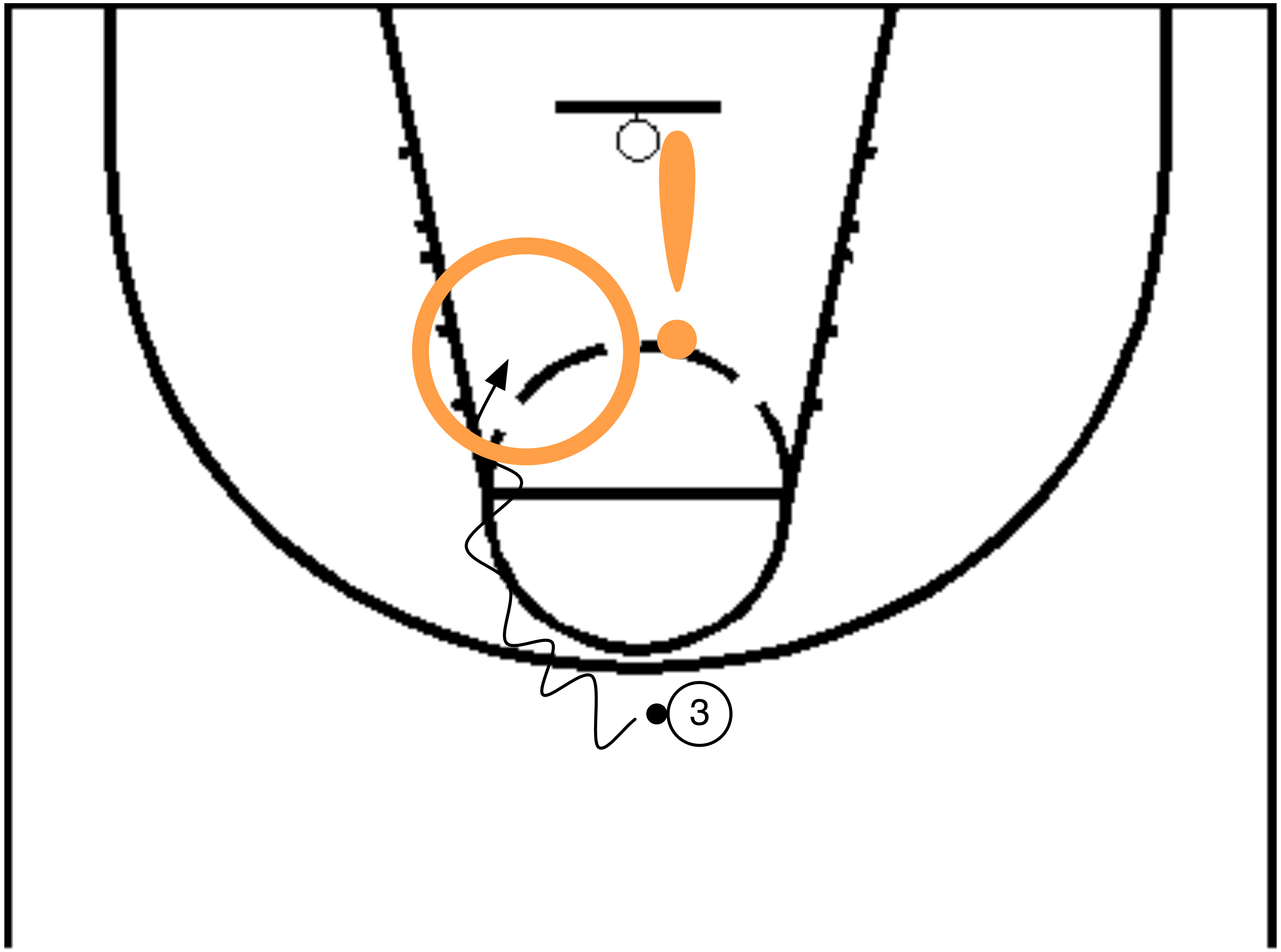
Think in „Situations“



How you got there, is not important.

Whether you just dribbled there, ...

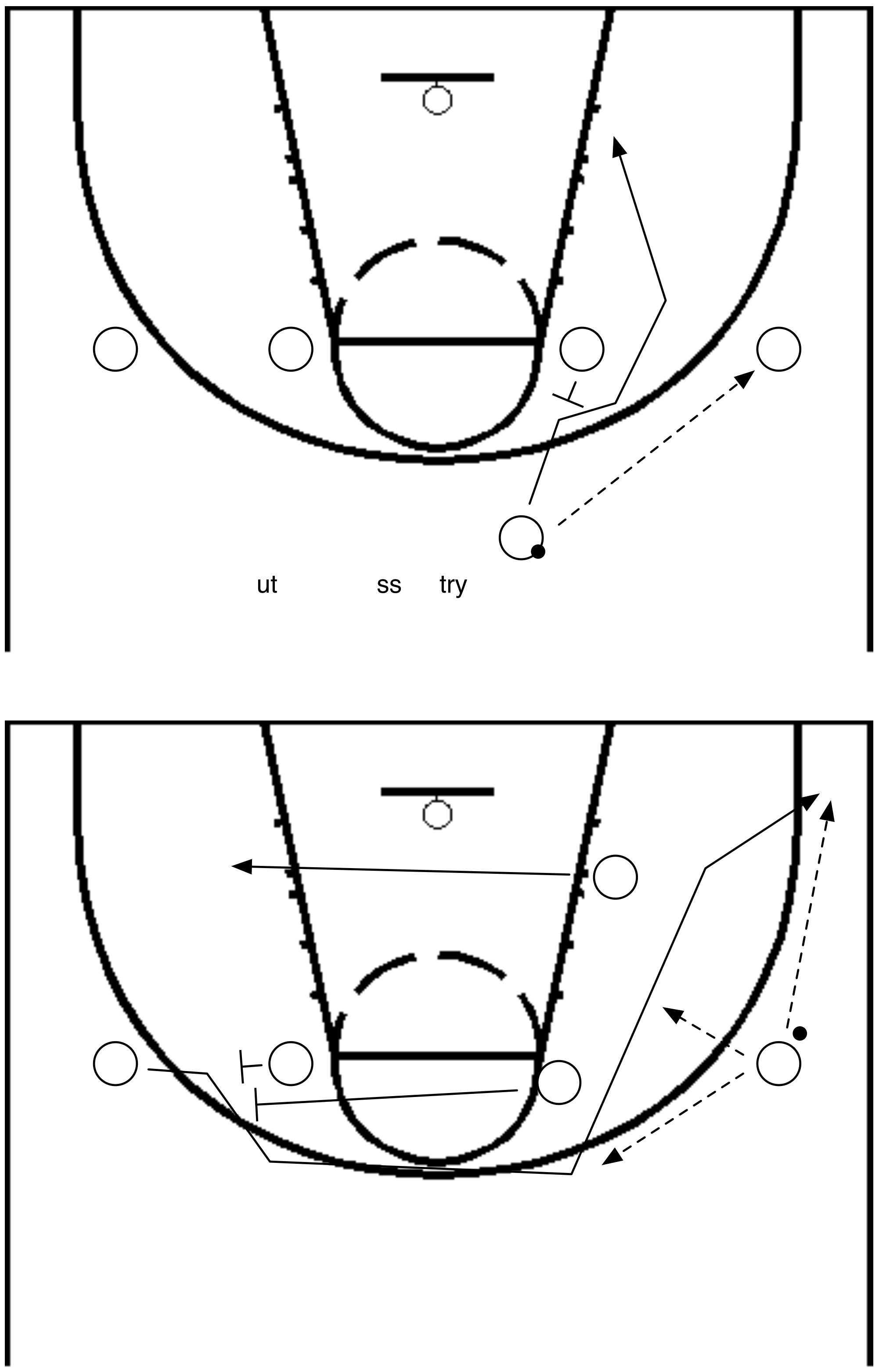
Think in „Situations“



How you got there, is not important.

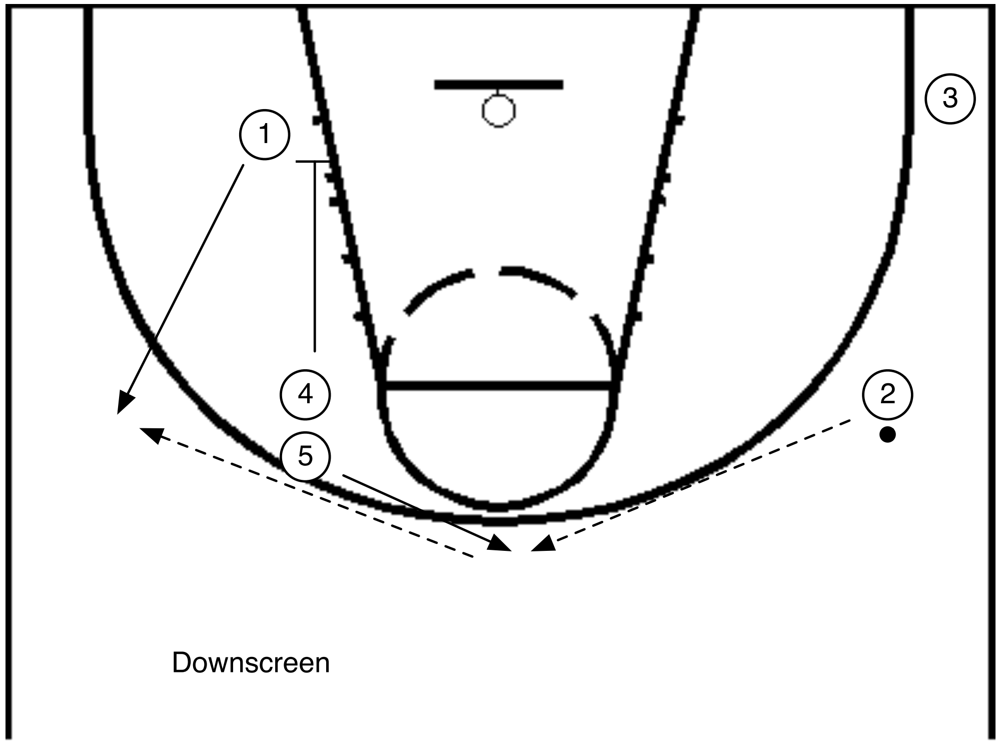
Whether you just dribbled there, ...

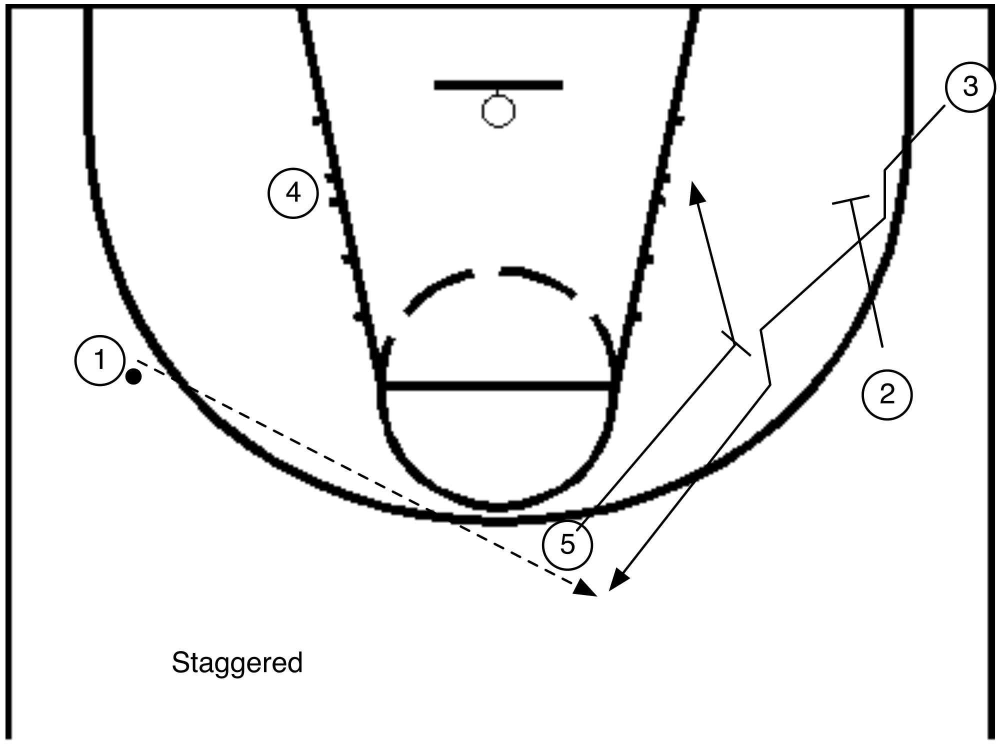
Think in „Situations“



... or you ran a really complex set play with several screens and passes...

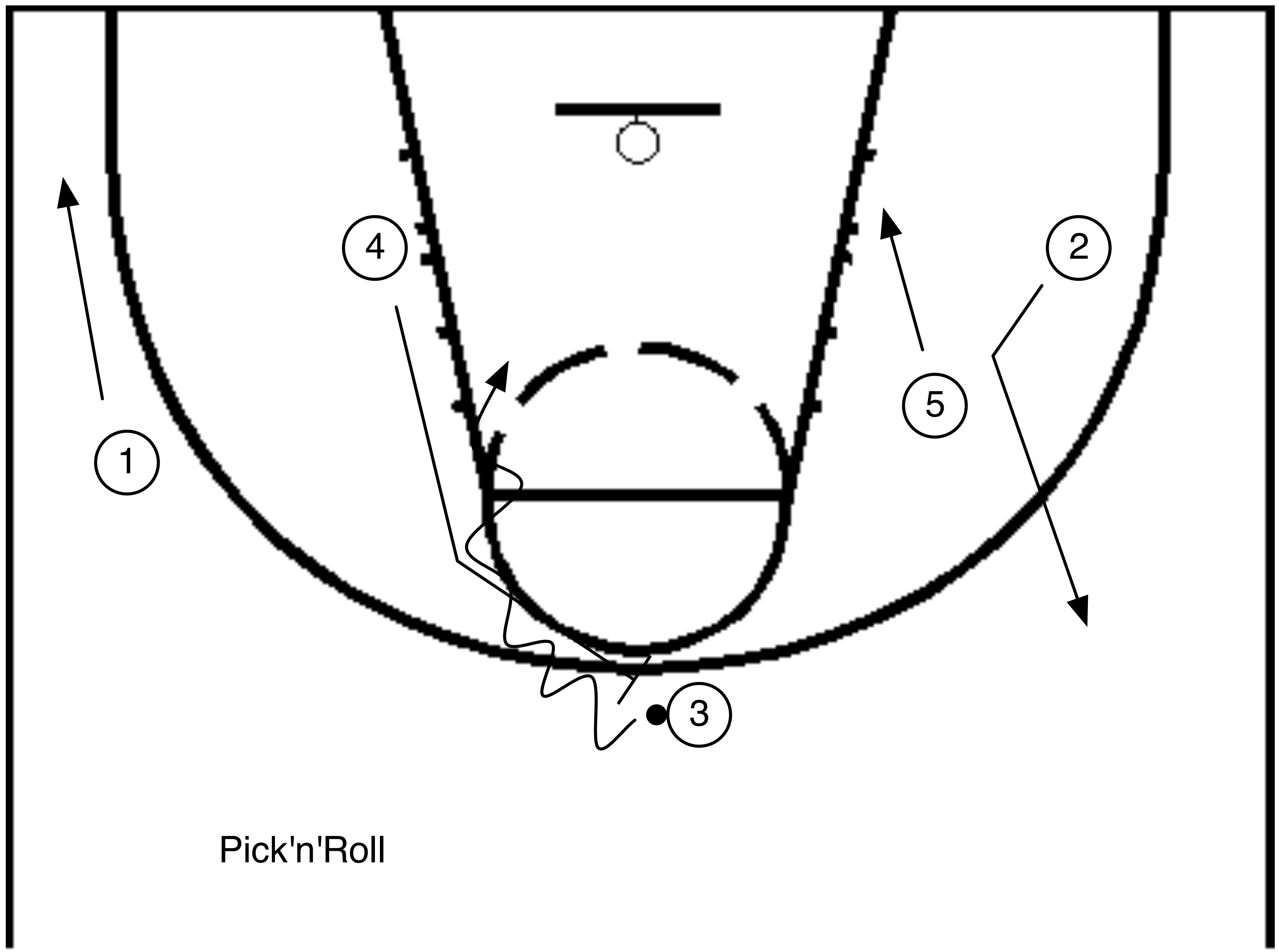
Think in „Situations“





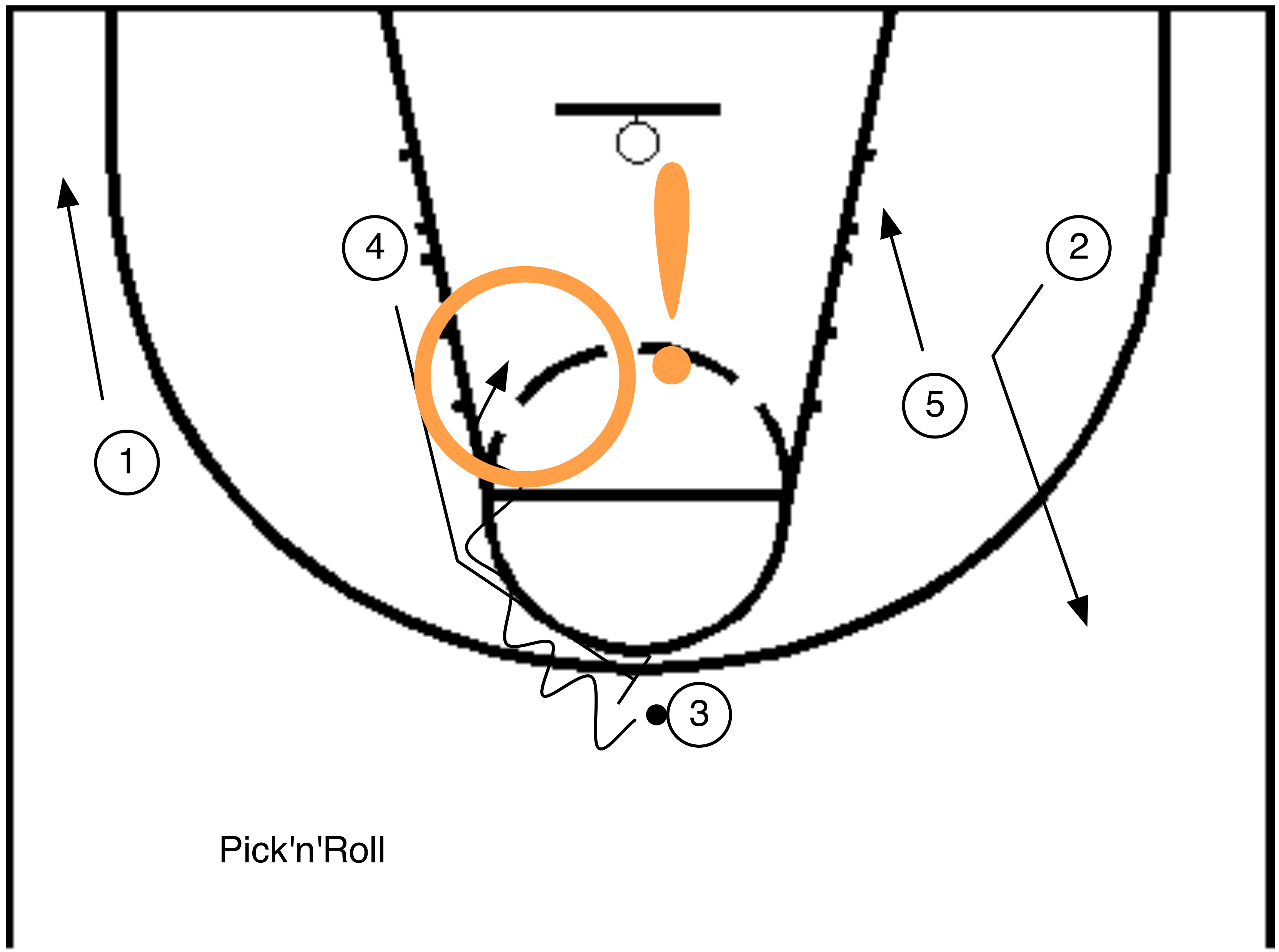
... or you ran a really complex set play with several screens and passes...

Think in „Situations“



... or you ran a really complex set play with several screens and passes...

Think in „Situations“



... or you ran a really complex set play with several screens and passes...

Think in „Situations“

... the correct decision will still be the same.

Shoot it.

Think in „Situations“

Basketball players try to keep decisions as simple as possible.

They train and find solutions for simplified „situations“ in „break-down drills“.

Think in „Situations“

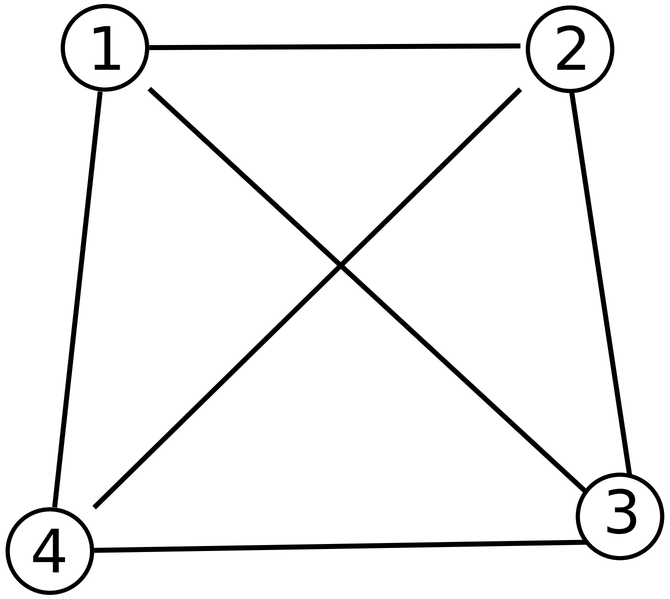
So don‘t think about the history or state transitions, just decide what‘s the best action in the present situation, as the smart players do ;-)

> it‘s an MDP ;-)

Analogy to Finite State Machines

Different way of thinking:

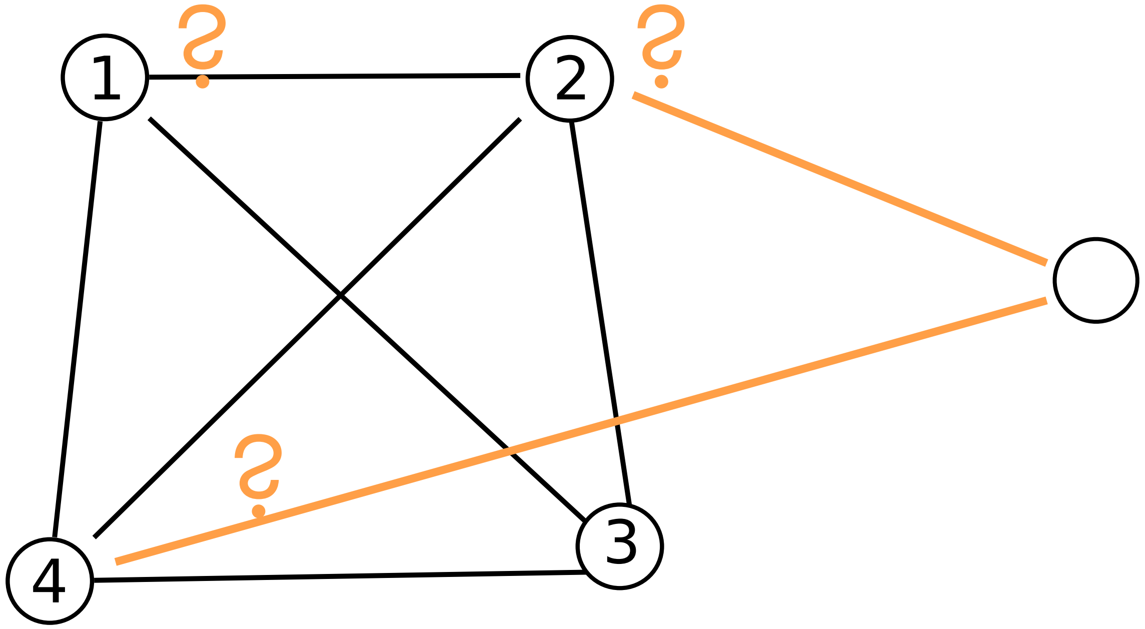
- no transitions are specified
- practical benefits:
  - simply insert and delete nodes
  - recurring transition conditions centrally formulated



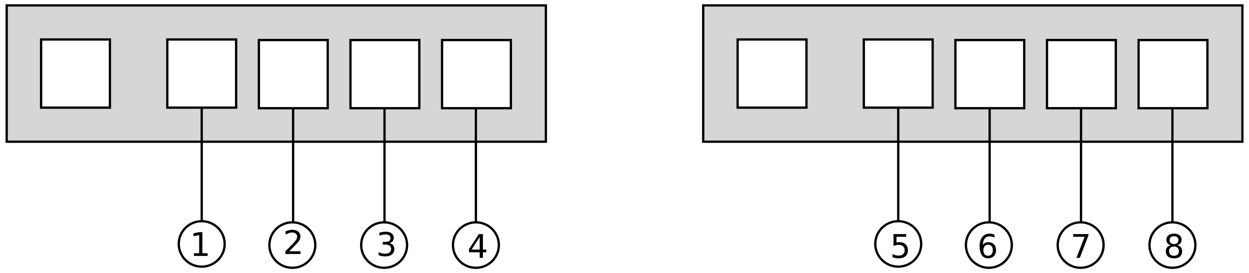
Analogy to Finite State Machines

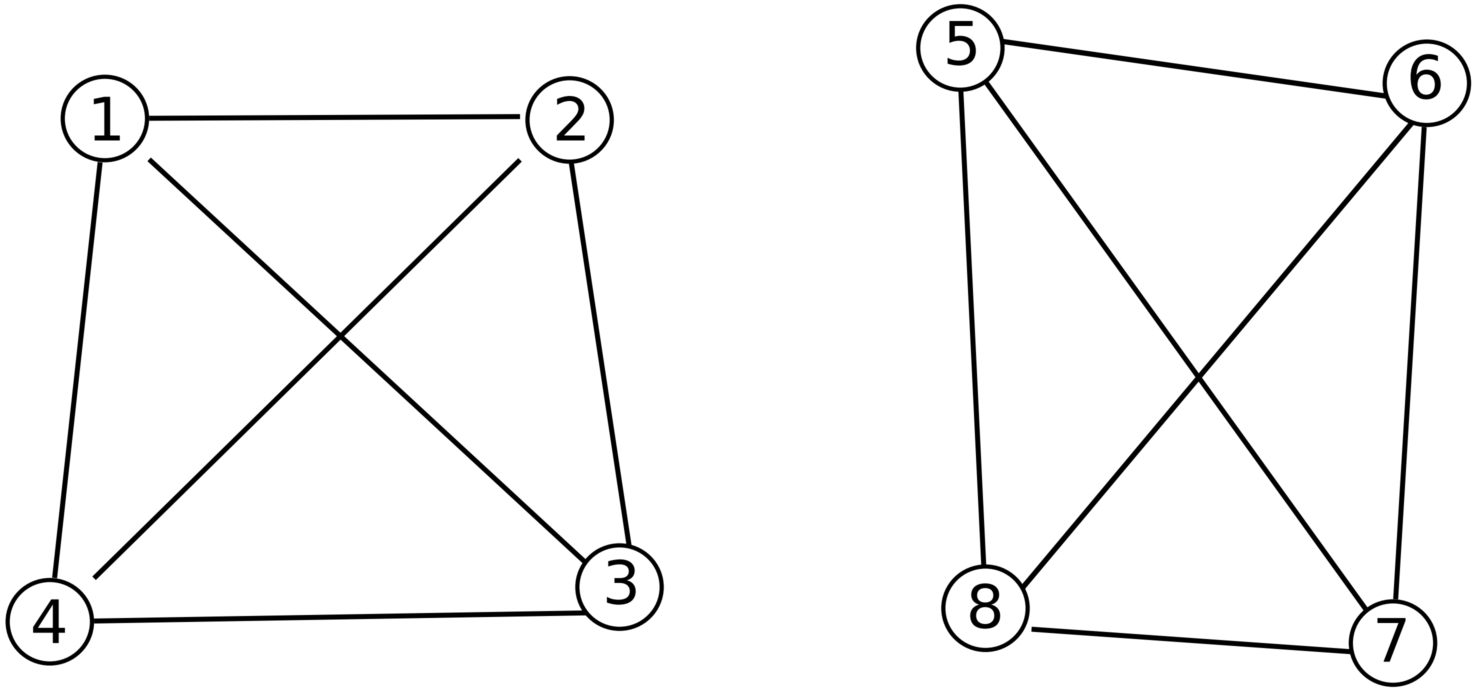
Different way of thinking:

- no transitions are specified
- practical benefits:
  - simply insert and delete nodes
  - recurring transition conditions centrally formulated

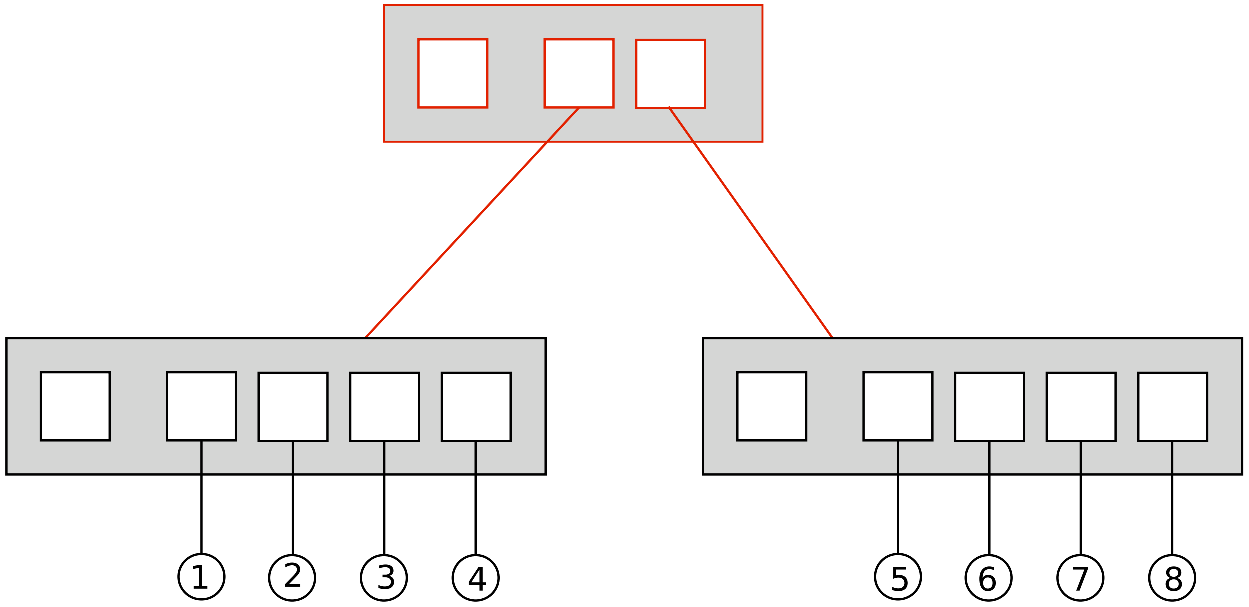


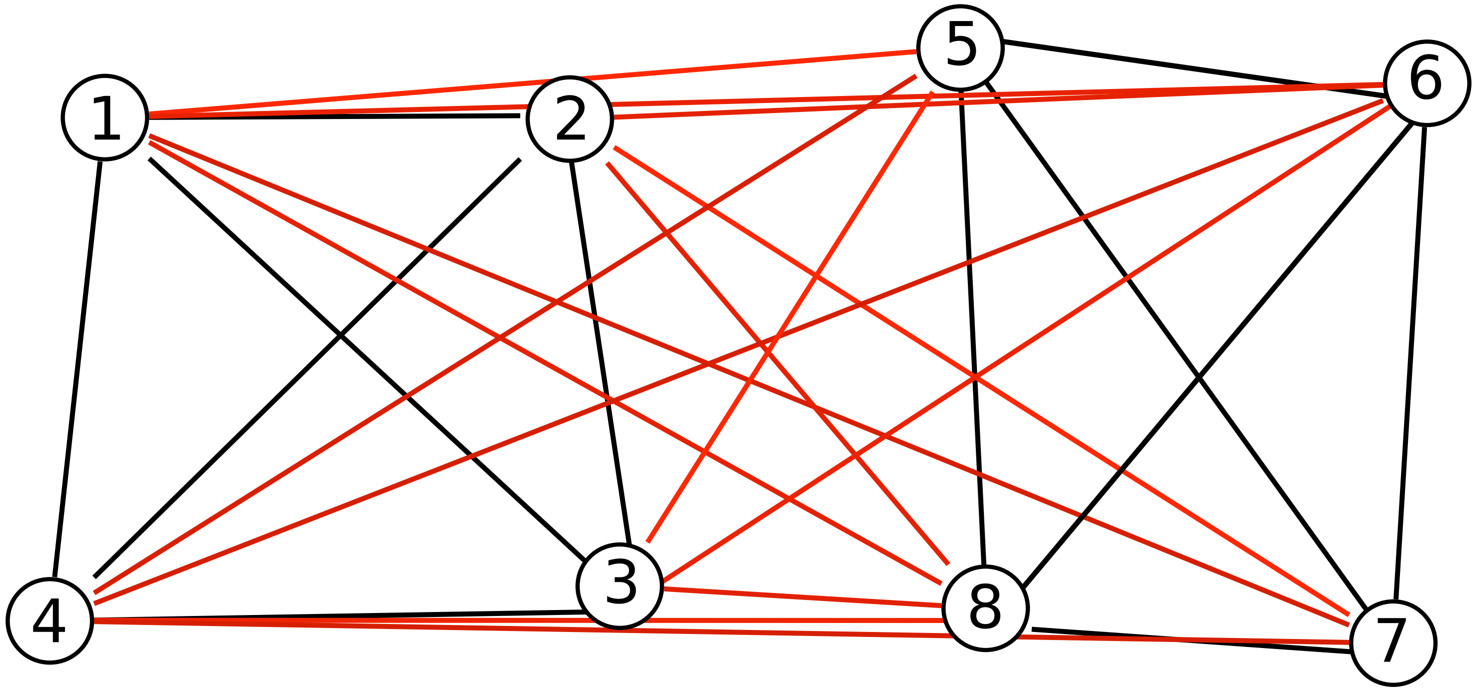
BDI Hierarchy / FSM



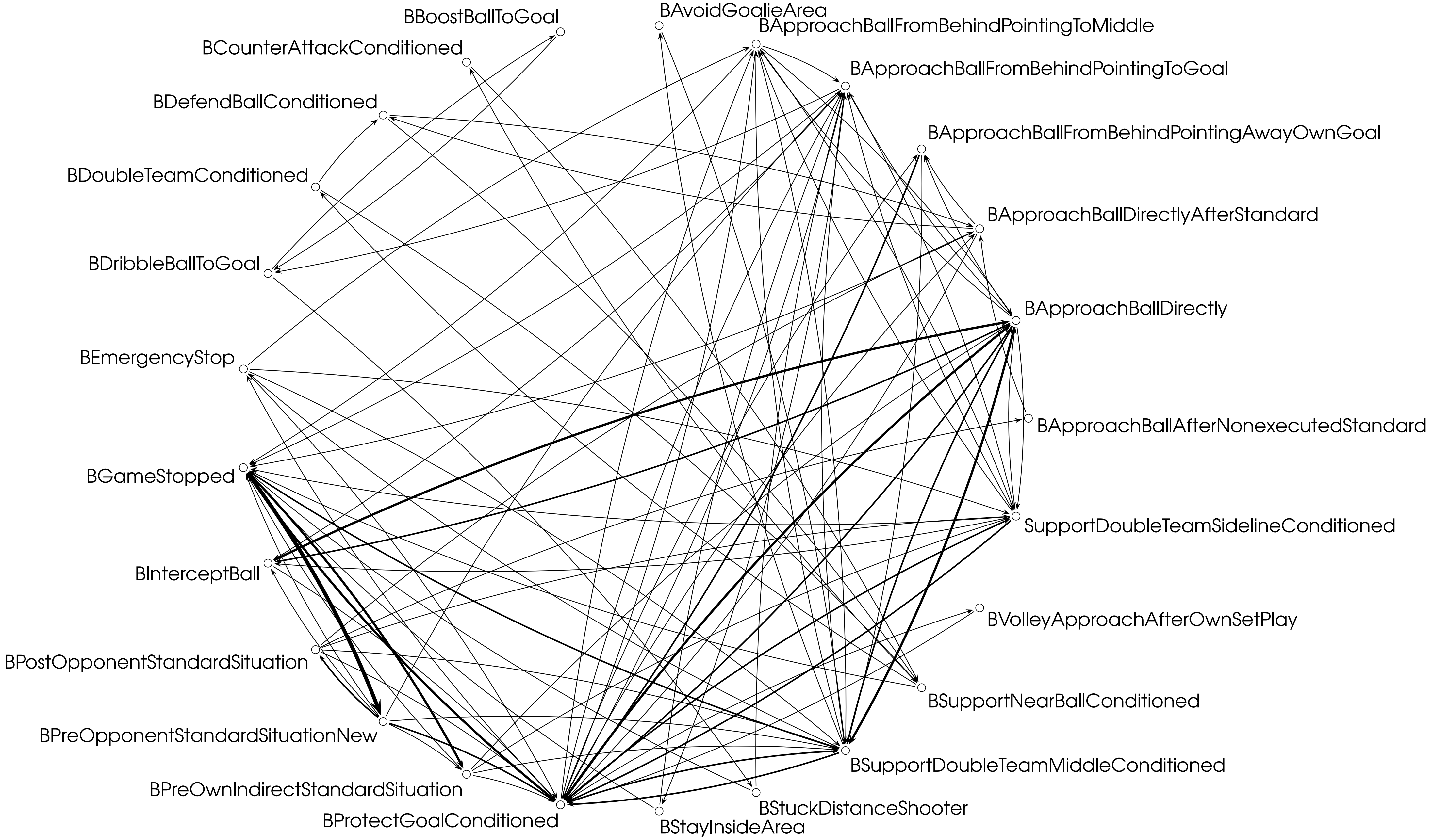


BDI Hierarchy / FSM





Transitions vs. Stuttgart


BBoostBallToGoal
BCounterAttackConditioned
BDefendBallConditioned
BDoubleTeamConditioned
BDribbleBallToGoal
BEmergencyStop
BGameStopped
BInterceptBall
BPostOpponentStandardSituation
BPreOpponentStandardSituationNew
BPreOwnIndirectStandardSituation
BProtectGoalConditioned
BStayInsideArea
BStuckDistanceShooter
BSupportDoubleTeamMiddleConditioned
BSupportNearBallConditioned
BVolleyApproachAfterOwnSetPlay
SupportDoubleTeamSidelineConditioned
BApproachBallAfterNonexecutedStandard
BApproachBallDirectly
BApproachBallDirectlyAfterStandard
BApproachBallFromBehindPointingAwayOwnGoal
BApproachBallFromBehindPointingToGoal
BApproachBallFromBehindPointingToMiddle
BAvoidGoalieArea

Conclusion

- Behavior based
- Generic arbitration schemes
- Behavior hierarchy using nested arbitrators
- Most behaviors are reactive
- Cut „situations“ from the state space instead of thinking in transition graphs
  - more easily separate individual behavior and plug in a RL training setup
  - more easily to integrate new behavior in an existing strategy