# SystemCoDesigner – An ESL Design Methodology Based on the FunState MoC

C. Haubelt, J. Falk, C. Zebelein, J. Keinert, and J. Teich

University of Erlangen-Nuremberg
Germany

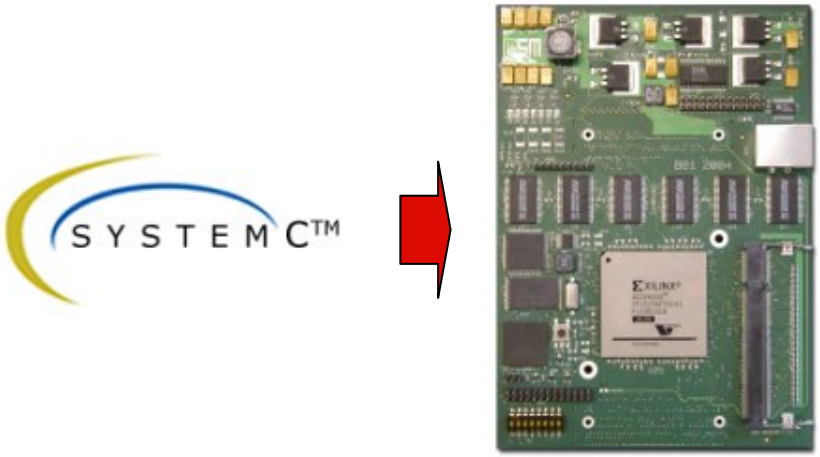
haubelt@cs.fau.de

Shuvra S. Bhattacharyya

University of Maryland
MD

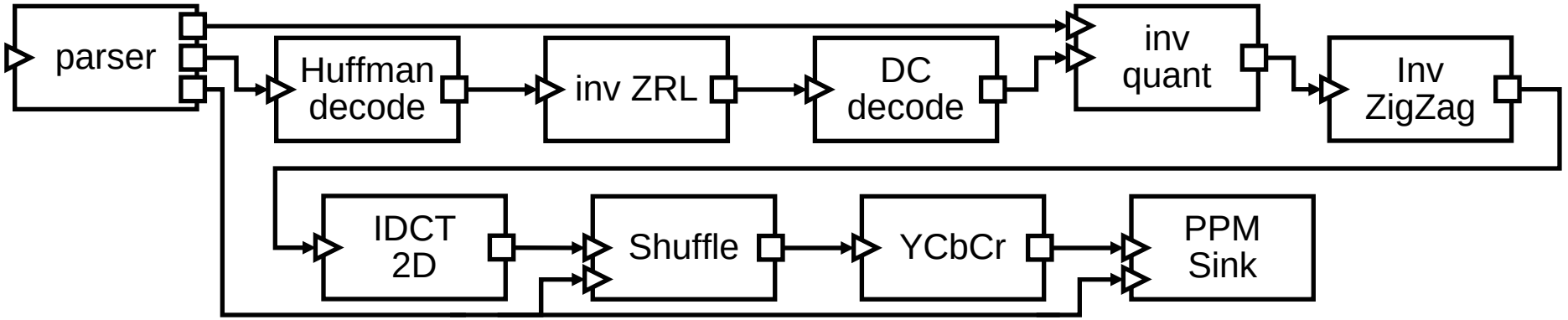
ssb@umd.edu

# SystemCoDesigner Goal



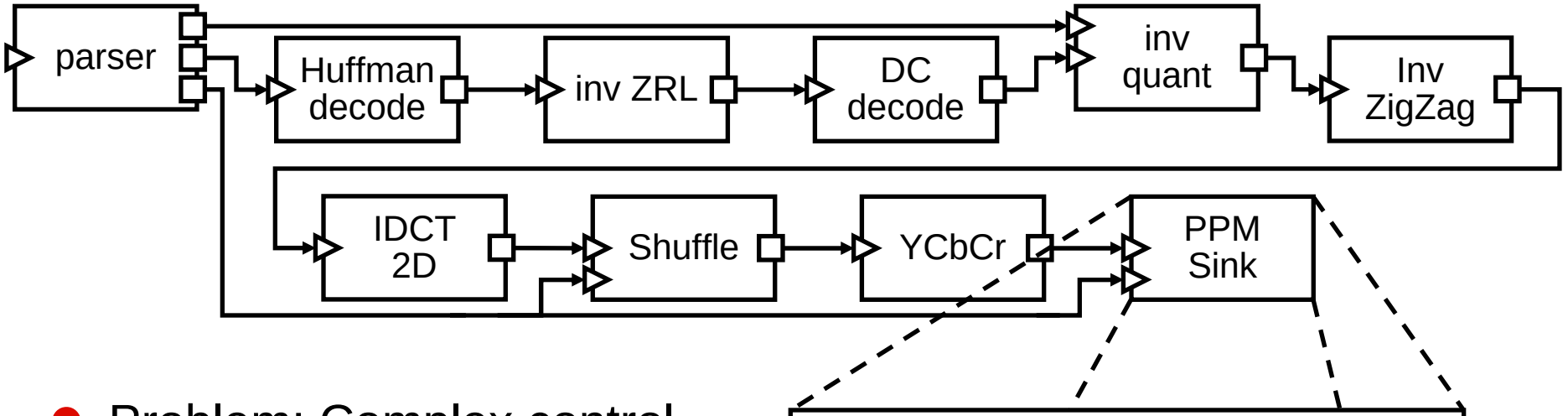

- Challenges:
  - How to optimize the implementation?
  - Early design decisions have a huge impact on the design quality

# SystemC Advantages




- Actor-oriented design
  - Network graph
  - Channels
  - Actors
- Actors are only allowed to communicate via channels
  - Point-to-point connections
  - Queues with FIFO semantics

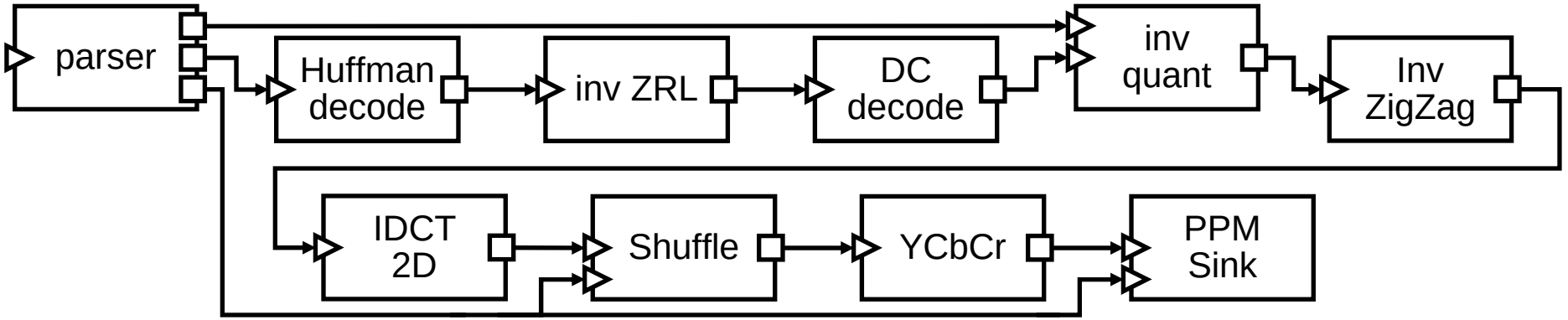
# SystemC Disadvantages




- Problem: Complex control flow and unstructured channel accesses prevent optimal implementations of the application

```
class PPMSink: public sc_module {
  void process() {
    while(1) {
      dimX = i2.read();
      dimY = i2.read();
      pixels = dimX*dimY;
      printHeader();
      for(n = 0; n < pixels; n++) {
        p = i1.read();
        printPixel(p);
      }
    }
  }
};
```

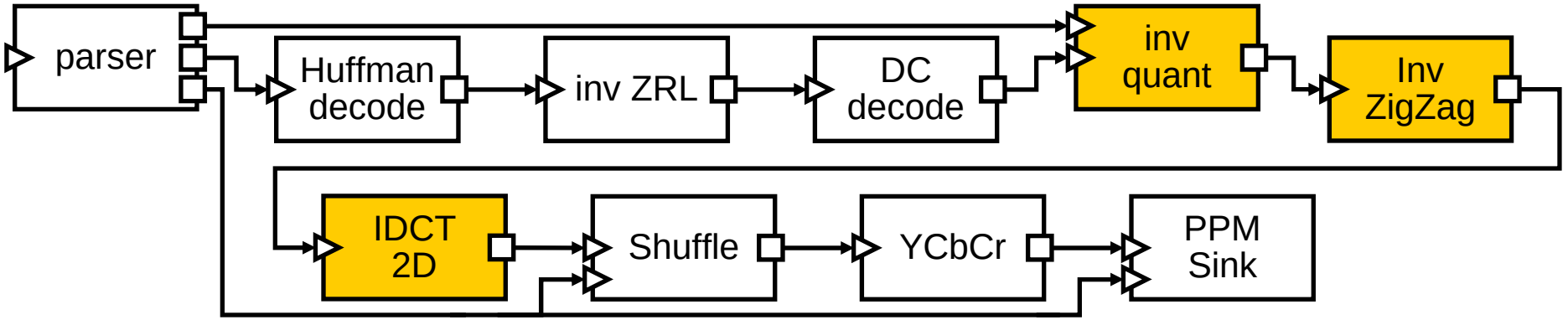
# Single Processor Scheduling




```
schedule() {
  while(true) {
    forall actors a ∈ A {
      if is_executable(a) {
        execute(a)
      }
    }
  }
}
```

Potential scheduling overhead

# Improvement




```
schedule() {
  while(true) {
    if is_executable(inv. quant) {
      execute(inv. quant)
      execute(inv. ZigZag)
      execute(IDCT2D)
    }
    forall actors a ∈ A_dyn {
      if is_executable(a) {
        execute(a)
      }
    }
  }
}
```

# Problem Formulation

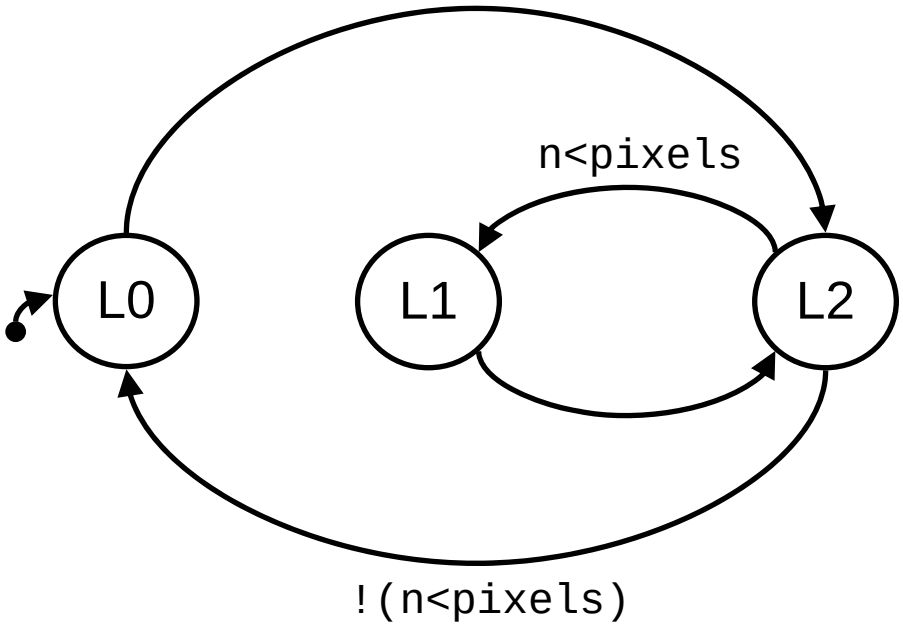
- How to identify restricted models of computation?
- How to perform (multi-processor) scheduling of static subgraphs?
- This work restricts the MoCs which can be recognized to two important static data flow MoCs:
  - SDF (synchronous data flow)
  - CSDF (cyclo-static data flow)

# Agenda

- Motivation
- The SystemCoDesigner modeling approach
- Automatic actor classification
- Clustering of static dataflow actors
- Results
- Conclusion

# Communication ⇔ Functionality

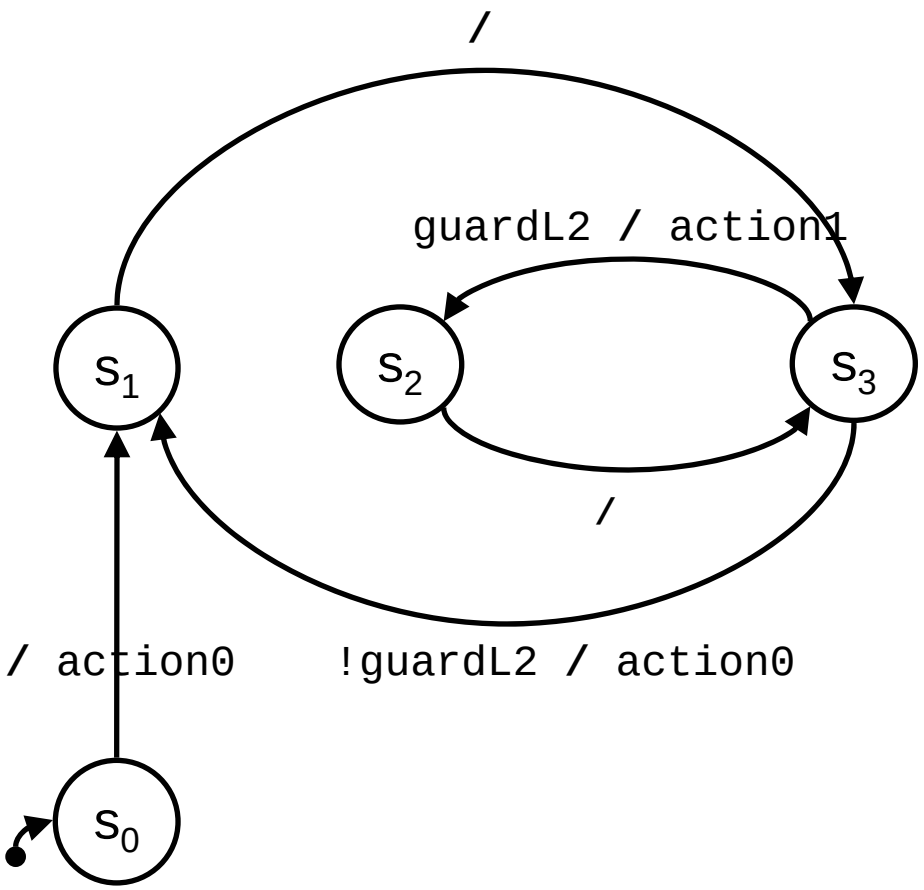
```
class PPMSink: public
sc_module {
  void process() {
  L0:
    dimX = i2.read();
    dimY = i2.read();
    pixels = dimX*dimY;
    printHeader();
    n = 0;
    goto L2;
  L1:
    p = i1.read();
    printPixel(p);
    ++n;
  L2:
    if(n < pixels) goto L1;
    else goto L0;
  }
};
```




- Generate **CFG** from SystemC thread method

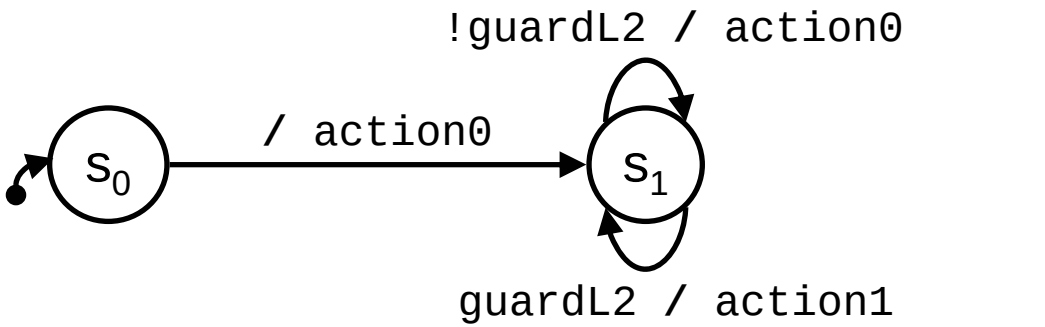
# Communication ⇔ Functionality

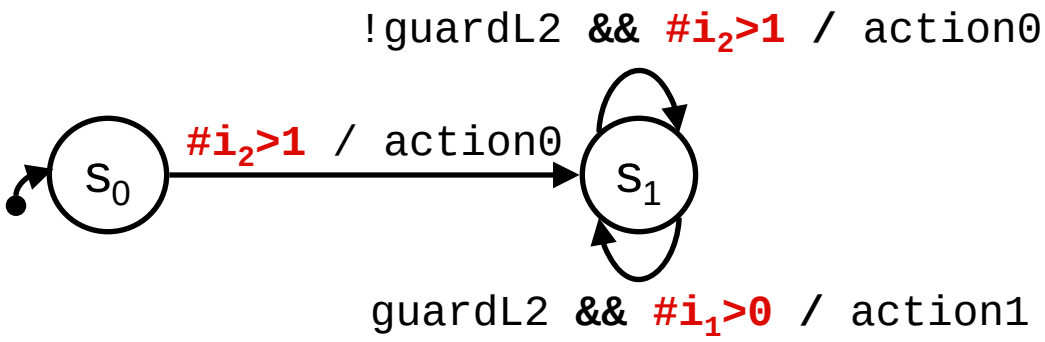
```
class PPMSink: public sc_module {
  void action0() {
    dimX = i2.read();
    dimY = i2.read();
    pixels = dimX*dimY;
    printHeader();
    n = 0;
  }
  void action1() {
    p = i1.read();
    printPixel(p);
    ++n;
  }
  bool guardL2() const {
    return n < pixels;
  }
  void process() {
  L0: action0(); goto L2;
  L1: action1();
  L2: if(guardL2()) goto L1;
      else goto L0;
  }
};
```




- Separate block code into **actions** (incoming transitions) and jump conditions into **guards** (outgoing transitions)

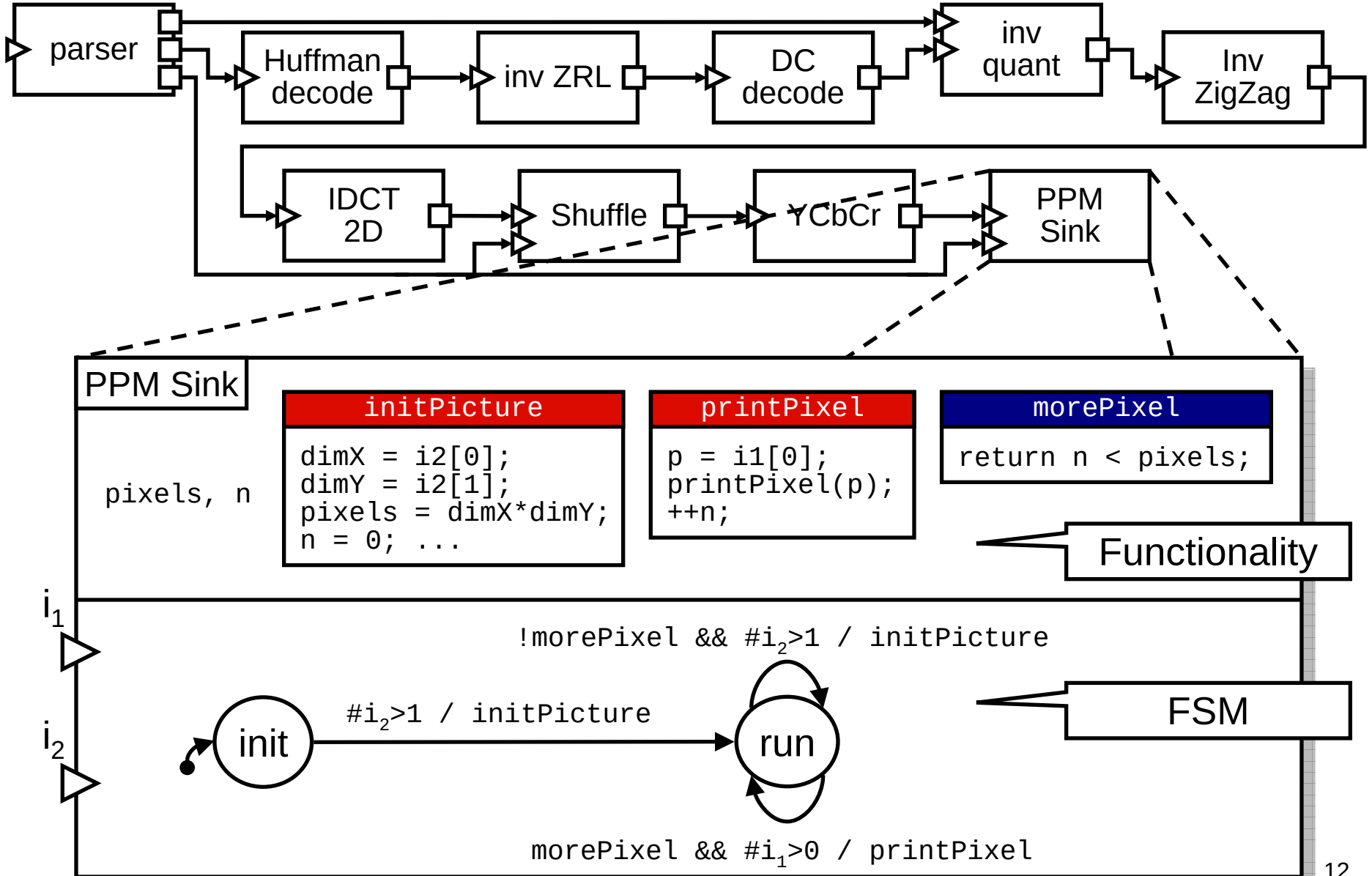
# Communication ⇔ Functionality




- Eliminate epsilon-transitions
- Add input/output requirements as guards to transitions

# FunState MoC



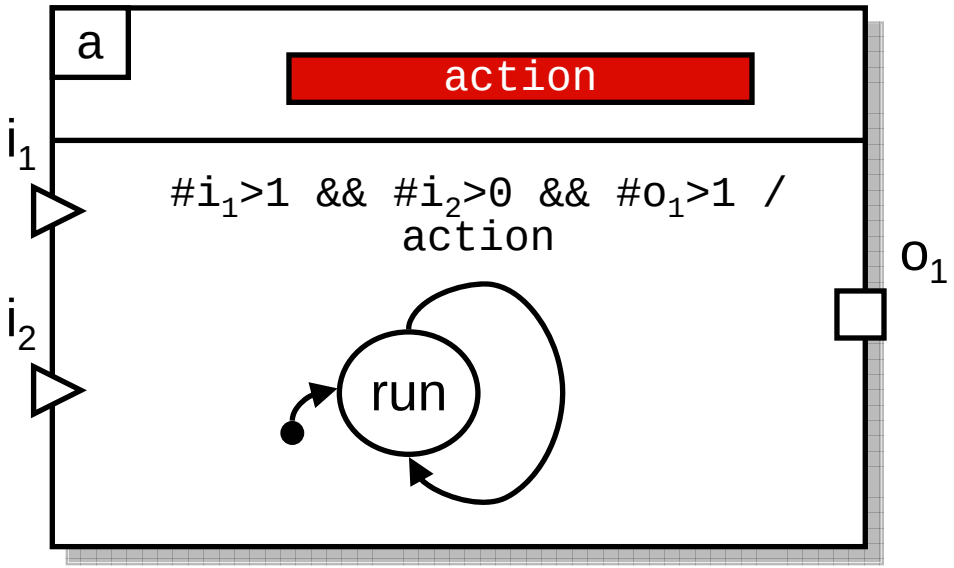
# Agenda

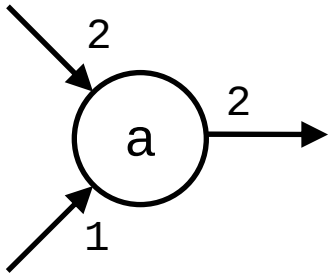
- Motivation
- The SystemCoDesigner modeling approach
- Automatic actor classification
- Clustering of static dataflow actors
- Results
- Conclusion

# Idea

- Classify actors by
  - minimizing the communication FSM
  - pattern matching
- Note that actor classification in its general form is undecidable
- This work restricts the MoCs which can be recognized to two important static data flow MoCs:
  - SDF (synchronous data flow)
  - CSDF (cyclo-static data flow)

# Classification



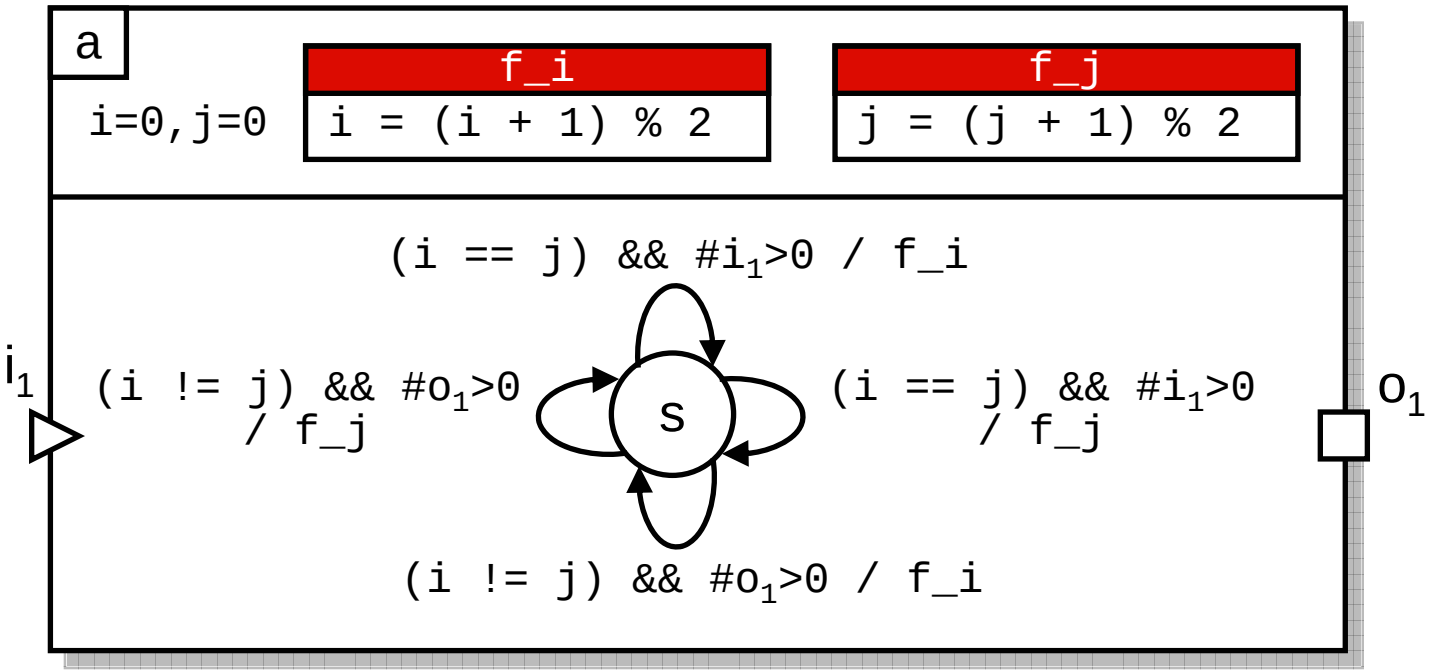




- Most basic representation of an actor which can be classified into the SDF MoC (left picture)
  - One state with one outgoing transition
- In this case, the communication behavior of the actor can be representend as commonly known from SDF graphs (right picture)
  - Edge e is annotated with cns(e) and prd(e)

# Actor state

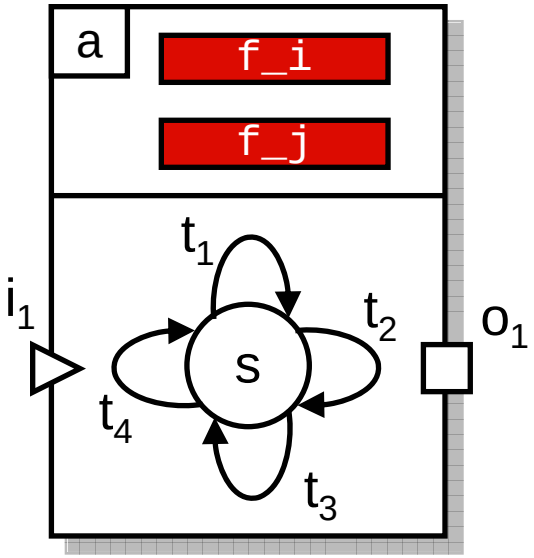
- Problem: FSM describes the possible communication behavior of the actor
  - However, only a subset of transitions may be enabled during the execution of the actor (→ Guards)
- To accomodate both FSM state and functionality state of an actor, we define an actor state as the pair $s = (s_{func}, s_{fsm})$
  - $S_{func}$ is the (possibly infinite) set of functionality states
  - $S_{fsm}$ is the finite set of FSM states
- When an enabled transition is executed, the next actor state s‘ is defined as follows:
  - The next FSM state $s'.s_{fsm}$ is destination state of the transition
  - The next functionality state $s'.s_{func}$ is the new functionality state after the execution of the associated action

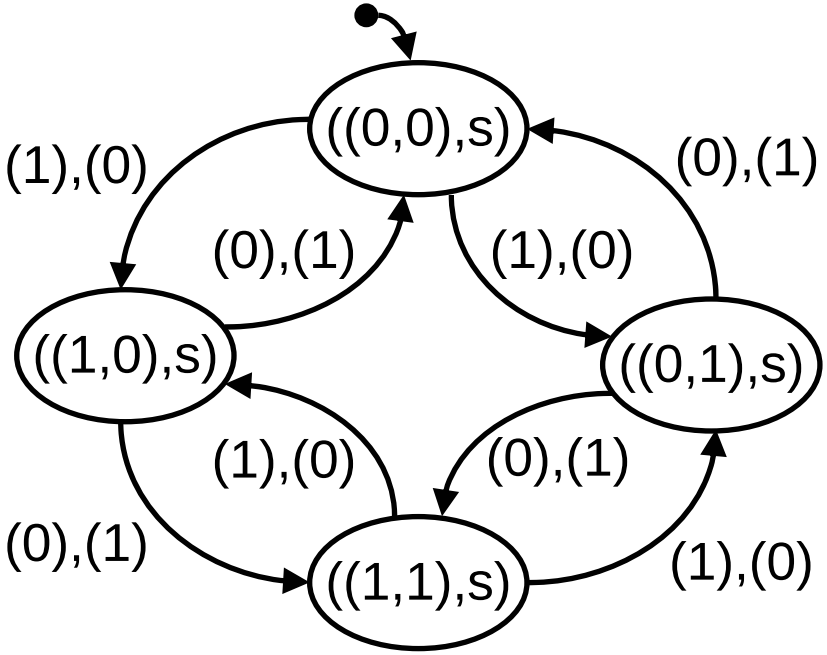
# Dynamic state transition diagram




- $S_{fsm} = \{ s \}$
- $S_{func} = \{ (0,0), (1,0), (0,1), (1,1) \}$
- The actor state space $S_{func} \times S_{fsm}$ induces a diagram which refines the possible communication behavior described by the FSM

# Dynamic state transition diagram



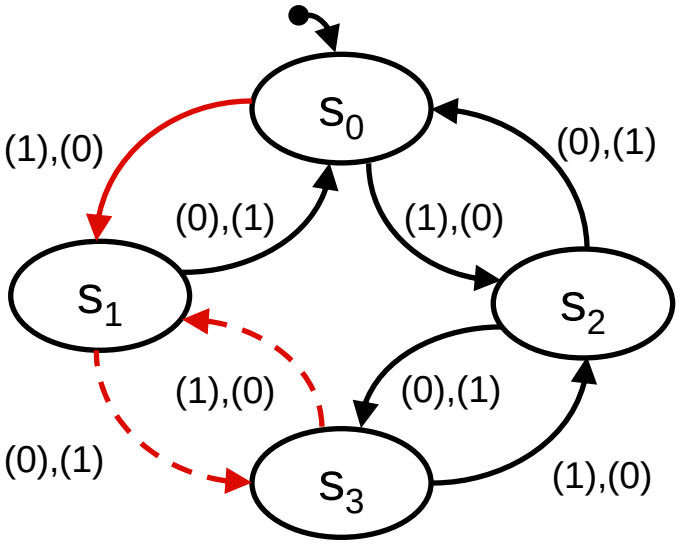



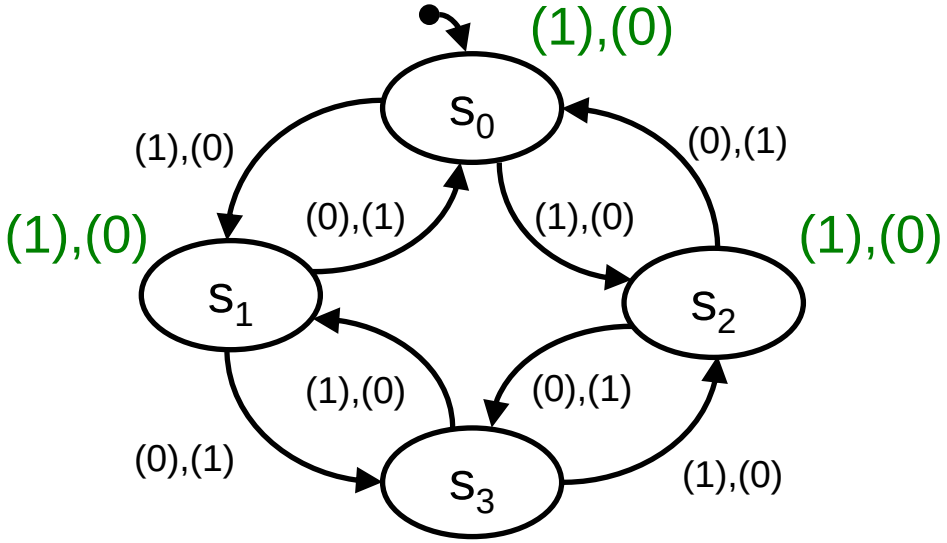

- For each actor state s = ($s_{func}$,$s_{fsm}$) and each enabled transition t:
  - Add edge e to dynamic state transition diagram connecting s with s‘
  - Annotate e with the number of tokens consumed and produced by t (as vectors e.**cns** = (t.cns($i_1$),t.cns($i_2$),…) and e.**prd** = (t.prd($o_1$),t.prd($o_2$),…))

# Classification algorithm

- Classification requirements:
  - Liveliness, i.e., each actor state must have at least one enabled outgoing transition (SDF / CSDF actors can always be activated if enough tokens available)
  - Active input / output behavior, i.e., no infinite sequence of edges which do not consume or produce any tokens
- If requirements are not met, deadlocks could be introduced by treating the actor as an SDF or CSDF actor
- Assume we have a hypothesis c („classification candidate“)
  - c.$\tau$: Number of phases
  - c.**cns**: Maps phase to vector of consumed tokens
  - c.**prd**: Maps phase to vector of produced tokens
- Accept / Reject c by annotating s.$\rho$, s.**cns**, s.**prd** to a visited state

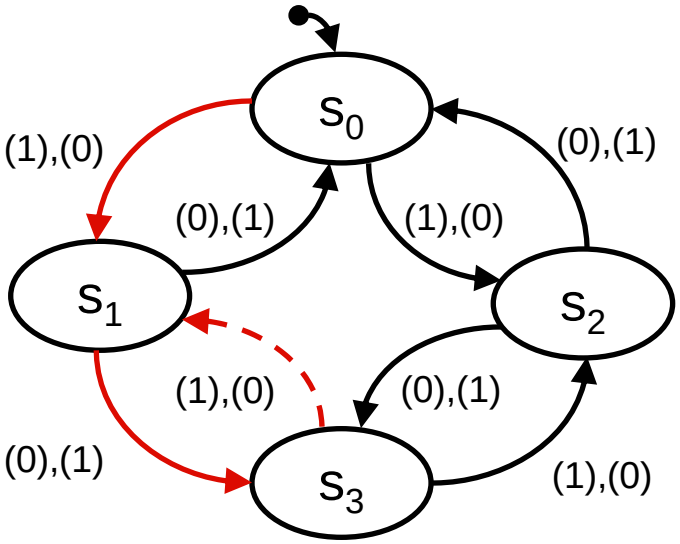
# Classification algorithm

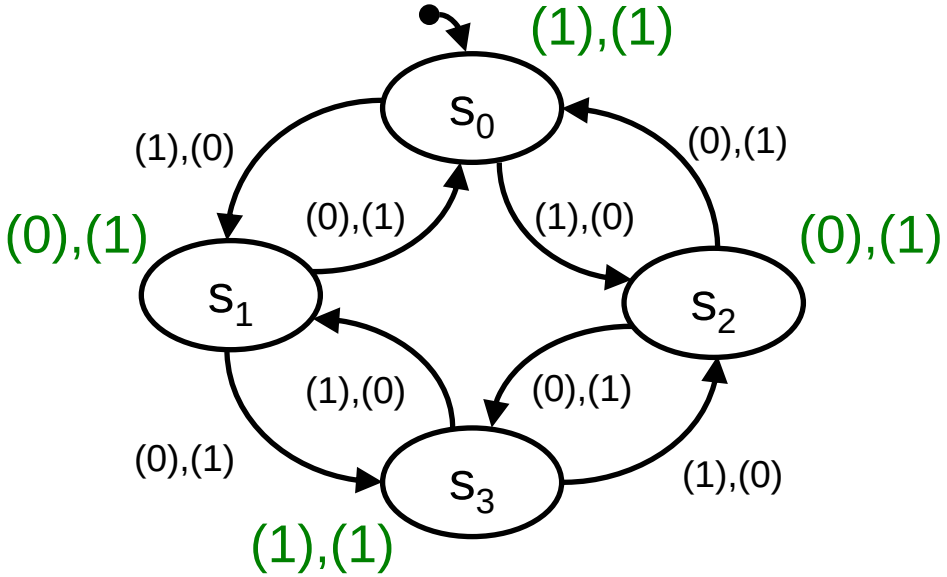



- Maximum number of different classification candidates constructed is |S| → Overall worst-case time complexity is $O(|S|(|S| + |E|)) = O(|S|^2 + |S||E|)$
- Example:
  - $c_1.\tau$ := 1, $c_1$.cns(1) := (1), $c_1$.prd(1) := (0)

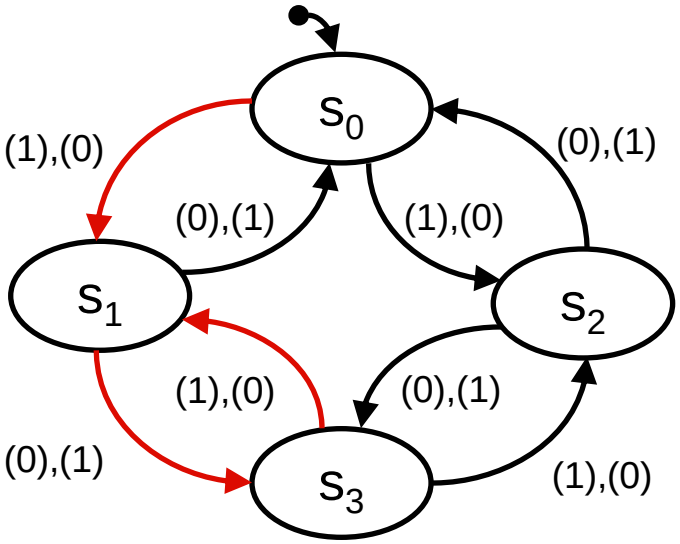
# Classification algorithm

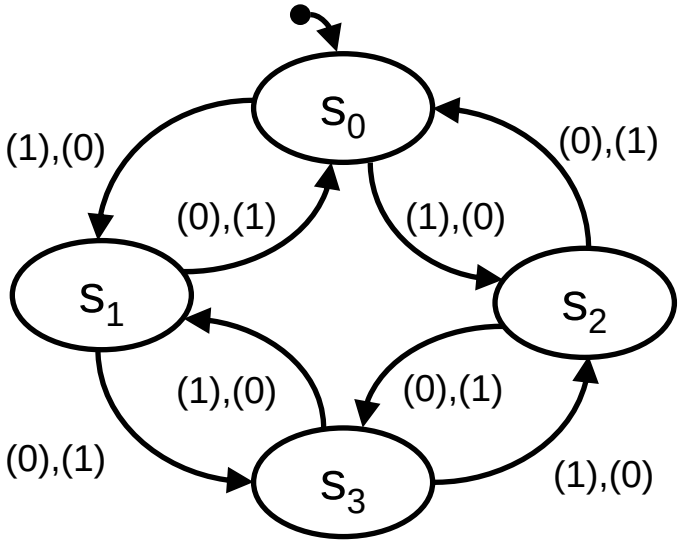



- Maximum number of different classification candidates constructed is |S| → Overall worst-case time complexity is $O(|S|(|S| + |E|)) = O(|S|^2 + |S||E|)$
- Example:
  - $c_1.\tau$ := 1, $c_1$.cns(1) := (1), $c_1$.prd(1) := (0) (discarded)
  - $c_2.\tau$ := 1, $c_2$.cns(1) := (1), $c_2$.prd(1) := (1)

# Classification algorithm




- Maximum number of different classification candidates constructed is |S| → Overall worst-case time complexity is $O(|S|(|S| + |E|)) = O(|S|^2 + |S||E|)$
- Example:
  - $c_1.\tau$ := 1, $c_1$.cns(1) := (1), $c_1$.prd(1) := (0) (discarded)
  - $c_2.\tau$ := 1, $c_2$.cns(1) := (1), $c_2$.prd(1) := (1) (accepted)
  - $c_3.\tau$ := 2, $c_3$.cns(2) := (1), $c_3$.prd(2) := (0) (not needed)

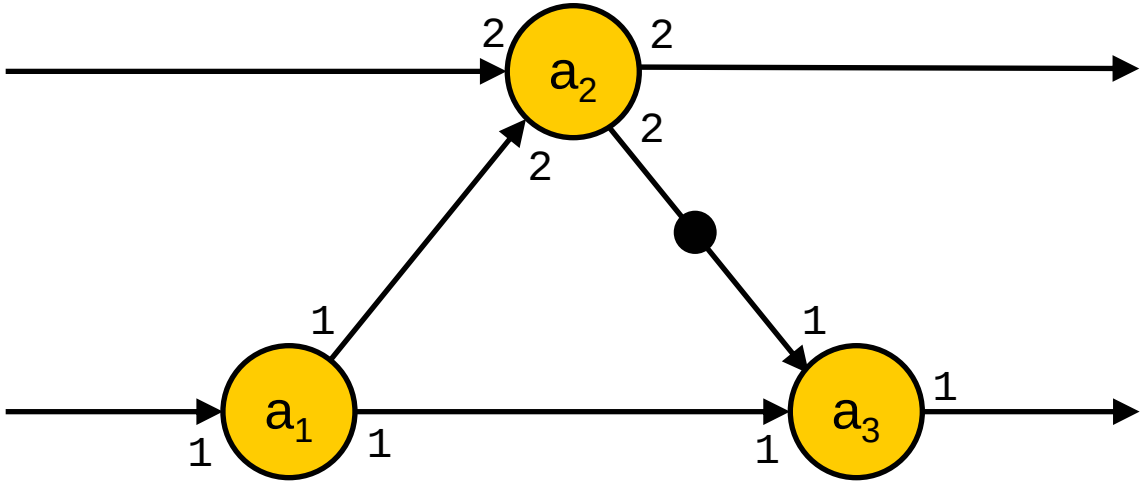
# Classification algorithm

- Problem: Functionality state of actor may be
  - not known in advance
  - infinite
  - non-deterministic
- Dynamic state transition diagram can not be constructed / analyzed in such a case
- In order to be able to classify those actors anyhow, one can treat the functionality state of an actor as „black box“ and assume guards are always evaluated to true, regardless of the current functionality state
- This leads to a state transition diagram based on the FSM
- Classification algorithm can also be applied to this diagram

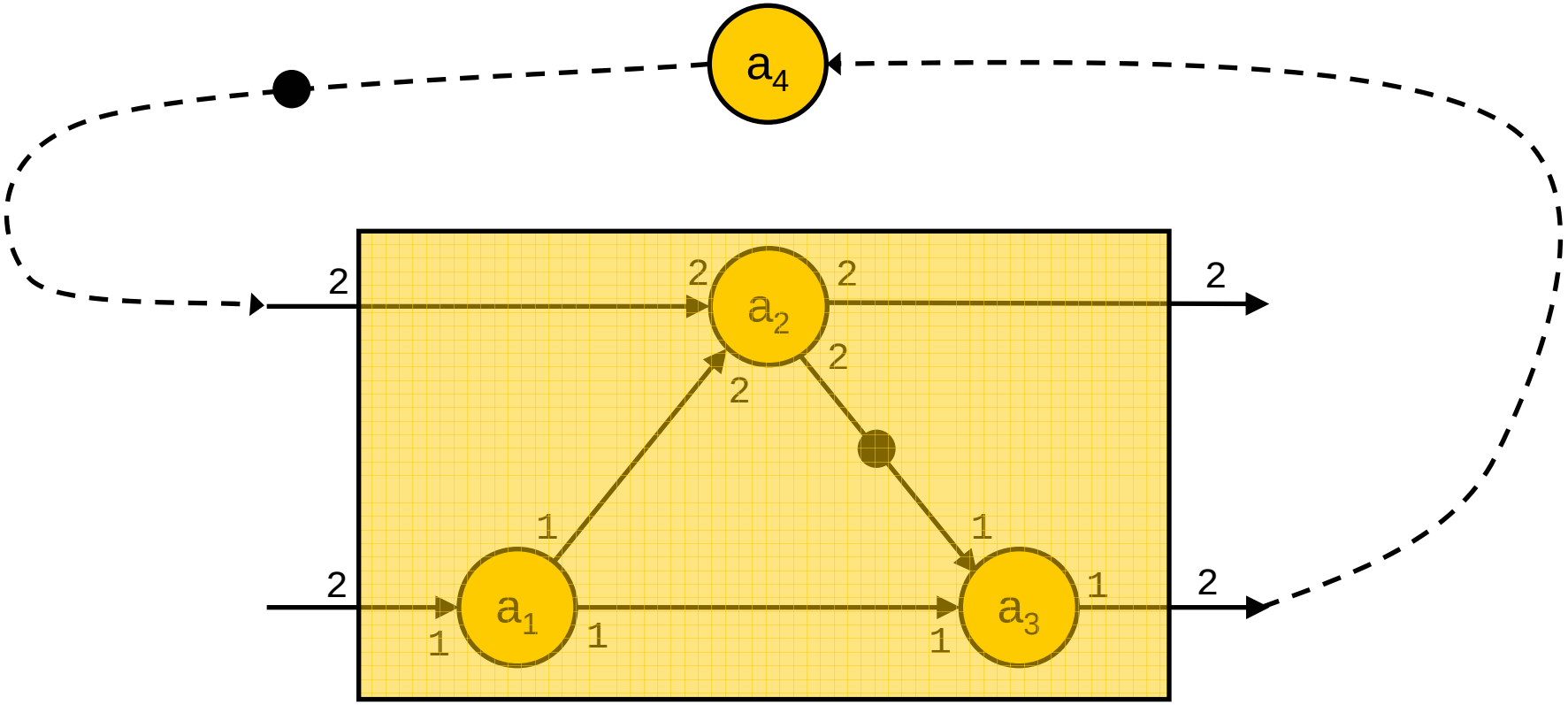
# Agenda

- Motivation
- The SystemCoDesigner modeling approach
- Automatic actor classification
- Clustering of static dataflow actors
- Results
- Conclusion

# Synchronous Dataflow Subgraph

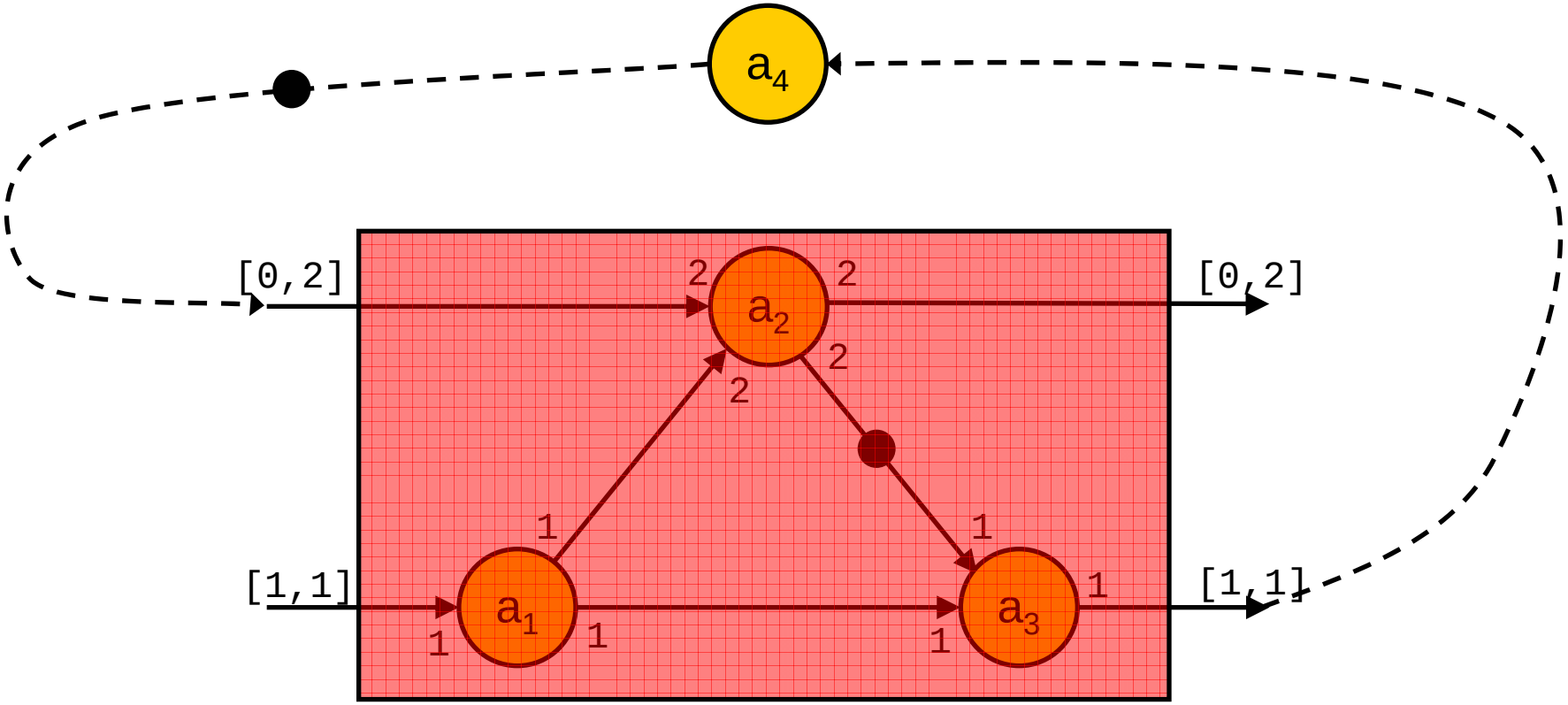


# Clustering (SDF)



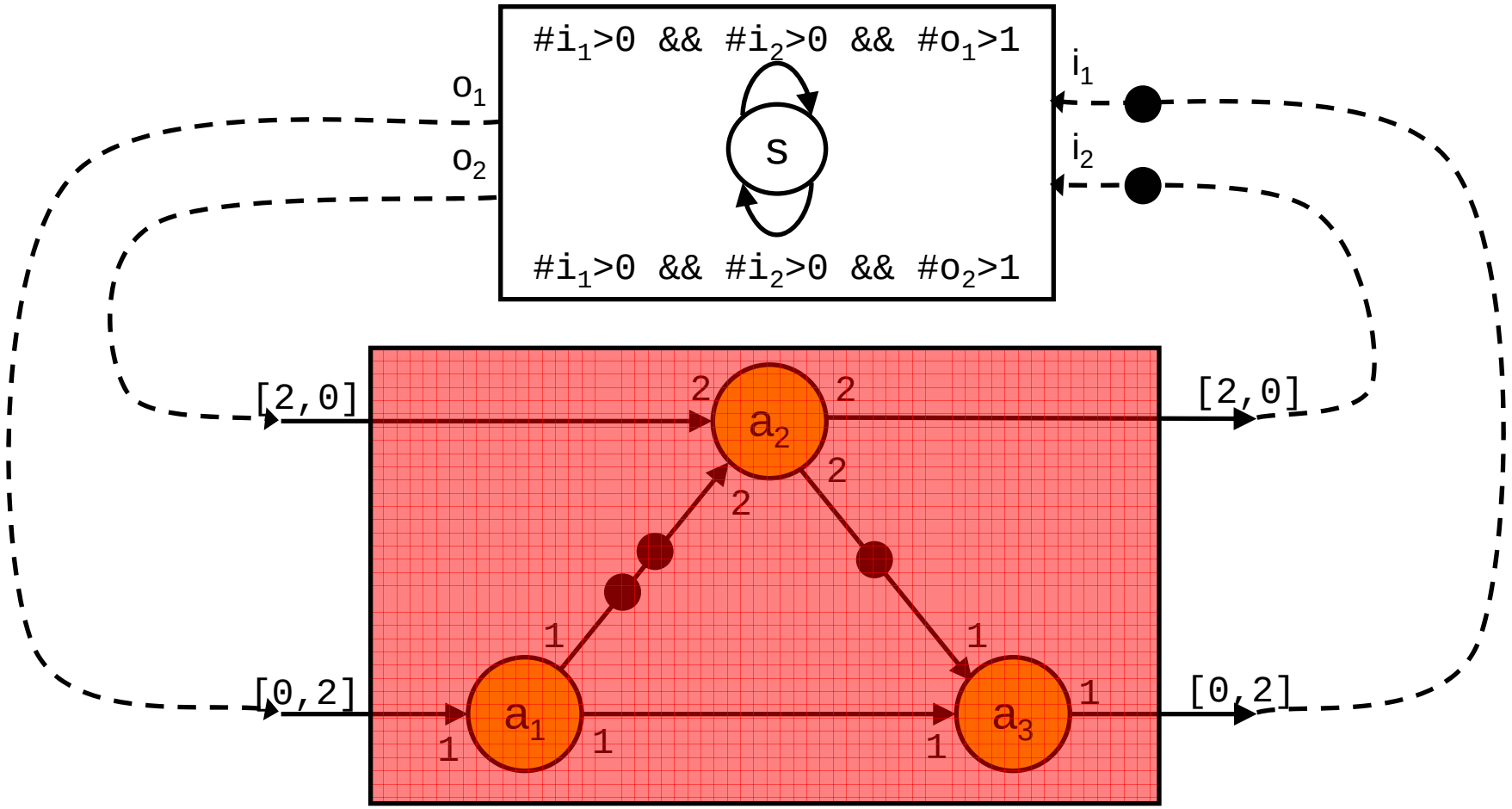

$\tau = (2xa_1, 1xa_2, 2xa_3)$

# Clustering (CSDF)



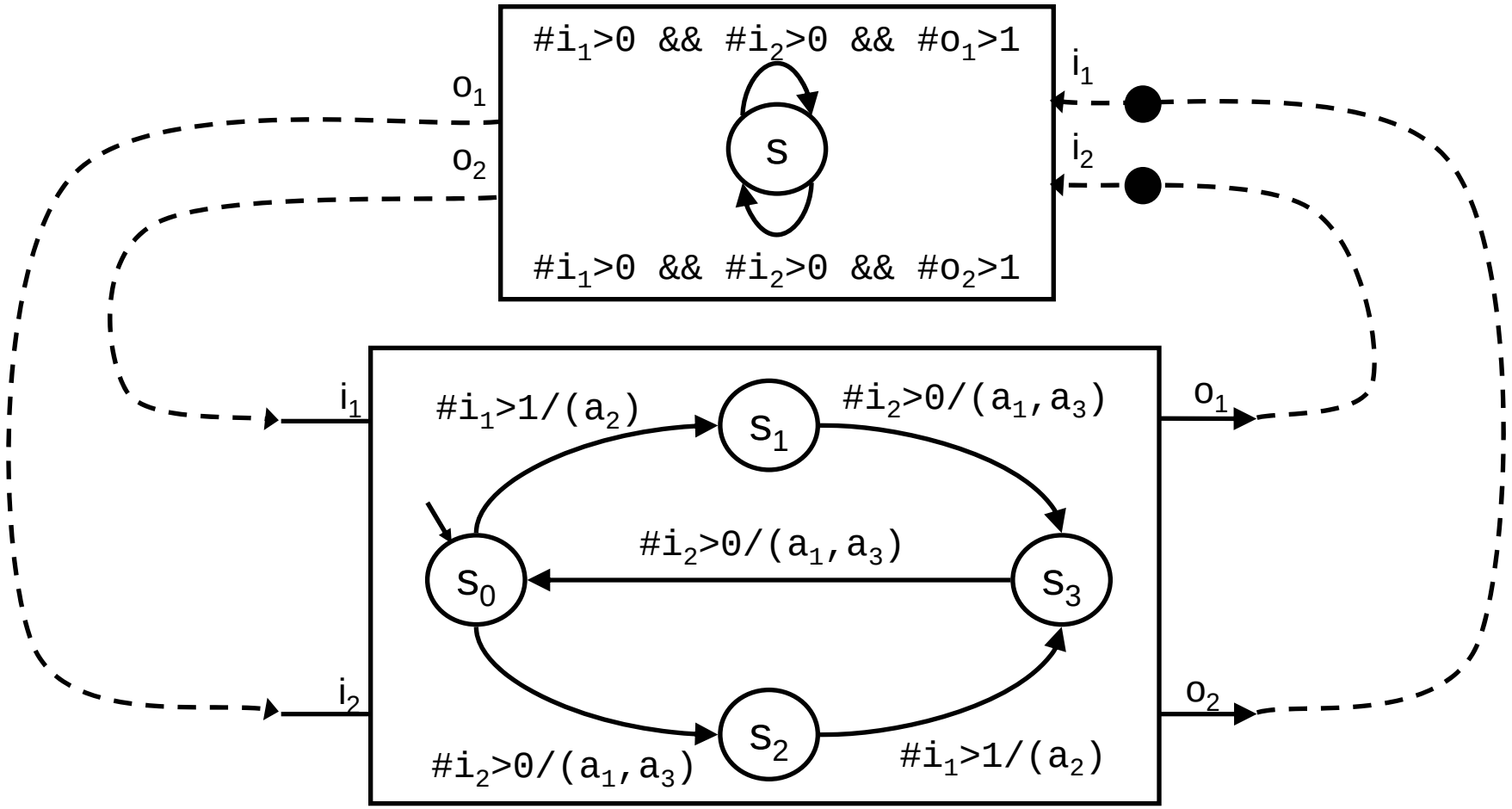

$\tau = ((1xa_1,1xa_3), (1xa_1,1xa_2, 1xa_3))$

# Clustering 2 (CSDF)



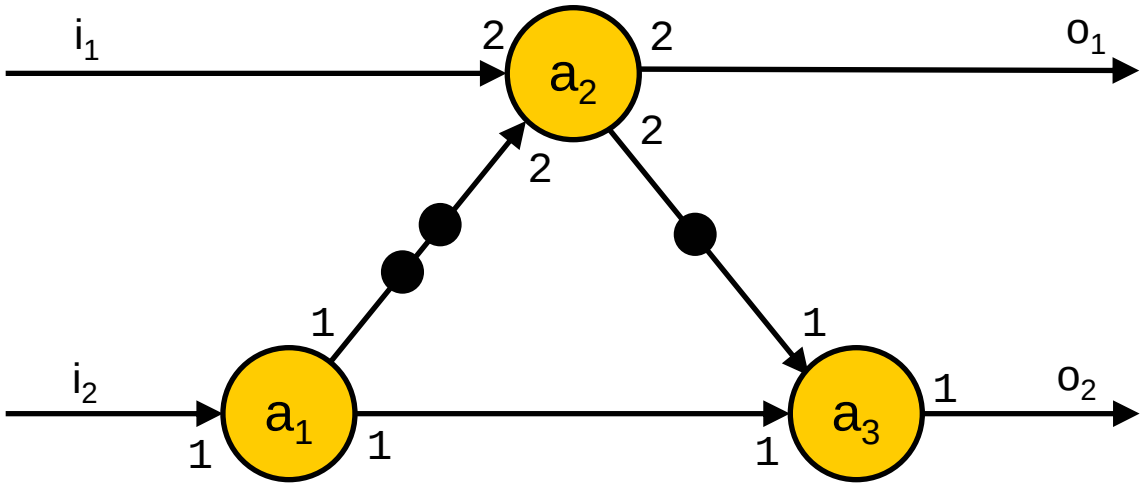

$\tau = ((1xa_2),(2xa_1,2xa_3))$

# Clustering (dynamic dataflow)

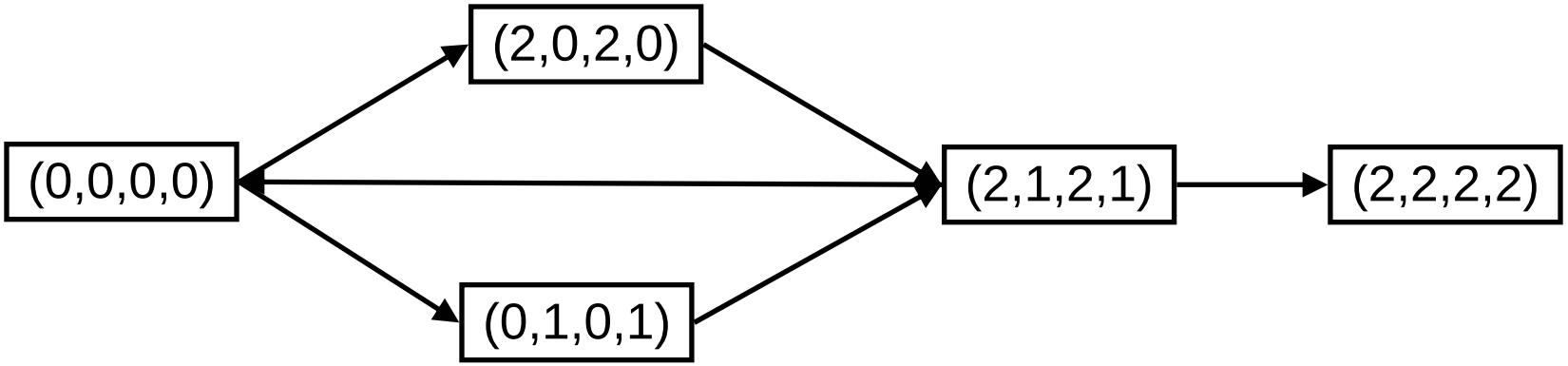


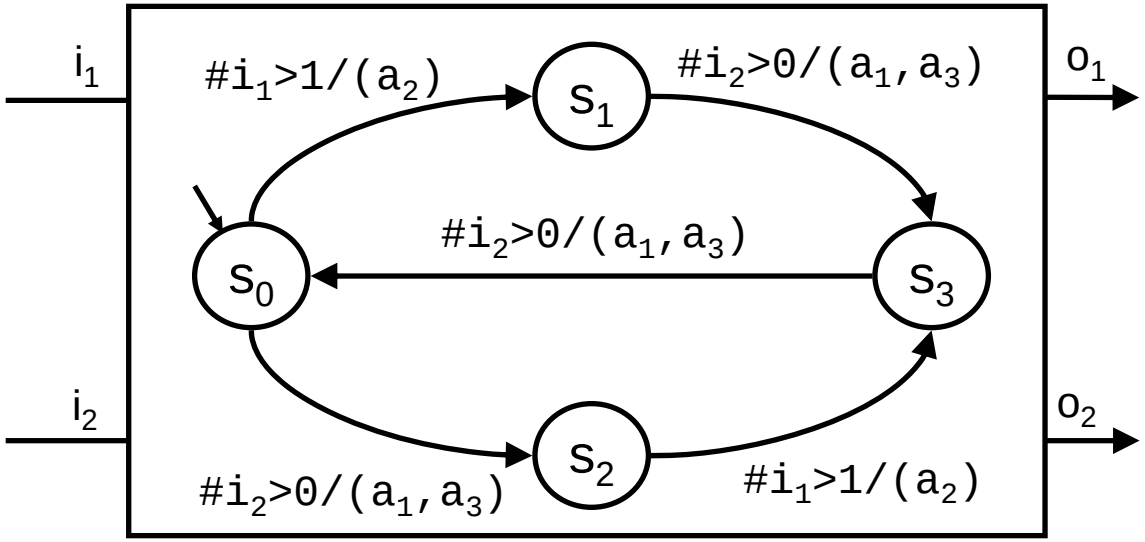
# Clustering Algorithm 1/2

| | dep(o, *n*) | | iostates | |
|---|---|---|---|---|
| | $o=o_1$ | $o=o_2$ | | |
| *n = 0* | (0,0) | (0,0) | (0,0,0,0) | (0,0,0,0) |
| *n = 1* | (2,0) | (0,1) | (2,0,2,0) | (0,1,0,1) |
| *n = 2* | (2,0) | (2,2) | (2,0,2,0) | (2,2,2,2) |
| *n = 3* | (4,2) | (2,3) | - | - |



# Clustering Algorithm 2/2

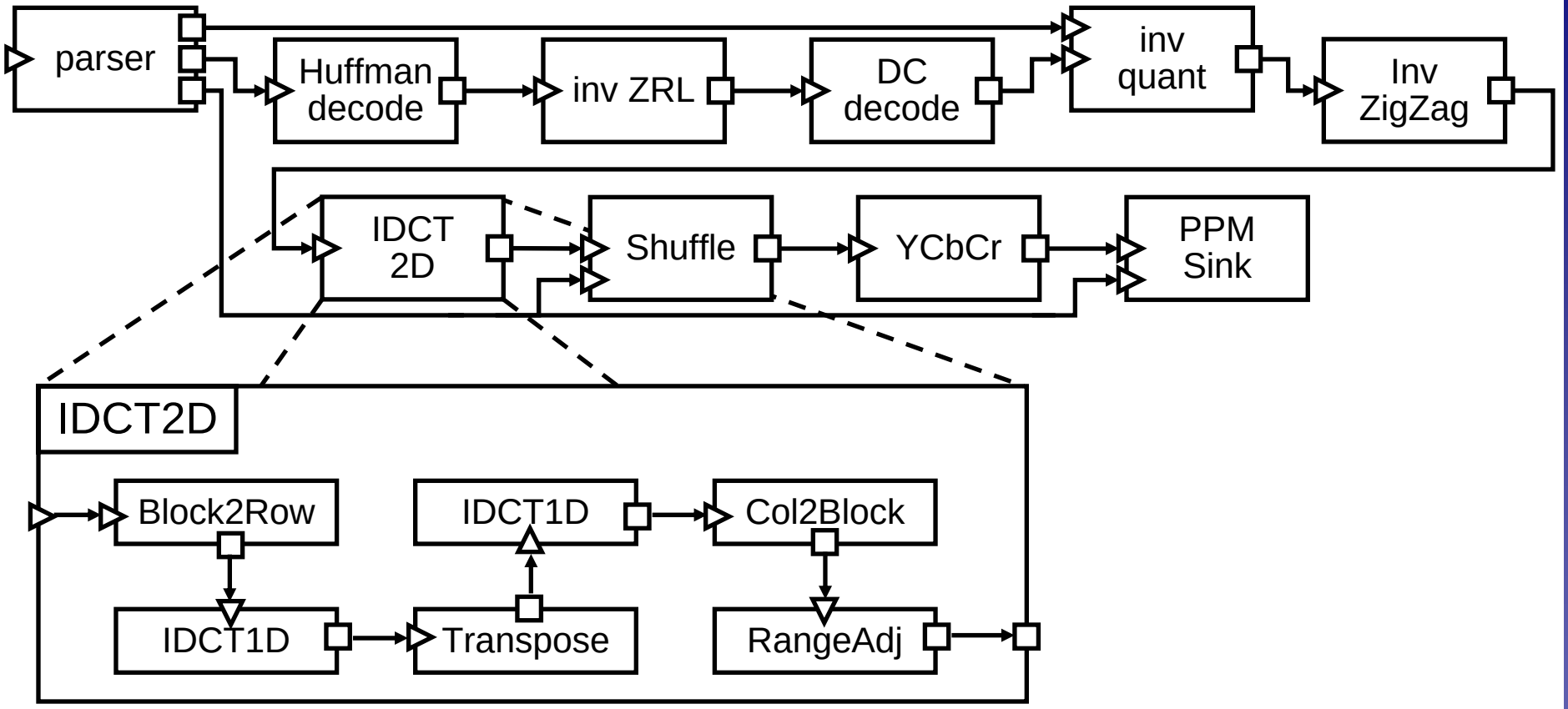


# Agenda

- Motivation
- The SystemCoDesigner modeling approach
- Automatic actor classification
- Clustering of static dataflow actors
- Results
- Conclusion

# Results 1/2




- Automatic actor classification → Inv. Quant, Inv. ZigZag, YCbCr and actors of hierarchical IDCT2D recognized as SDF / CSDF

# Results 2/2

- Automatic clustering → Quasi-static schedule for static parts (embedded in dynamic data flow environment)
  - Throughput improvement: 57%

- IDCT2D mapped onto four processor MPSoC implemented on a Xilinx Virtex II Pro
  - Throughput improvement: 95%

# Agenda

- Motivation
- The SystemCoDesigner modeling approach
- Automatic actor classification
- Clustering of static dataflow actors
- Results
- Conclusion

# Conclusion

- Application in the signal processing domain are often modeled as data flow graphs
- Complex nature of today‘s systems →Static and dynamic data flow actors
- Our formal notation accomodates both of these extremes
- Classification enables us to recognize a subset of SDF / CSDF actors modeled using our notation
- Domain-specific optimization methods can be subsequently applied to static parts of the network graph

# Conclusion

**Thank you for your attention!**

Questions?