# $f_{CASP}$: a Forgetting Operator and its Application to Energy Distribution Under a Goal-Directed ASP Decision Model

Luciana Camila Fidilio Allende

unir LA UNIVERSIDAD EN INTERNET

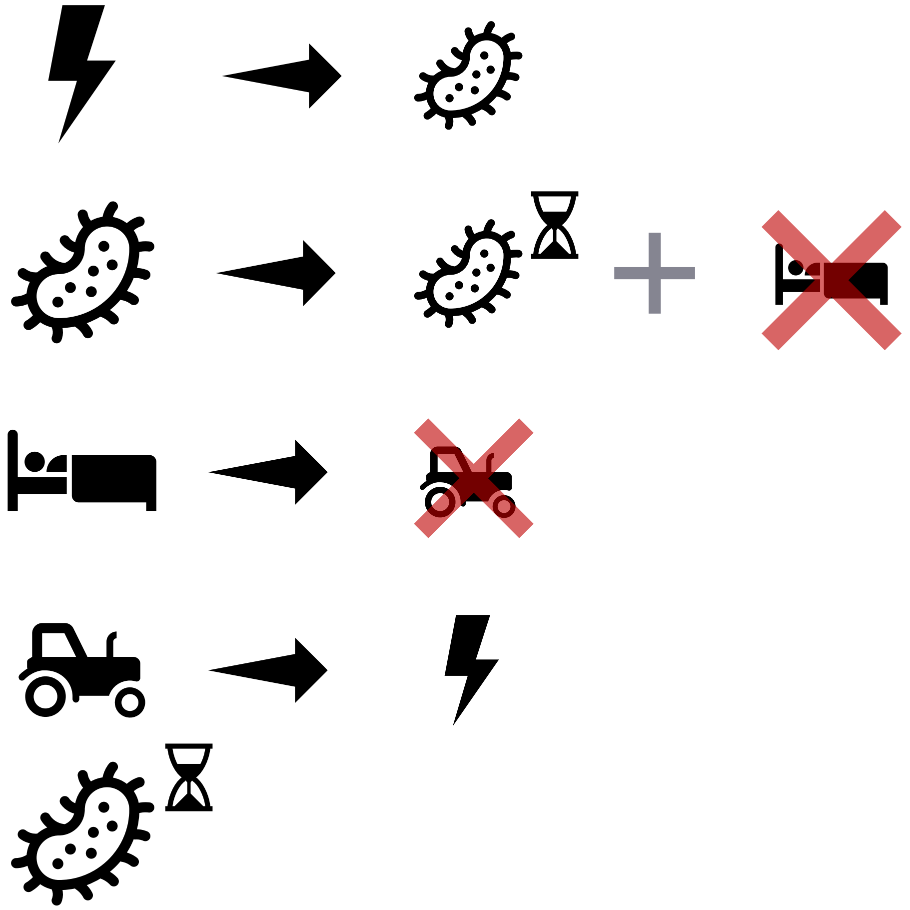
Universidad de Alcalá

Universidad Rey Juan Carlos

# Introduction

- **Decision models** can automate the allocation of crucial resources in cooperative/competitive contexts.
  - We can use **ASP** to program decision models.

```
energy_pepe :-
    sick_pepe.
sick_pepe:-
    past_sick_pepe,
    not rest_pepe.
rest_pepe :-
    not machine_pepe.
machine_pepe :-
    energy_pepe.
past_sick_pepe.
```



# Introduction

- Decision models can automate the allocation of crucial resources in cooperative/competitive contexts.
  - We can use ASP to program decision models.
- Explainability is needed to ensure trustworthiness.
  - ASP programs/models can provide justifications.

```
1.  energy_pepe :-
2.      sick_pepe.
3.  sick_pepe:-
4.      past_sick_pepe,
5.      not rest_pepe.
6.  rest_pepe :-
7.      not machine_pepe.
8.  machine_pepe :-
9.      energy_pepe.
10. past_sick_pepe.
```

Model: { energy_pepe, ... }

Justification:

```
energy_pepe :-
  sick_pepe :-
    past_sick_pepe,
    not rest_pepe :-
      machine_pepe :-
        chs(energy_pepe).
```

- **Decision models** can automate the allocation of crucial resources in cooperative/competitive contexts.
  - We can use **ASP** to program decision models.
- **Explainability** is needed to ensure trustworthiness.
  - ASP programs/models can provide justifications (in NL).

```
energy_pepe :-
    sick_pepe.
sick_pepe:-
    past_sick_pepe,
    not rest_pepe.
rest_pepe :-
    not machine_pepe.
machine_pepe :-
    energy_pepe.
past_sick_pepe.
```

Model: { energy_pepe, ... }

Justification:

```
energy_pepe :-                    Pepe gets electric power, because
  sick_pepe :-                    Pepe is sick, because
    past_sick_pepe,               Pepe was sick yesterday, and
    not rest_pepe :-              there is no evidence that Pepe has rested, because
      machine_pepe :-             Pepe has used the machine, because
        chs(energy_pepe).         it is assumed that Pepe gets electric power.
```

UNIR LA UNIVERSIDAD EN INTERNET · Universidad de Alcalá · Universidad Rey Juan Carlos

# Introduction

- However, the justifications (and the models) may expose private and confidential information.

```
1.  energy_pepe :-
2.      sick_pepe.
3.  sick_pepe:-
4.      past_sick_pepe,
5.      not rest_pepe.
6.  rest_pepe :-
7.      not machine_pepe.
8.  machine_pepe :-
9.      energy_pepe.
10. past_sick_pepe.
```

Model: { energy_pepe, ... }

Justification:

```
energy_pepe :-
  sick_pepe :-
    past_sick_pepe,
    not rest_pepe :-
      machine_pepe :-
        chs(energy_pepe).
```

Sensitive information

UNIR LA UNIVERSIDAD EN INTERNET · Universidad de Alcalá · Universidad Rey Juan Carlos

# Introduction

- However, the justifications (and the models) may **expose** private and confidential information.
- So, we must **protect** these sensitive information:
  a) By **manipulating** the justification.

```
1.  energy_pepe :-
2.      sick_pepe.
3.  sick_pepe:-
4.      past_sick_pepe,
5.      not rest_pepe.
6.  rest_pepe :-
7.      not machine_pepe.
8.  machine_pepe :-
9.      energy_pepe.
10. past_sick_pepe.
```

Model: { energy_pepe, ... }

Justification:

```
energy_pepe :-
  ~~sick_pepe :-~~
    past_sick_pepe,
    not rest_pepe :-
      machine_pepe :-
        chs(energy_pepe).
```

# Introduction

- However, the justifications (and the models) may **expose** private and confidential information.
- So, we must **protect** these sensitive information:
  a) By **manipulating** the justification.
  b) Applying **forgetting** (removing predicates in ASP programs).

```
1.  energy_pepe :-
2.      ~~sick_pepe.~~
3.  ~~sick_pepe:-~~
4.      past_sick_pepe,
5.      not rest_pepe.
6.  rest_pepe :-
7.      not machine_pepe.
8.  machine_pepe :-
9.      energy_pepe.
10. past_sick_pepe.
```

Model: { energy_pepe, ... }

Justification:

```
energy_pepe :-
  past_sick_pepe,
  not rest_pepe :-
    machine_pepe :-
      chs(energy_pepe).
```

UNIR LA UNIVERSIDAD EN INTERNET · Universidad de Alcalá · Universidad Rey Juan Carlos

# Introduction

- However, the justifications (and the models) may expose private and confidential information.
- So, we must protect these sensitive information:
  a) By manipulating the justification.
  b) Applying forgetting (removing predicates in ASP programs).

We present $\mathbf{f_{CASP}}$, a forgetting technique based on s(CASP) that removes predicates in ASP programs with Denials.

# Background: ASP and s(CASP)

- Answer Set Programming is based on the stable model semantics:
  - Supports non-stratified negation (even loops).
  - May provide multiple models.
  - We extend ASP with double default negations (not not).

- s(CASP) is a goal-directed interpreter of ASP with Constraints:

2

solves negated atoms 'not p(X)' against **dual rules**

provides **justifications** in natural language

can **manipulate** the justifications (#show and --short)

# $f_{CASP}$: Design

$f_{CASP}$ removes predicates from ASP programs with denials:

- Supports even and odd loops.
- Based on s(CASP) dual rules.
- Implemented as part of s(CASP).

The $f_{CASP}$ algorithm consists of 4+1 steps:

1. Add auxiliary predicates (neg_x).
2. Generate the simplified dual rule(s) using s(CASP).
3. Forget the predicate and its negation.
4. Clean extra clauses and add double negations.
5. (Optional) Transform double negations.

- Could be extended to support variables and constraints.

# $f_{CASP}$: Implementation

```
f_casp([Pred|Preds], P_0, P_Forgetting, Flag) :-
        transform_even_loops(Pred, P_0, P_1a, Neg_Pred),        % Step 1
        add_clauses_if_needed(Pred, P_1a, P_1b),
        delete_auto_calls(Pred, P_1b, P_1c),
        generate_dual(Pred, P_1c, Dual_Rule),                    % Step 2
        forget_pred(Pred, Dual_Rule, P_1c, P_3),                 % Step 3
        restore_even_loop(Neg_Pred, P_3, P_4a),                  % Step 4
        restore_facts_missing(P_4a, P_4b),
        f_casp(Preds, P_4b, P_Forgetting, Flag).                 % Repeat 1,2,3,4
f_casp([], P_Forgetting, P_Forgetting, 0).                       % Skip Step 5
f_casp([], P_Forgetting, P_sCASP, 1) :                           % Step 5
        transform_double_negations(P_Forgetting, P_Scasp).
```

- Available at [https://gitlab.software.imdea.org/ciao-lang/sCASP](https://gitlab.software.imdea.org/ciao-lang/sCASP)

# $f_{CASP}$: Usage instructions

- To apply $f_{CASP}$, just run s(CASP) using the following flag:

--forget = LIST[/F]

List of predicates to be forgotten

F = 0 skip step 5
F = 1 (default) execute step 5

- E.g.: scasp energy.pl --forget='sick_pepe'

DEMO

unir LA UNIVERSIDAD EN INTERNET · Universidad de Alcalá · Universidad Rey Juan Carlos

# $f_{CASP}$: Example

- Forgetting ‘sick_pepe’.

Original program:

1. energy_pepe :-
2. sick_pepe.
3. sick_pepe:-
4. past_sick_pepe,
5. not rest_pepe.
6. rest_pepe :-
7. not machine_pepe.
8. machine_pepe :-
9. energy_pepe.
10. past_sick_pepe.

{ energy_pepe, sick_pepe, past_sick_pepe, not rest_pepe, machine_pepe }

$f_{CASP}$ {sick_pepe}:

1. energy_pepe :-
2. past_sick_pepe,
3. not rest_pepe.
4. rest_pepe :-
5. not machine_pepe.
6. machine_pepe :-
7. energy_pepe.
8. past_sick_pepe.

{ energy_pepe, past_sick_pepe, not rest_pepe, machine_pepe }

# Preliminary validation through examples

- Example with double negations from Berthold et al. 2019 :

Original program:

```
q :- not not q,
        b.
a :- q.
c :- not q.
```

{c}

$f_{CASP}$ {p,q}:

```
a :-  not neg_1,
        b.
c :- not not neg_1.
neg_1 :- not not neg_1.
neg_1 :- not b.
```

{c, neg_1}

unir LA UNIVERSIDAD EN INTERNET · Universidad de Alcalá · Universidad Rey Juan Carlos

# Preliminary validation through examples

- Comparing the required auxiliary predicates ($f_{AC}$ vs. $f_{CASP}$):

Original program:

```
q :- not not q,
        b.
a :- q.
c :- not q.
```

{c}

$f_{CASP}$ {p,q}:

```
a :-  not neg_1,
        b.
c :- not not neg_1.
neg_1 :- not not neg_1.
neg_1 :- not b.
```

{c, neg_1}

$f_{AC}$ {p,q}:

$a$ :- $b$,
$\quad\quad \delta_q$.
$c$ :- not $\delta_q$.
$c$ :- not $b$.
$\delta_q$ :- not not $\delta_q$.

{c}  $\{c, \delta_q\}$

# Comparison between forgetting operators

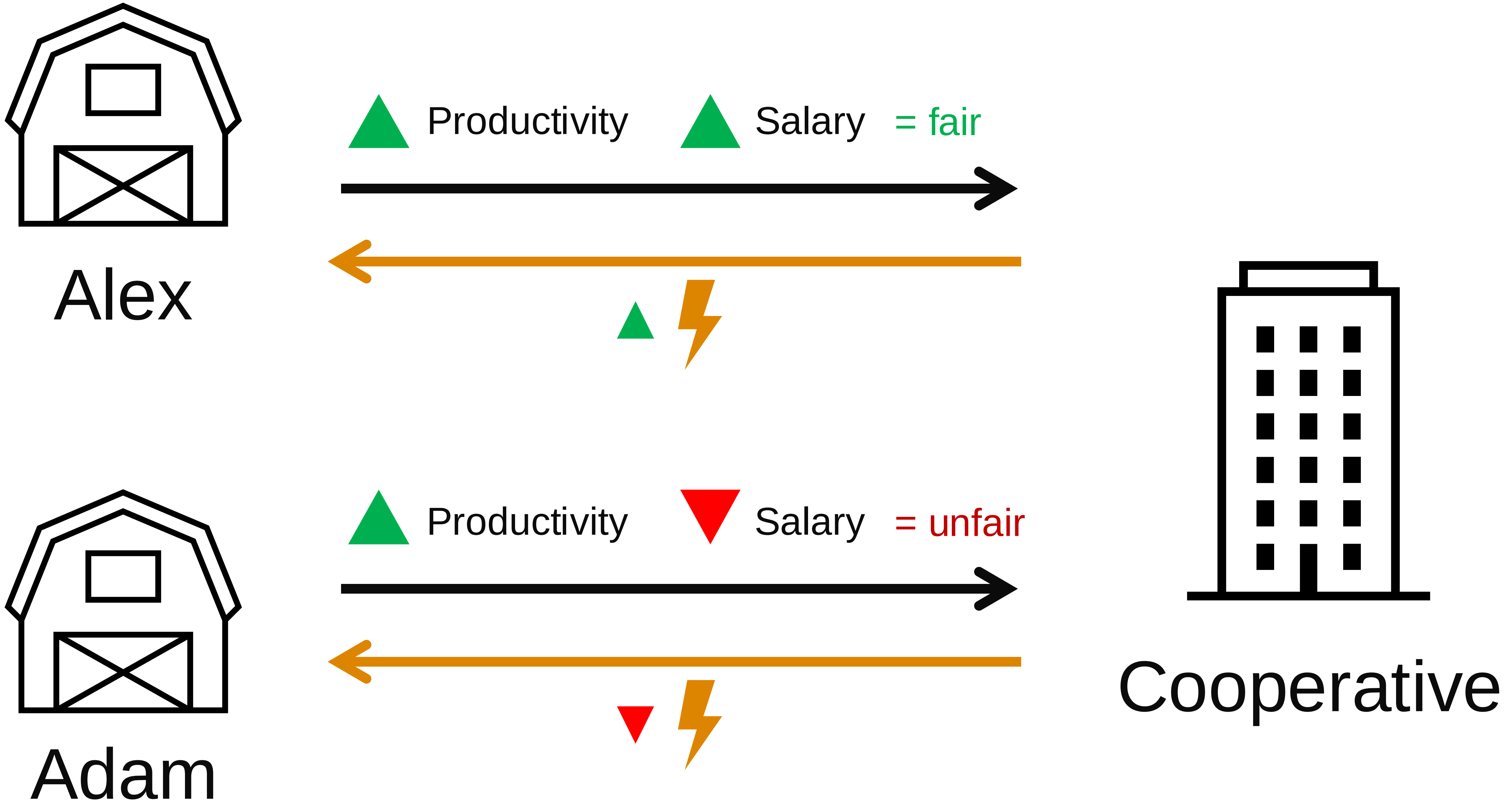
| | (UP) | (SP) | Loops | Commutative | Predicates | Constraints |
|---|---|---|---|---|---|---|
| $f_{SU}$ | ✓ | x | ✓ | x | x | x |
| $f_{SP}$ | ✓ | Limited | x | x | x | x |
| $f^*_{SP}$ | ✓ | Limited | ✓ | ✓ | x | x |
| $f_{AC}$ | ✓ | ✓ | ✓ | ✓ | x | x |
| $f_{CASP}$ | ✓ | ✓ | ✓ | ✓ | WiP | WiP |

# Transparent and fair energy assignment applying $f_{CASP}$

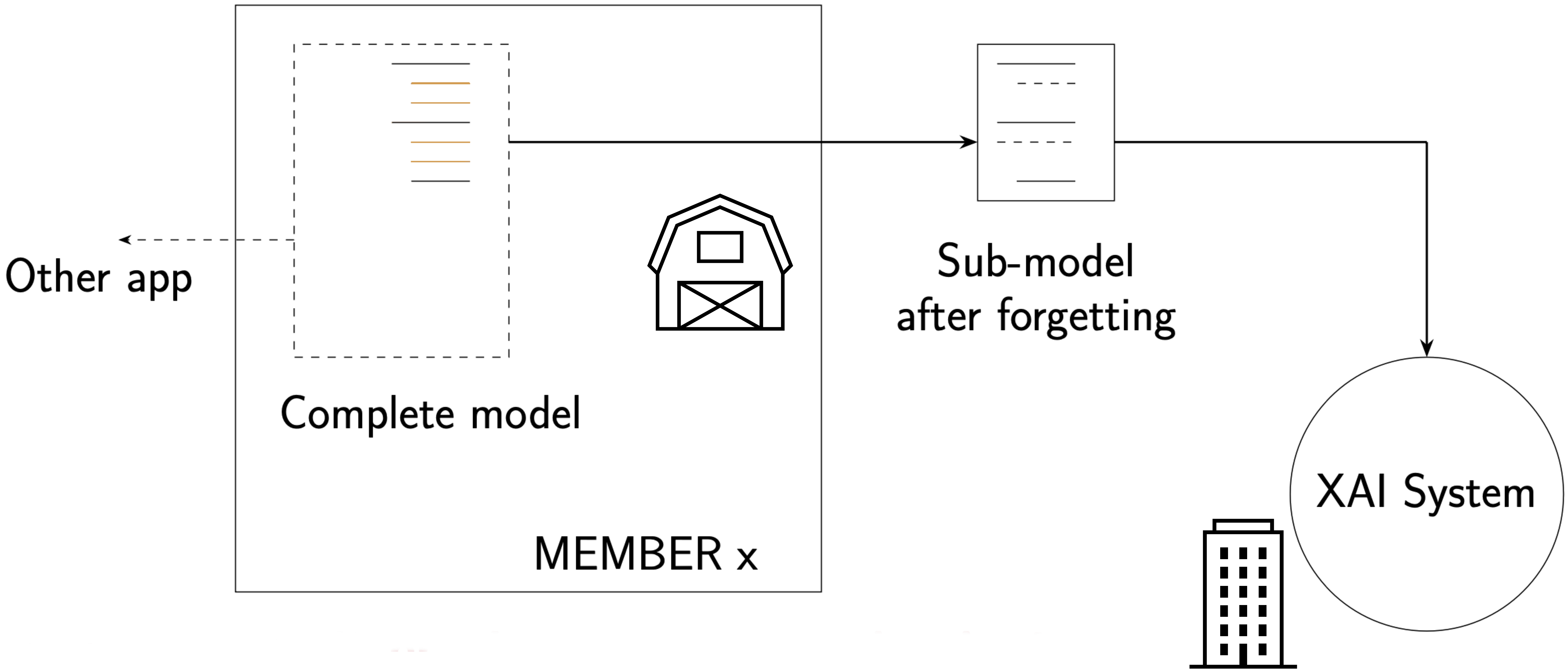
- Consider local power generation managed by an agricultural cooperative to provide an alternative energy supply.
- Its assignment can encourage better practices if we base the energy distribution criteria on human-values.
  - E.g., on a fair income for agricultural workers.
- But to rely on the decision process, the members want an explanation.

However, the explanation of a decision must not expose members' trade secrets.

# Transparent and fair energy assignment applying $f_{CASP}$



# Transparent and fair energy assignment applying $f_{CASP}$



# Transparent and fair energy assignment applying $f_{CASP}$

## Cooperative program:

```
percentages([alex(PercentageAlex), ...]):-
    fair_income_alex(PointsAlex),
    fair_income_adam(PointsAdam),
    ...
    ponder(PointsAlex, Max, PercentageAlex).

fair_income_alex(PointsAlex):-
     salary(bea, Salary),
     ...
     productivity(bea, Productivity),
     ...
fair_income_adam(PointsAdam), :-
    ...
```

## Alex's program:

```
over_40_bea :- not neg_over_40_bea.
neg_over_40_bea :- not over_40_bea.

generational_renewal(bea,0) :- over_40_bea.
generational_renewal(bea,100) :- not over_40_bea.

salary(bea,Salary) :-
        base_salary(bea,S0),
        generational_renewal(bea,S1),
        holiday_worked(bea,S2),
        Salary is S0+S1+S2.
```

UNIR LA UNIVERSIDAD EN INTERNET
Universidad de Alcalá
Universidad Rey Juan Carlos

# Transparent and fair energy assignment applying $f_{CASP}$

Alex's program:

```
over_40_bea :- not neg_over_40_bea.
neg_over_40_bea :- not over_40_bea.

generational_renewal(bea,0) :- over_40_bea.
generational_renewal(bea,100) :- not over_40_bea.

salary(bea,Salary) :-
        base_salary(bea,S0),
        generational_renewal(bea,S1),
        holiday_worked(bea,S2),
        Salary is S0+S1+S2.
```

business secrecy
(salary supplements)

# Transparent and fair energy assignment applying $f_{CASP}$

Alex’s program:

```
over_40_bea :- not neg_over_40_bea.
neg_over_40_bea :- not over_40_bea.

generational_renewal(bea,0) :- over_40_bea.
generational_renewal(bea,100) :- not over_40_bea.

salary(bea,Salary) :-
        base_salary(bea,S0),
        generational_renewal(bea,S1),
        holiday_worked(bea,S2),
        Salary is S0+S1+S2.
```

After forgetting sensitive predicates:

```
neg_1 :- not neg_2.
neg_2 :- not neg_1.

salary(bea, Salary) :-
        S0 = 900,
        neg_2,
        S1 = 0,
        S2 = 0,
        Salary is S0 + S1 + S2.

salary(bea,Salary) :-
        S0 = 900,
        neg_1,
        S1 = 100,
        S2 = 0,
        Salary is S0+S1+S2.
```

# Transparent and fair energy assignment applying $f_{CASP}$

## Original justification

```
percentages([adam(10.13), alex(29.27), ...]) :-
  ...
  fair_income_alex(1.08) :-
      salary(bea,900) :-
            base_salary(bea,900),
            generational_renewal(bea,0) :-
              over_40_bea :-
                not neg_over_40_bea :-
                  chs(over_40_bea).
            holiday_worked(bea,0),
            900 is 900+(0+0).
      productivity(bea,1040) :-
            ...
  fair_income_adam(0.74) :-
      ...
```

## After manipulating the justification

```
percentages([adam(10.13), alex(29.27), ...]) :-
  ...
  fair_income_alex(1.08) :-
      salary(bea,900) :-
         900 is 900+(0+0).
         ...
```

## After forgetting sensitive predicates

```
percentages([adam(10.13), alex(29.27), ...]) :-
...
  fair_income_alex(1.08) :-
      salary(bea,900) :-
         neg_2 :-
           not neg_1 :-
             chs(neg_2).
         900 is 900+(0+0).
         ...
```

## Conclusions

- We have presented and evaluated $f_{CASP}$, a forgetting operator that:
  - Supports **even loops**
  - Can be extended to support **predicates and constraints**
- We have applied $f_{CASP}$ to achieve a **fair (and trustworthy)** energy distribution decision model.

## Future work

- **Extend it with** variables, arithmetic constraints and recursive predicates.
- Determining and **proving formally** the $f_{CASP}$'s forgetting properties.
- **Splitting CASP programs** based on their predicate stratification.

# thank you

unir LA UNIVERSIDAD EN INTERNET
Universidad de Alcalá
Universidad Rey Juan Carlos

# Bibliography I


Arias, J., Carro, M., Chen, Z., and Gupta, G. (2020). Justifications for Goal-Directed Constraint Answer Set Programming. In: Proceedings 36th International Conference on Logic Programming (Technical Communications). Vol. 325. EPTCS. Open Publishing Association, pp. 59–72. doi: 10.4204/ EPTCS.325.12.

Arias, J., Carro, M., Salazar, E., Marple, K., and Gupta, G. (2018). Constraint Answer Set Programming without Grounding. In: Theory and Practice of Logic Programming 18(3-4), pp. 337–354. doi: 10.1017/S1471068418000285.

Berthold, M. (2022). On Syntactic Forgetting with Strong Persistence. In: Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning. Vol. 19, pp. 43–52.

Berthold, M., Gonçalves, R., Knorr, M., and Leite, J. (2019). Forgetting in Answer Set Programming with Anonymous Cycles. In: Progress in Artificial Intelligence: 19th Conference on Artificial Intelligence EPIA. Springer, pp. 552–565. doi: 10.1007/978-3-030-30244-3_46.

Fidilio-Allende, L. and Arias, J. (2024). fCASP: A forgetting technique for XAI based on goal-directed constraint ASP models. In: XXIII Jornadas sobre Programación y Lenguajes (PROLE). url: https://hdl.handle.net/11705/PROLE/2024/13.

Fidilio-Allende, L. and Arias, J. (2024). Private-Safe (Logic-based) Decision Systems for Energy Assignment in Agricultural Cooperatives. In: Workshop on Adaptive Smart Areas and Smart Agents at PAAMS.

# Bibliography II

Gonçalves, R., Knorr, M., and Leite, J. (2016). You can't always forget what you want: on the limits of forgetting in Answer Set Programming. In: Proceedings of the Twenty-second European Conference on Artificial Intelligence, pp. 957–965.

Gonçalves, R., Knorr, M., and Leite, J. (2023). Forgetting in Answer Set Programming–A Survey. In: Theory and Practice of Logic Programming 23(1), pp. 111–156.

Knorr, M. and Alferes, J. J. (2014). Preserving Strong Equivalence while Forgetting. In: Logics in Artificial Intelligence: 14th European Conference, JELIA 2014. Springer, pp. 412–425. doi: 10.1007/978-3-319-11558-0_29.