Lecture 25

CS 6260 Automated Planning and Learning

Active Model Learning

Department of Computer Science and Engineering

Indian Institute of Technology Madras

Logistics

Logistics

- 3 Exams:
  1. [Quiz 1] Feb 19: 08:00 AM – 08:50 AM
  2. [Quiz 2] Apr 10: 05:00 PM – 05:50 PM
  3. [End Sem] May 12

- Office Hours: 1 hour after class every week Tu, W, Th
  - No office hours today

Recap: FAMA

Propositional Encoding: Instantiating Predicates

Assign objects in Ω to the arguments of the predicates in Ψ, i.e.
$\Psi_v = p(\omega) : p \in \Psi, \omega \in \Omega^{ar(p)}$

Example with $\Omega_v = \{v_1, v_2\}$

$$\begin{aligned}\Psi_v = \{&\mathit{handempty},\\ &\mathit{holding}(v_1),\ \mathit{holding}(v_2),\\ &\mathit{clear}(v_1),\ \mathit{clear}(v_2),\\ &\mathit{ontable}(v_1),\ \mathit{ontable}(v_2),\\ &\mathit{on}(v_1, v_1),\ \mathit{on}(v_1, v_2),\ \mathit{on}(v_2, v_1),\ \mathit{on}(v_2, v_2)\}\end{aligned}$$

Propositional Encoding: Instantiating Actions

For a given action model ξ, we define $\Psi_\xi \subseteq \Psi_v$ as the subset of elements of Ψ_v that can appear in ξ.

Example

$\Psi_{stack} = \Psi_v$

$\Psi_{pickup} = \{handempty, holding(v_1), clear(v_1), ontable(v_1), on(v_1, v_1)\}$

We now create propositional encoding of the model fluents of $pre(\xi)$, $del(\xi)$ and $add(\xi)$.

Propositional Encoding: Instantiating Actions

For every ξ and $p \in \Psi_\xi$, we create:

- $pre_p(\xi)$: pre + $name(\xi)$ + (fluent of arity 0 using elements of p).
  Example: $pre_p(\xi) =$ `pre_stack_on_v1_v2`, when $name(\xi) = stack$ and $p = on(v_1, v_2)$.

- $del_p(\xi)$: del + $name(\xi)$ + (fluent of arity 0 using elements of p).
  Example: $del_p(\xi) =$ `del_stack_on_v1_v2`, when $name(\xi) = stack$ and $p = on(v_1, v_2)$.

- $add_p(\xi)$: add + $name(\xi)$ + (fluent of arity 0 using elements of p).
  Example: $add_p(\xi) =$ `add_stack_on_v1_v2`, when $name(\xi) = stack$ and $p = on(v_1, v_2)$.

Planning Task

Given a learning task $\Lambda = \langle \mathcal{M}, \tau \rangle$, with τ formed by an n-action sequence $\langle a_1, \ldots, a_n \rangle$ and a m-state trajectory $\langle s_0, s_1, \ldots, s_m \rangle$ ($\tau = \langle s_0, a_1, \ldots, a_n, s_m \rangle$), the compilation outputs a classical planning task $P_\Lambda = \langle F_\Lambda, A_\Lambda, I_\Lambda, G_\Lambda \rangle$.

Lets see one by one how we get $F_\Lambda, A_\Lambda, I_\Lambda$, and G_Λ.

F_Λ

In addition to F, F_Λ contains:

- Model fluents $pre_p(\xi)$, $del_p(\xi)$, and $add_p(\xi)$, for every $\xi \in \mathcal{M}$ and $p \in \Psi_\xi$.
- A set of fluents $F_\pi = \{plan(name(a_i), \Omega^{ar(a_i)}, i)\}_{1 \leq i \leq n}$ to represent i^{th} observable action of τ.
- Two fluents to iterate through the n observed actions of τ. Given $1 \leq i \leq n$:
  - at_i: Ensures that actions are executed in the same order as in τ.
  - $next_{i,i+1}$: iterates to the next planning step when solving P_Λ.
- Set of fluents $\{test_j\}_{0 \leq j \leq m}$, to point at the state observation $s_j \in \tau$ where the action model is validated.
- A fluent, $mode_{prog}$, to indicate whether action models are being programmed or validated.
- A fluent, $action_{applied}$, to force the execution of at least one action between two observed states.

I_Λ and G_Λ

I_Λ contains:

- s_0
- $mode_{prog}$
- $test_0$
- F_π
- at_1
- $\{next_{i,i+1}\}$ where $1 \leq i \leq n$.

G_Λ contains:

- $test_m$
- at_n

A_Λ: Insert Actions

Inserting Precondition

Actions which support the addition of a precondition $p \in \Psi_\xi$ to the action model $\xi \in \mathcal{M}$.

$$pre(insertPre_{p,\xi}) = \{\neg pre_p(\xi), \neg del_p(\xi), \neg add_p(\xi), mode_{prog}\},$$
$$cond(insertPre_{p,\xi}) = \{\emptyset\} \triangleright \{pre_p(\xi)\}.$$

Inserting Effects

Actions which support the addition of a negative or positive effect $p \in \Psi_\xi$ to the action model $\xi \in \mathcal{M}$.

$$\begin{aligned} pre(insertEff_{p,\xi}) =& \{\neg del_p(\xi), \neg add_p(\xi), mode_{prog}\}, \\ cond(insertEff_{p,\xi}) =& \{pre_p(\xi)\} \triangleright \{del_p(\xi)\}, \\ & \{\neg pre_p(\xi)\} \triangleright \{add_p(\xi)\}. \end{aligned}$$

A_Λ: Apply Actions; NO case

Since action headers are known, the variables $pars(\xi)$ are bounded to the objects in ω that appear in the same position.

$$
\begin{aligned}
pre(apply_{\xi,\omega}) =& \neg action_{applied} \cup \{pre_p(\xi) \implies p(\omega)\}_{\forall p \in \Psi_\xi}, \\
cond(apply_{\xi,\omega}) =& \{del_p(\xi)\} \triangleright \{\neg p(\omega)\}_{\forall p \in \Psi_\xi}, \\
& \{add_p(\xi)\} \triangleright \{p(\omega)\}_{\forall p \in \Psi_\xi}, \\
& \{\emptyset\} \triangleright \{action_{applied}\}, \\
& \{mode_{prog}\} \triangleright \{\neg mode_{prog}\}.
\end{aligned}
$$

A_Λ: Apply Actions Example

```
  (:action apply_stack
:parameters (?o1 - object ?o2 - object)
:precondition
(and (or (not(pre_stack_on_v1_v1)) (on ?o1 ?o1))
(or (not(pre_stack_on_v1_v2)) (on ?o1 ?o2))
(or (not(pre_stack_on_v2_v1)) (on ?o2 ?o1))
(or (not(pre_stack_on_v2_v2)) (on ?o2 ?o2))
(or (not(pre_stack_ontable_v1)) (ontable ?o1))
(or (not(pre_stack_ontable_v2)) (ontable ?o2))
(or (not(pre_stack_clear_v1)) (clear ?o1))
(or (not(pre_stack_clear_v2)) (clear ?o2))
(or (not(pre_stack_holding_v1)) (holding ?o1))
(or (not(pre_stack_holding_v2)) (holding ?o2))
(or (not(pre_stack_handempty)) (handempty))
(not(action_applied)))

:effect
(and (when (del_stack_on_v1_v1) (not(on ?o1 ?o1)))
(when (del_stack_on_v1_v2) (not(on ?o1 ?o2)))
(when (del_stack_on_v2_v1) (not(on ?o2 ?o1)))
(when (del_stack_on_v2_v2) (not(on ?o2 ?o2)))
(when (del_stack_ontable_v1) (not(ontable ?o1)))
 (when (del_stack_ontable_v2) (not(ontable ?o2)))
(when (del_stack_clear_v1) (not(clear ?o1)))
(when (del_stack_clear_v2) (not(clear ?o2)))
(when (del_stack_holding_v1) (not(holding ?o1)))
(when (del_stack_holding_v2) (not(holding ?o2)))
(when (del_stack_handempty) (not(handempty)))
(when (add_stack_on_v1_v1) (on ?o1 ?o1))
(when (add_stack_on_v1_v2) (on ?o1 ?o2))
(when (add_stack_on_v2_v1) (on ?o2 ?o1))
(when (add_stack_on_v2_v2) (on ?o2 ?o2))
(when (add_stack_ontable_v1) (ontable ?o1))
(when (add_stack_ontable_v2) (ontable ?o2))
(when (add_stack_clear_v1) (clear ?o1))
(when (add_stack_clear_v2) (clear ?o2))
(when (add_stack_holding_v1) (holding ?o1))
(when (add_stack_holding_v2) (holding ?o2))
(when (add_stack_handempty) (handempty))
(when (modeProg) (not(modeProg)))
(action_applied)))
```

A_Λ: Validate Actions

Aimed at checking that the observable data of the input plan trace τ follows after the execution of the apply actions.

$$pre(validate_j) = s_j \cup \{test_{j-1}\},$$
$$eff(validate_j) = \{\emptyset\} \triangleright \{\neg test_{j-1}, test_j\}.$$

- There will be a validate action in π_A for every observed state in τ.
- FOr PO and FO actions, an extra precondition, at_{i+1} is added, where i is the index of the last observed action before the state we are validating.

A_Λ: Validate Actions Example

```
(:action validate_2
    :parameters ()
    :precondition (and (not (modeProg ))(clear a)(clear b)(clear c)(clear d)
        (clear e)(not (clear f))(not (clear g)) (handempty) (not (holding a))
        (not (holding b))(not (holding c)) (not (holding d))(not (holding e))
        (not (holding f))(not (holding g))(not (on a a))(not (on a b))
        (not (on a c))(not (on a d))(not (on a e))(not (on a f))(on a g)
        (not(on b a))(not (on b b))(not (on b c))(not (on b d))(not (on b e))
        (not (on b f))(not (on b g))(not (on c a))(not (on c b))(not (on c c))
        (not (on c d))(not (on c e))(not (on c f))(not (on c g))(not (on d a))
        (not (on d b))(not (on d c))(not (on d d))(not (on d e))(not (on d f))
        (not (on d g))(not (on e a))(not (on e b))(not (on e c))(not (on e d))
        (not (on e e))(on e f)(not (on e g))(not (on f a))(not (on f b))
        (not (on f c))(not (on f d))(not (on f e))(not (on f f))(not (on f g))
        (not (on g a))(not (on g b))(not (on g c))(not (on g d))(not (on g e))
        (not (on g f))(not (on g g))(not (ontable a))(ontable b)(ontable c)
        (ontable d)(not (ontable e))(ontable f)(ontable g)(test1))
    :effect (and (test2)(not (test1))))
```

Example Plan

Assume the same example as before, and that action models for pickup, putdown and unstack are already known.

```
;;;;;; Action headers in M
(pickup ?v1) (unstack ?v1 ?v2)
;;;;;; Plan trace
;;; Initial state observation
(clear B) (ontable A) (handempty)
(on B A)(not (clear A))(not (ontable B))
(not (holding A))(not (holding B))
(not (on A A))(not (on A B))(not (on B B))
;;; Action observation
(putdown B)
;;; Action observation
(stack A B)
;;; Final state observation
(clear A) (on A B) (ontable B)
```

Output Plan

```
00 : (insert pre stack holding v1)

01 : (insert pre stack clear v2)
02 : (insert eff stack clear v1)
03 : (insert eff stack clear v2)
04 : (insert eff stack handempty)
05 : (insert eff stack holding v1)
06 : (insert eff stack on v1 v2)
07 : (apply unstack blockB blockA i1 i2)
08 : (apply putdown blockB i2 i3)
09 : (apply pickup blockA i3 i4)
10 : (apply stack blockA blockB i4 i5)
11 : (validate 1)
```

FAMA Code

Code Example

- https://github.com/daineto/meta-planning

- https://github.com/daineto/meta-planning/blob/master/notebooks/5.%20Model%20Learning.ipynb

Limitations of Learning from Passive Observations

- Susceptible to incorrect or incomplete model learning.
- E.g., if all packages are brown in color, a possible precondition will be that the package must be brown to unload them.
- Such methods don’t capture correct causal relationships.

Solution: Active/Online Acquisition of Observations

Active Model Learning

Deterministic and Stationary Setting

Input

- Predicates (User vocabulary)
  - With their evaluation functions
- List of capabilities.

Output

- PDDL-like description of each capability.

Siddharth Srivastava

Shashank R Marpally

Asking the Right Questions: Learning Interpretable Action Models Through Query Answering

Pulkit Verma, Shashank Rao Marpally, Siddharth Srivastava

In Proceedings of the Thirty-Fifth AAAI Conference on Artificial Intelligence, 2021 (CORE Ranking: A*)

Assumptions

- User's vocabulary matches simulator's vocabulary.
- Black-Box AI provides a list of capabilities.
- Stationary agent model.
- Deterministic environment.
- Fully observable setting.

Exponential Search for Learning Correct Description

- Consider the following 4 predicates/concepts:
  - (has_key)
  - (door_open)
  - (door_adjacent ?x)
  - (player_at ?x)
- Consider just one capability: (open-door ?x)
- $9^{|C|\times|P|} = 9^{1\times4}$=6561 possible models (Assuming deterministic models/ descriptions, i.e., no probabilities).

```
(:action open-door
  :parameters (?l1)
  :precondition (and
    (+/-/∅)(has_key)
    (+/-/∅)(door_open)
    (+/-/∅)(door_adjacent ?l1)
    (+/-/∅)(player_at ?l1))
  :effect (and
    (+/-/∅)(has_key)
    (+/-/∅)(door_open)
    (+/-/∅)(door_adjacent ?l1)
    (+/-/∅)(player_at ?l1))
```

Simple Queries

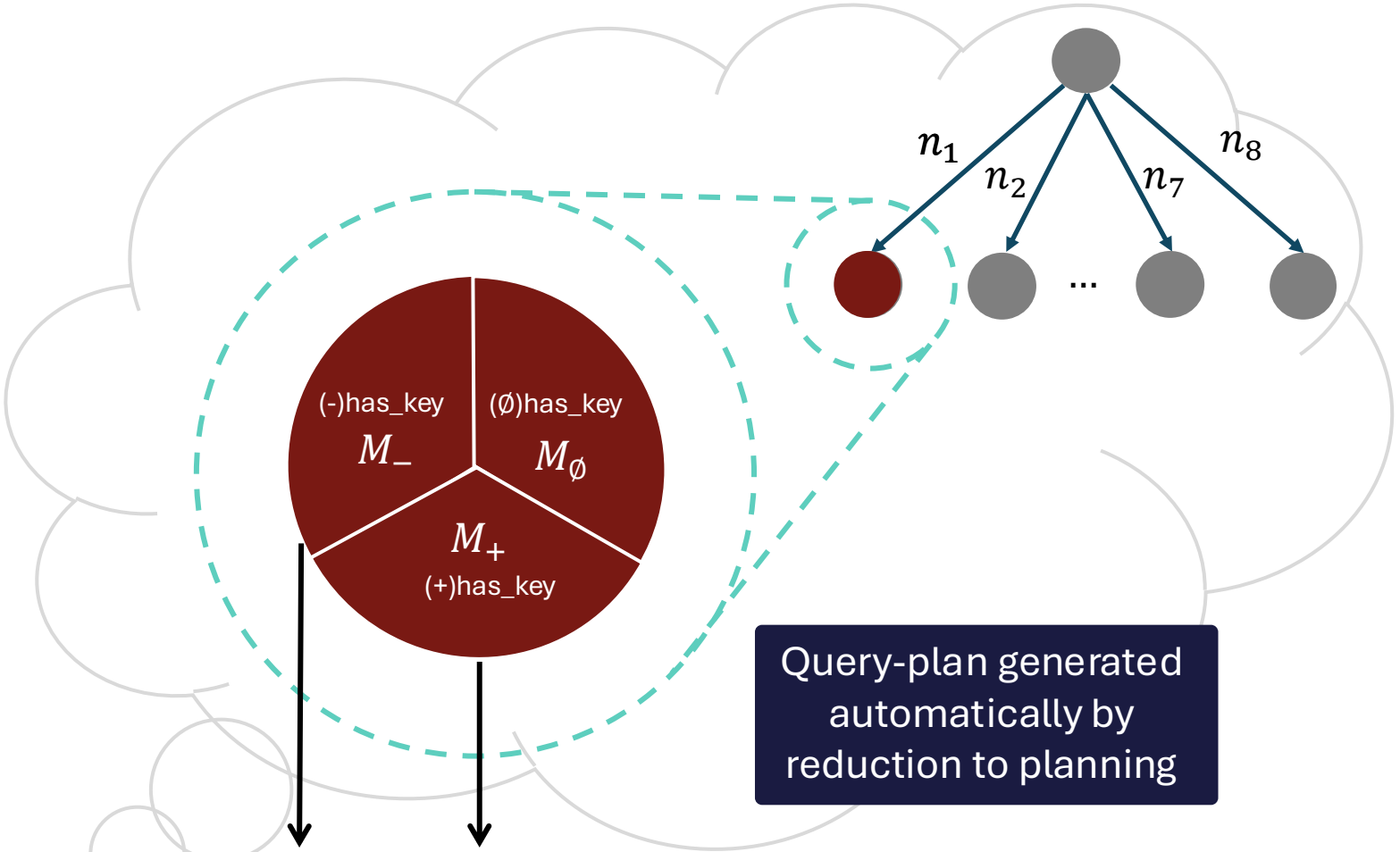
Query	In state s_I, what will happen if you execute the plan $\pi = \langle c_1, \ldots, c_n \rangle$?
Response	I can execute first ℓ steps of the plan, ending up in state s_F.

Plan Outcome Queries

- How to generate the queries?
- How to use the responses to generate models?

We have a reduction that converts this to a planning problem, so it automatically generates queries.

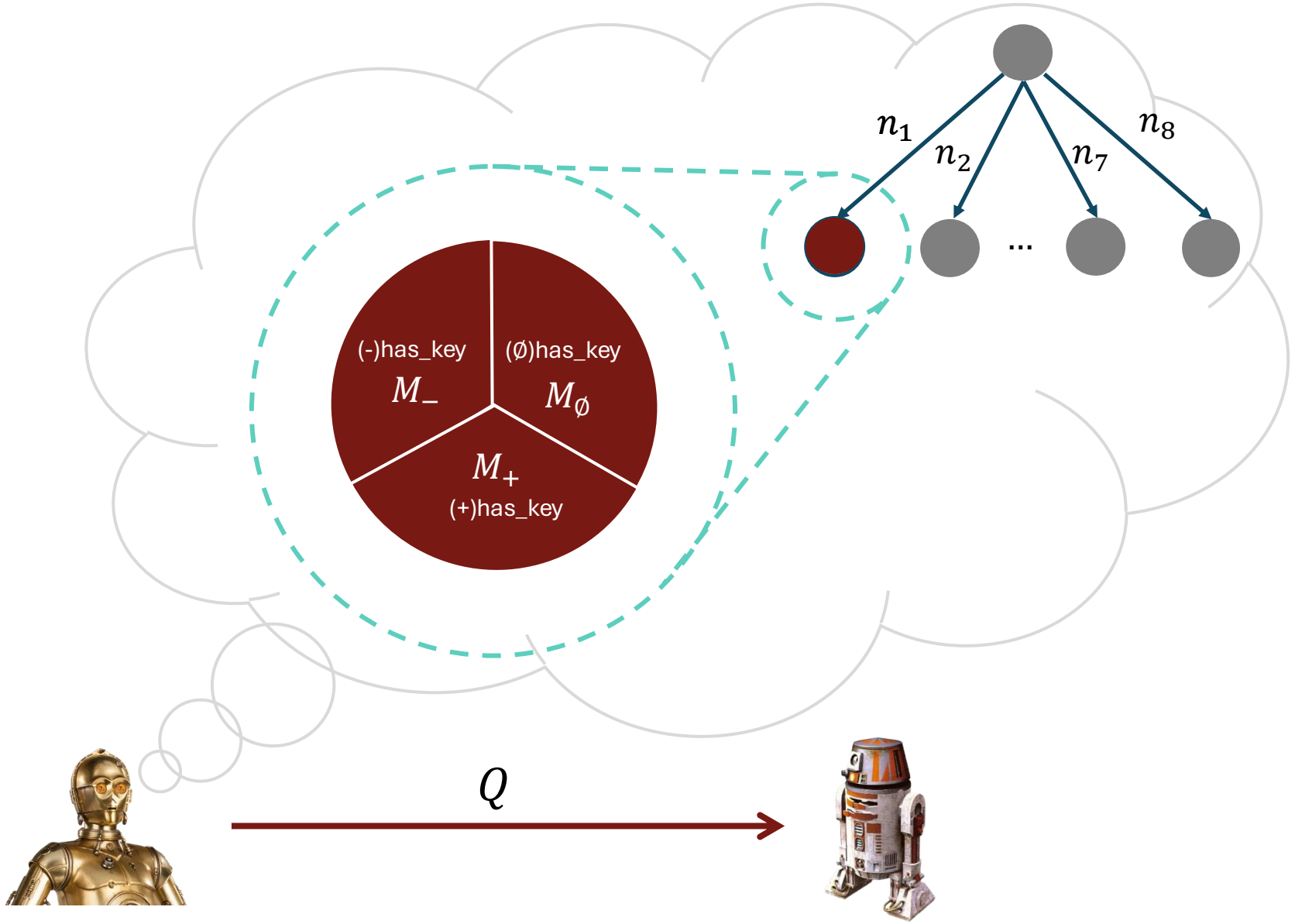
Hierarchical Query Synthesis



```
(:action open-door
  :parameters (?l1)
  :precondition (and
n1 (+/-/∅)(has_key)
n2 (+/-/∅)(door_open)
n3 (+/-/∅)(door_adjacent ?l1)
n4 (+/-/∅)(player_at ?l1))
  :effect (and
n5 (+/-/∅)(has_key)
n6 (+/-/∅)(door_open)
n7 (+/-/∅)(door_adjacent ?l1)
n8 (+/-/∅)(player_at ?l1))
```

[**Verma**, Marpally, Srivastava; AAAI '21]

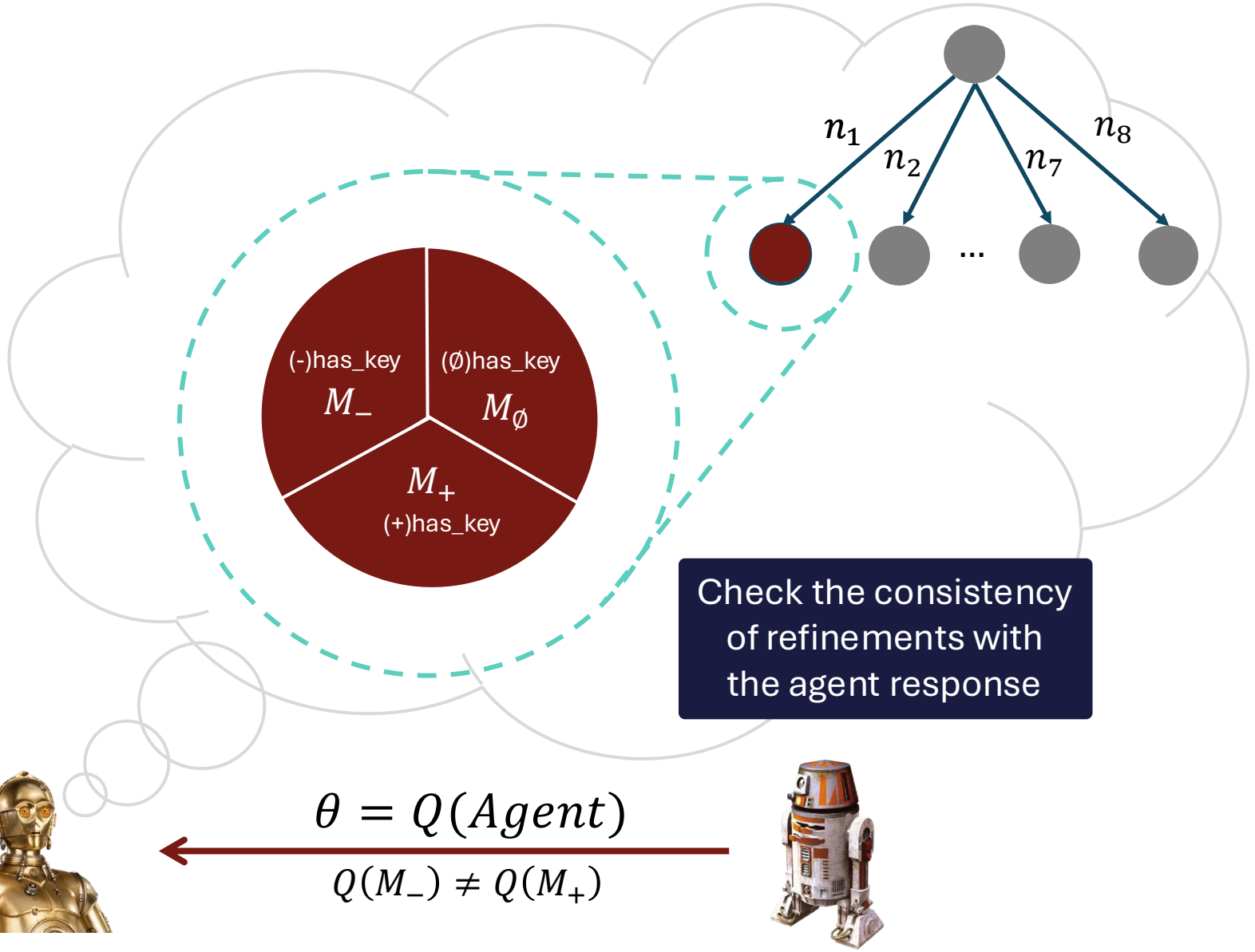
Hierarchical Query Synthesis




```
(:action open-door
  :parameters (?l1)
  :precondition (and
n1 (+/-/∅)(has_key)
n2 (+/-/∅)(door_open)
n3 (+/-/∅)(door_adjacent ?l1)
n4 (+/-/∅)(player_at ?l1))
  :effect (and
n5 (+/-/∅)(has_key)
n6 (+/-/∅)(door_open)
n7 (+/-/∅)(door_adjacent ?l1)
n8 (+/-/∅)(player_at ?l1))
```

[**Verma**, Marpally, Srivastava; AAAI '21]

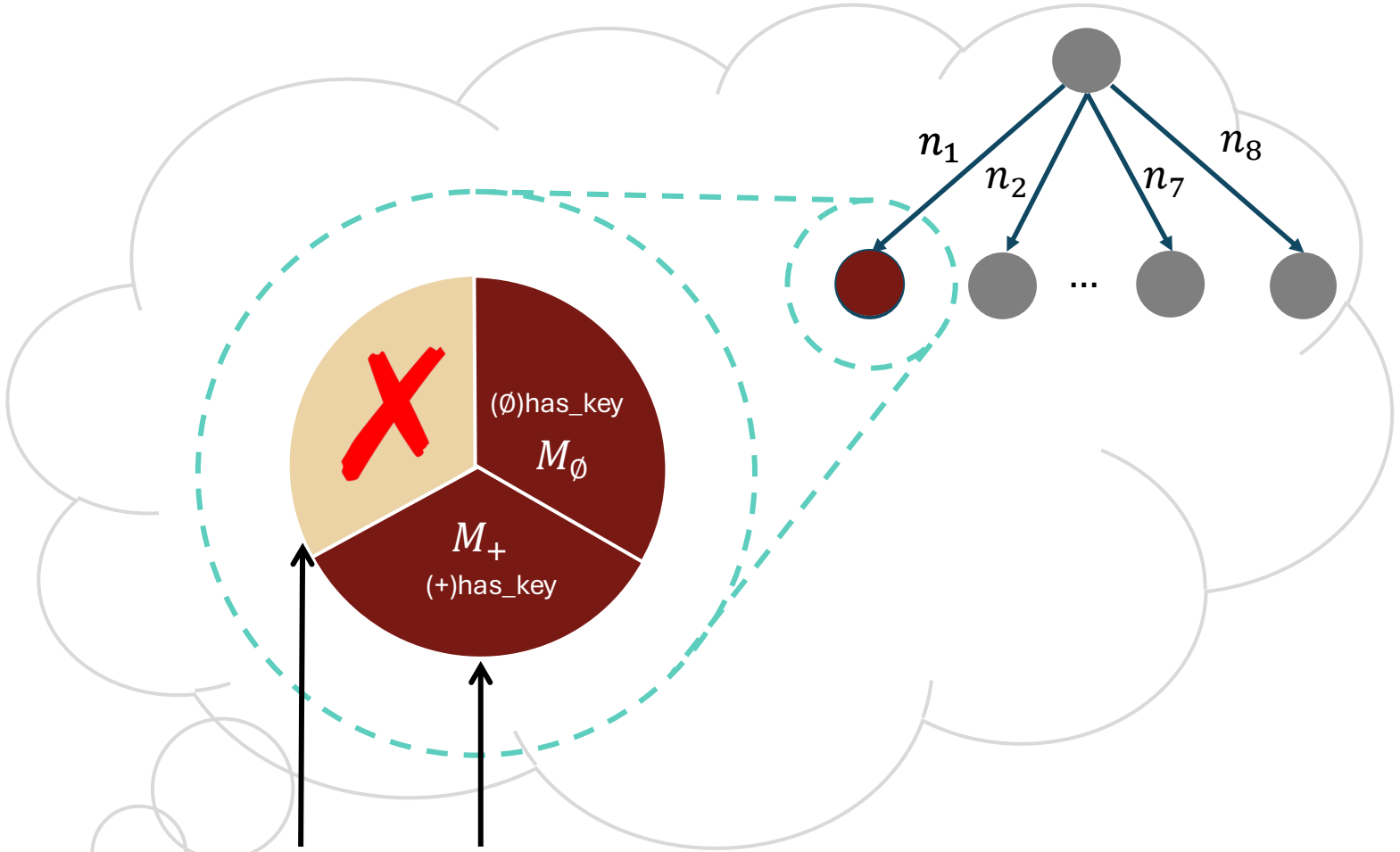
Hierarchical Query Synthesis




```
(:action open-door
  :parameters (?l1)
  :precondition (and
n1 (+/-/Ø)(has_key)
n2 (+/-/Ø)(door_open)
n3 (+/-/Ø)(door_adjacent ?l1)
n4 (+/-/Ø)(player_at ?l1))
  :effect (and
n5 (+/-/Ø)(has_key)
n6 (+/-/Ø)(door_open)
n7 (+/-/Ø)(door_adjacent ?l1)
n8 (+/-/Ø)(player_at ?l1))
```

[**Verma**, Marpally, Srivastava; AAAI '21]

Hierarchical Query Synthesis

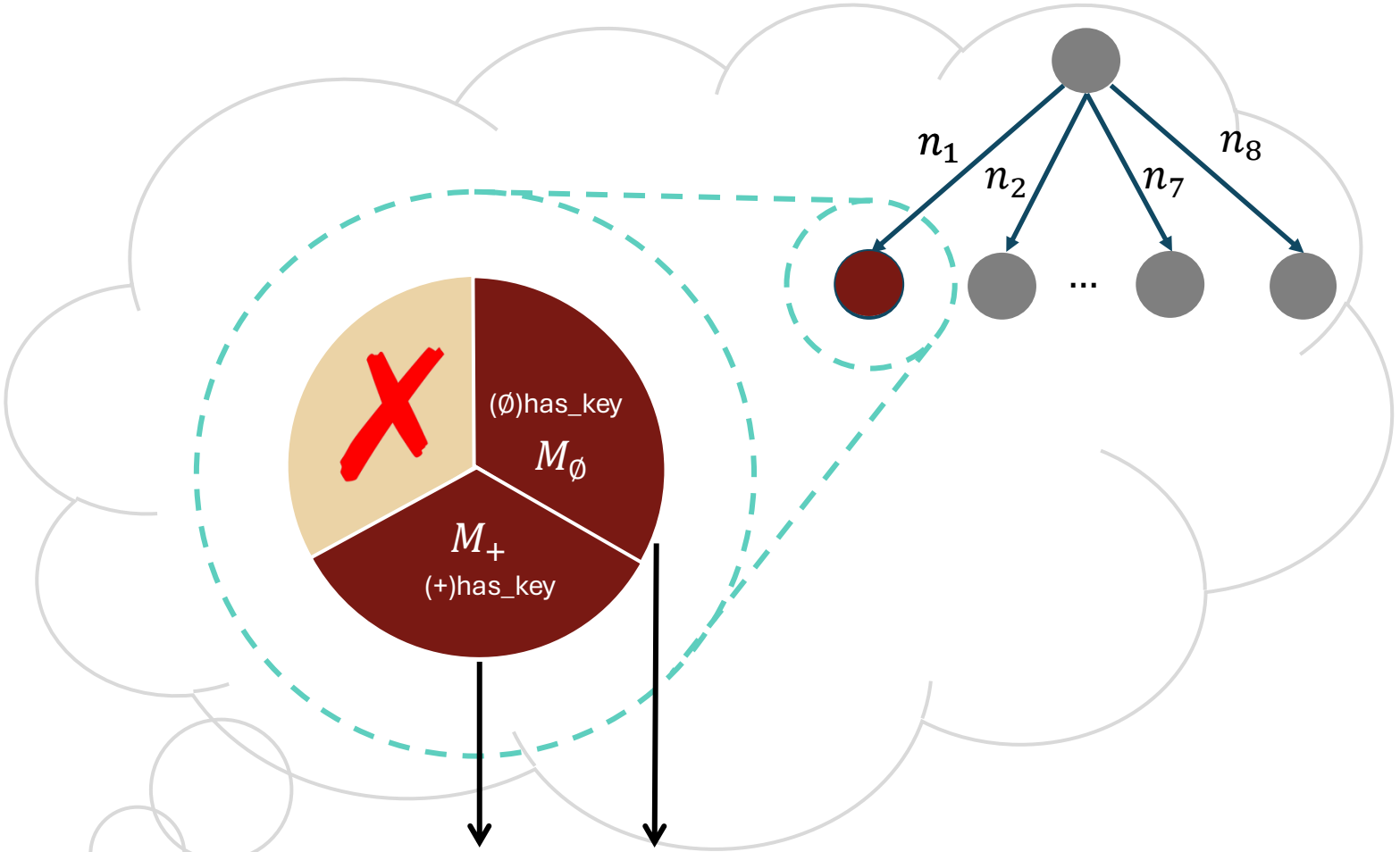



```
(:action open-door
  :parameters (?l1)
  :precondition (and
n1 (+/-/∅)(has_key)
n2 (+/-/∅)(door_open)
n3 (+/-/∅)(door_adjacent ?l1)
n4 (+/-/∅)(player_at ?l1))
  :effect (and
n5 (+/-/∅)(has_key)
n6 (+/-/∅)(door_open)
n7 (+/-/∅)(door_adjacent ?l1)
n8 (+/-/∅)(player_at ?l1))
```

Reject abstract model(s) that are not consistent with the agent

[**Verma**, Marpally, Srivastava; AAAI '21]

Hierarchical Query Synthesis

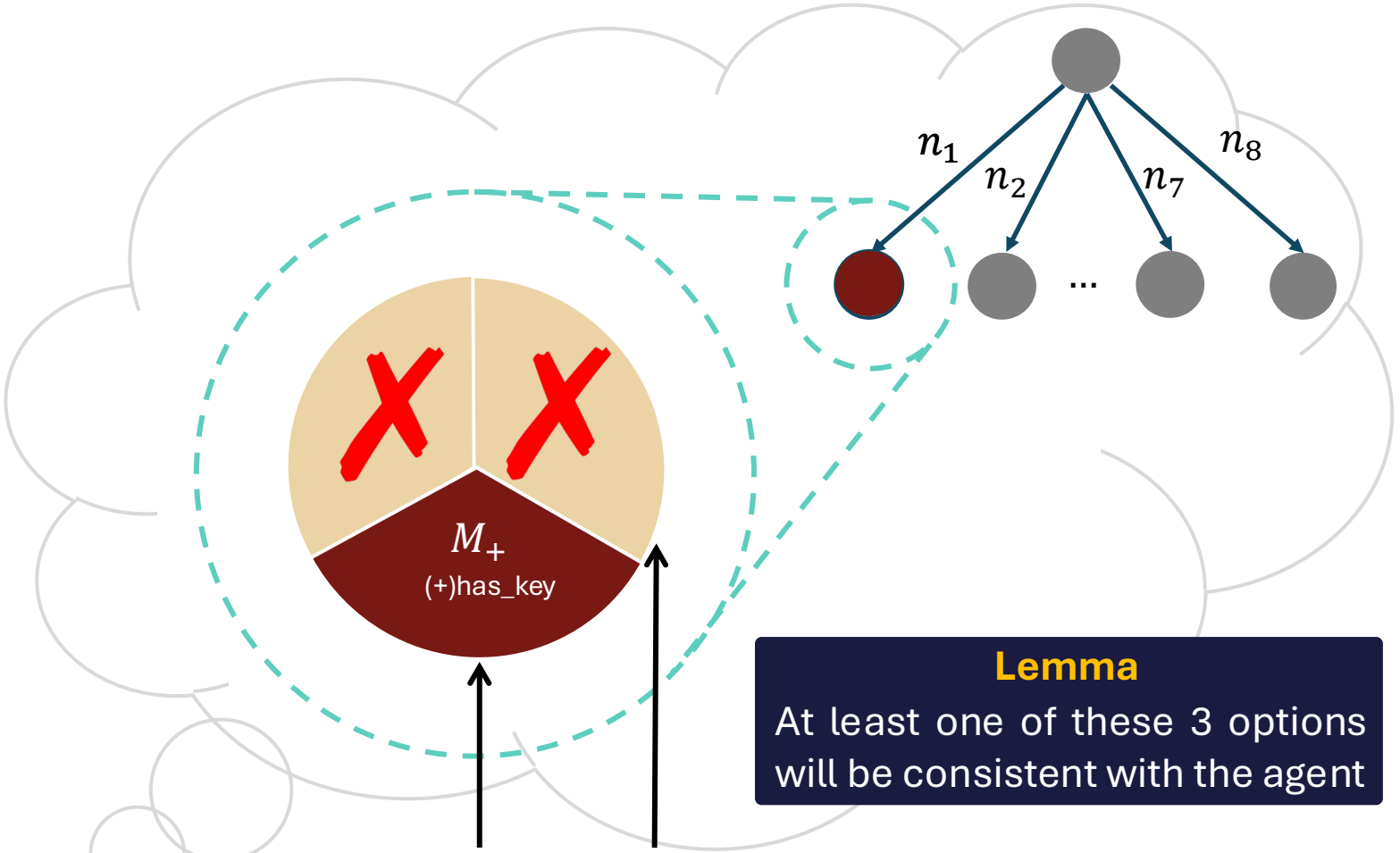



```
(:action open-door
  :parameters (?l1)
  :precondition (and
n1 (+/-/∅)(has_key)
n2 (+/-/∅)(door_open)
n3 (+/-/∅)(door_adjacent ?l1)
n4 (+/-/∅)(player_at ?l1))
  :effect (and
n5 (+/-/∅)(has_key)
n6 (+/-/∅)(door_open)
n7 (+/-/∅)(door_adjacent ?l1)
n8 (+/-/∅)(player_at ?l1))
```

Generate a distinguishing query for these two abstract models

[**Verma**, Marpally, Srivastava; AAAI '21]

Hierarchical Query Synthesis



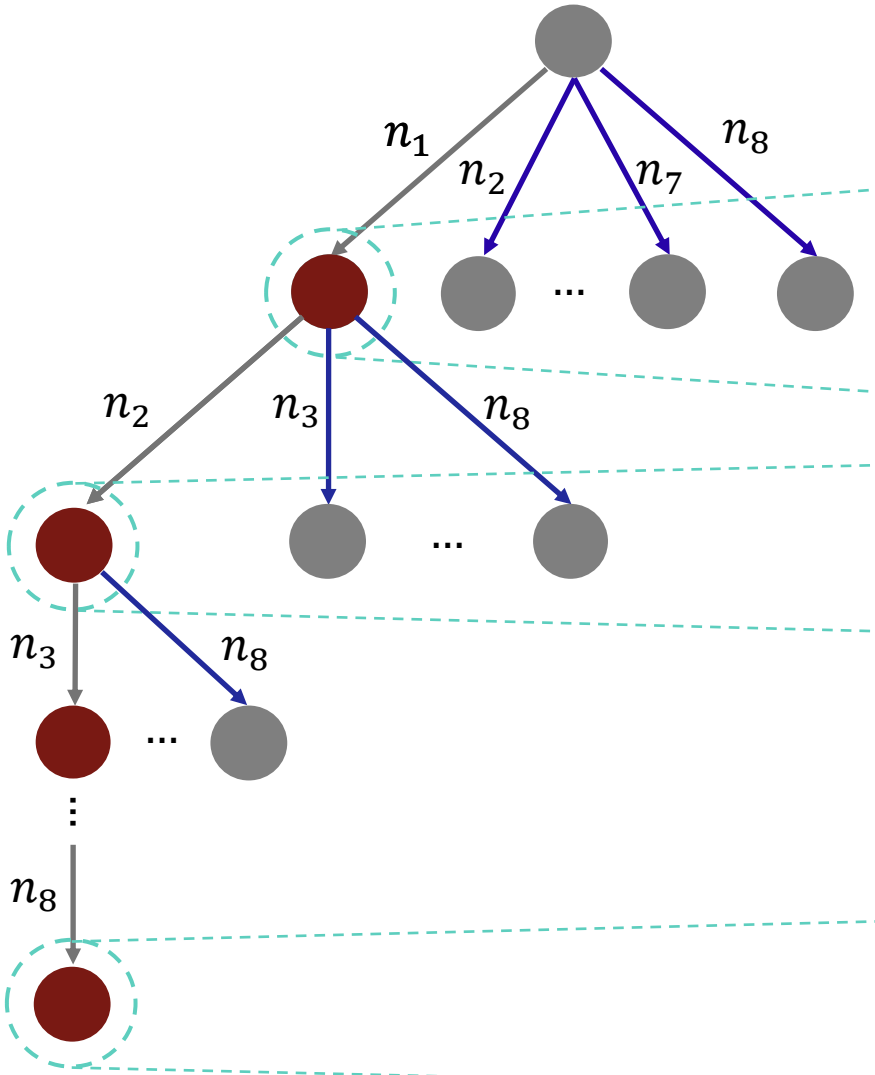
Lemma
At least one of these 3 options will be consistent with the agent

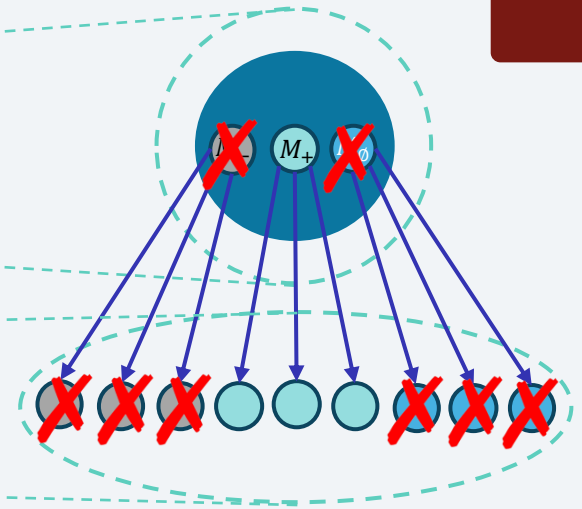
```
(:action open-door
  :parameters (?l1)
  :precondition (and
n1      (+)(has_key)
n2   (+/-/∅)(door_open)
n3   (+/-/∅)(door_adjacent ?l1)
n4   (+/-/∅)(player_at ?l1))
  :effect (and
n5   (+/-/∅)(has_key)
n6   (+/-/∅)(door_open)
n7   (+/-/∅)(door_adjacent ?l1)
n8   (+/-/∅)(player_at ?l1))
```

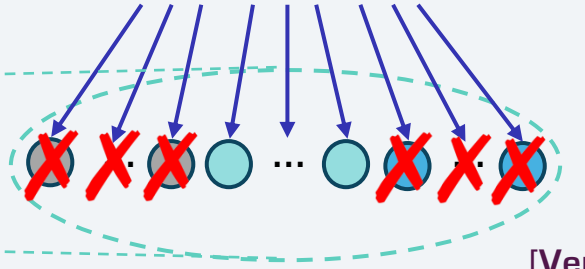
Reject the abstract model
that is not consistent with the agent

[**Verma**, Marpally, Srivastava; AAAI '21]

Hierarchical Query Synthesis







Key feature of the algorithm

Whenever we prune an abstract model, we prune a large number of concrete models.

Active Learning

[**Verma**, Marpally, Srivastava; AAAI '21]