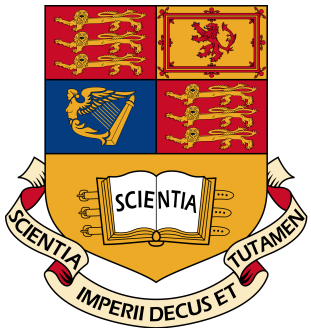

DEPARTMENT OF COMPUTING

IMPERIAL COLLEGE OF SCIENCE, TECHNOLOGY AND MEDICINE

H-PERL: The Hybrid Property, Event and Relation Learner

Author:
Ross Irwin

Supervisors:
Prof. Alessandra Russo
Dr. Krysia Broda
Dr. Jorge Lobo

June 6, 2020

Chapter 1

Hardcoded Model

Our first H-PERL implementation is the 'hardcoded' model. The hardcoded model makes use of a mixture of manually engineered components and components which are trained on the full-data version of the OceanQA dataset. This model should not be taken as a solution to the general VideoQA problem, since it would be labourious to rewrite components for each new dataset environment. Instead we intend this model to be used as a benchmark for the OceanQA dataset, against which other VideoQA implementations can be evaluated.

Full details on the performance of the model, as well as the performance of some of the individual components is given in Chapter **??**.

1.1 Properties

As mentioned in Chapter **??**, the role of the properties component is to take an image of an object (as produced by the object detector) and, in the case of the OceanQA dataset, to return the colour and rotation of the object.

Since the object image is a 3D tensor[1] of raw pixel values, a convolutional neural network is an excellent candidate for the implementation of the properties component. Other computer vision techniques for machine learning such as decision trees, random forests and SVMs require thousands of manually engineered filters to be applied to the image, whereas convolutional networks can learn a much smaller set of filters based on the training data.

Figure 1.1 shows the architecture of the properties network. The first part of this network encodes the object image into a 1024 dimenional latent vector using a series of convolutional and fully-connected layers. A set of fully-connected layers, one for each property, each take the vector encoding of the object and produce a vector of real numbers. The size of each vector is equal to the number of possible values that property can take.

[1] Height, width and RGB colour channels make up the three dimensions

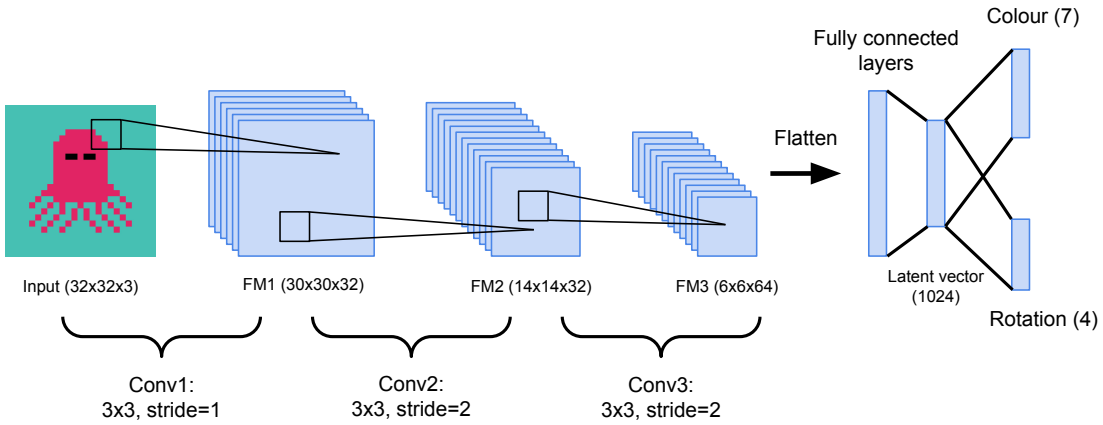

Figure 1.1: Architecture of the properties component neural network. An object is encoded into a latent vector, before a set of fully-connected layers produce a set of probability distributions over the property values. Batch normalisation is applied between convolutional layers. FM stands for feature maps.

The final output of the network is a set of probability distributions, one for each property, over the set of possible values that property can take. As is standard in a multiclass classification problem, the softmax function is applied to each set of property values in order to construct the probability distribution. The softmax function guarantees that the elements of a vector sum to one and that each element is greater than or equal to zero, hence softmax creates a discrete probability distribution. The softmax function for a vector $\mathbf{z} \in \mathbb{R}^K$ is as follows:

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \text{ for } i = 1, ..., K \tag{1.1}$$

The full-data version of the OceanQA dataset labels both the colour and rotation of every object in every frame. This makes training a neural network relatively straightforward; we collate all of the objects and their properties in the dataset, resize each object to 32x32 pixels, convert each property into a one-hot encoded vector and train the network using batches of 256 objects with a learning rate of 0.001 for 2. The cross-entropy loss is calculated for each property and these are summed to give an overall loss. The cross-entropy loss between a predicted probability distribution vector $\mathbf{p} \in \mathbb{R}^K$ and a one-hot encoded classification vector $\mathbf{y} \in \mathbb{R}^K$, where K is the number of classes, is as follows:

$$H(\mathbf{p}, \mathbf{y}) = -\sum_{c=1}^{K} y_c \log(p_c) \tag{1.2}$$

When the H-PERL model is being evaluated each object in the video is applied to the network (batched together for efficiency) and a set of probability distributions is produced. For each object, the property value for a particular property is given by the most probable element of the vector.

1.2 Relations

The job of the relations component is to list all instances of relevant binary relations between objects in each frame of the video. Since there is only one relevant binary relation in the OceanQA dataset, the job of this component is simply to list the instances of the *close* relation.

The relations component of the hardcoded model contains a hand-written binary classification algorithm for determining whether two objects are close or not. For a given video, this algorithm is applied to every pair of objects in every frame of the video in order to list all instances of the relation. The object arguments of the closeness algorithm are given in symbolic form, rather than as raw pixel matrices. This means the algorithm is heavily reliant on accurate information from the object detector - the position tuple in particular. Although the closeness algorithm doesn't make use of the property component's extracted features, other algorithms for determining binary relations may require these.

The algorithm for determining the closeness of two objects is, in fact, identical to the algorithm used when constructing the dataset. The algorithm uses the idea of an expanded box around each object; the objects are close if their boxes overlap, as can be seen in Figure 1.2. The algorithm was discussed alongside techniques used to create the dataset in Chapter **??** and is shown in Algorithm **??**.

Clearly, since the algorithm used to construct the dataset and the algorithm used to find the relations between objects are exactly the same, the relations component achieves perfect accuracy provided the object detection is exactly correct. Although this may seem like cheating, since in general it is not possible to know the underlying rules of the dataset, we reiterate that the hardcoded model is not a solution to VideoQA tasks in general, but rather is specific to the OceanQA dataset, and can be used for comparisons with other models.

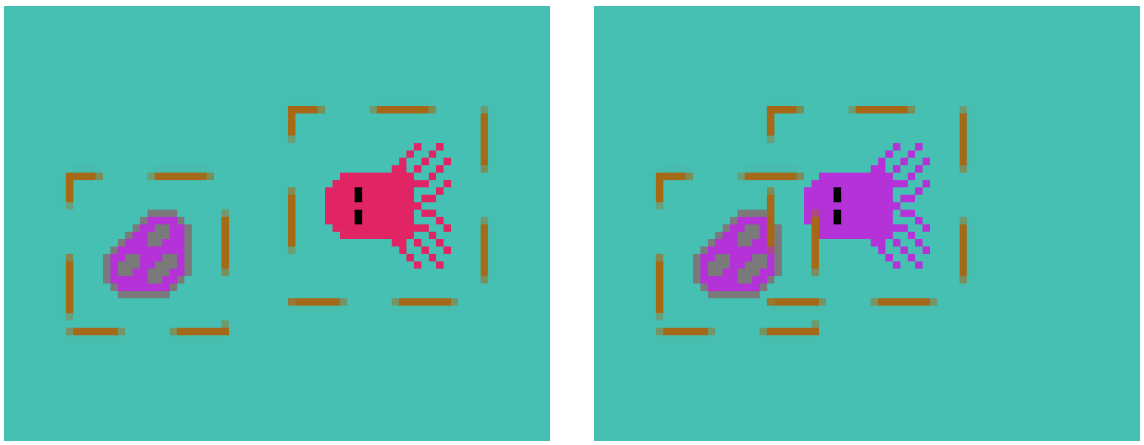

Figure 1.2: Diagrams showing the octopus before and after moving close to a purple rock, and therefore turning purple itself. The brown dashed lines show the bounding boxes expanded by 5 pixels on each side around the objects. If these boxes overlap the objects are deemed to be close to one another.

Another consideration for this approach to relation classification is speed; each relation classification algorithm (of which there is only one in the OceanQA dataset, but, in general there may be many) looks at every possible pair of objects in every frame of the video. Assuming that each relation classifier operates in constant time, this creates an overall algorithmic complexity of $\mathcal{O}(kmn^2)$, where k is the number of frames per video, m is the number of relations to be classified and n is the number of objects in each frame. Despite this we found that the time taken by the relation component was small relative to other components in the H-PERL model. Chapter **??** outlined the full details of the component's performance, including details on the time taken during evaluation.

1.3 Events

As mentioned in Chapter **??**, the role of the event detection component is to list all of the actions and effects which take place between each pair of consecutive frames of a given video. Since preceeding components have extracted a number of object and frame-level symbolic features already, the event detector in the hardcoded model works only with these features, as opposed to viewing the raw pixels in each frame.

For the hardcoded model, the event detector is implemented by, firstly, searching through all possible combinations of actions. The component assumes that there is only one action per frame and that only non-static objects can be the cause of an action. Each combination of actions is then applied to a hand-written $\mathcal{AL}$ model of the environment to generate the set of symbolic features which would be expected to be observed should the set of actions have occurred. This set of features is then compared to the set of features which were actually observed - the set of features extracted by preceeding components. Finally, the combination of actions which lead to the best fit with the observed data are selected.

For each set of possible actions and their associated features generated by the $\mathcal{AL}$ model, a set of manually engineered ASP rules is used to find the corresponding set of effects. For example, if the octopus moves[2] during frame i and is close to a blue rock during frame $i+1$, then we can infer that the *change colour* effect occurs during frame i.

1.3.1 The $\mathcal{AL}$ Model

As described in Chapter **??**, $\mathcal{AL}$ is a formal model for describing the behaviour of a dynamic system. An $\mathcal{AL}$ system consists of a set of states and some way of transitioning between states using actions. We model our video as an $\mathcal{AL}$ system by considering each frame as a state and object actions between frames as $\mathcal{AL}$ actions. In $\mathcal{AL}$ a state is described by a set of fluents. Each fluent, $<f>$, can be

[2]Moving during frame i refers to an object moving between frame i and frame $i+1$.

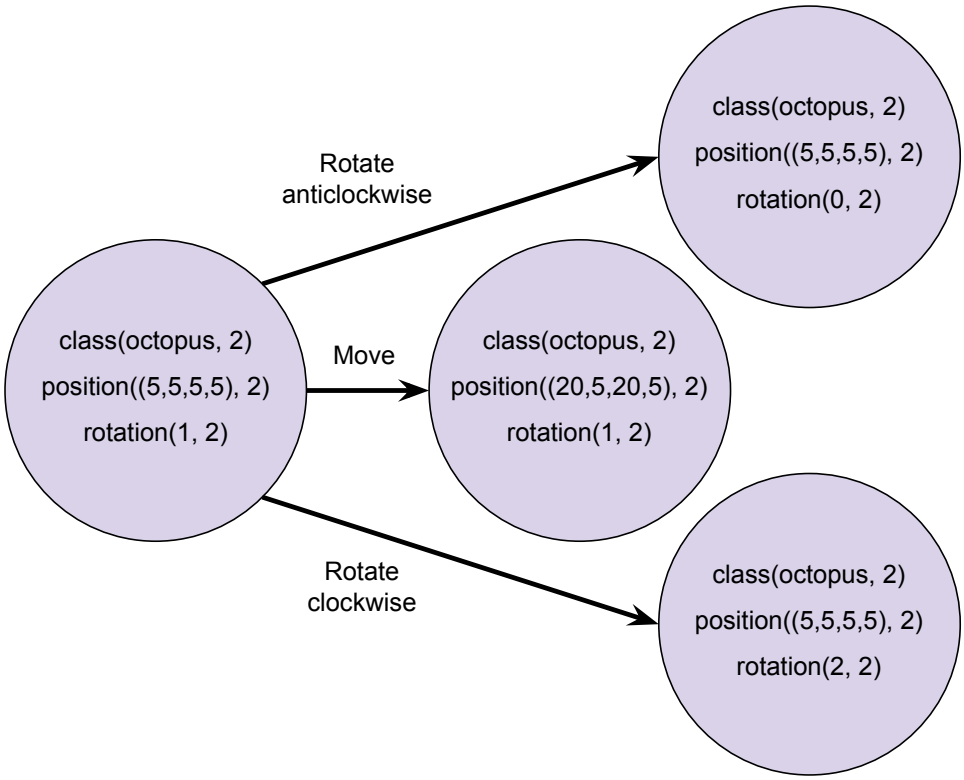

Figure 1.3: A simplified example set of $\mathcal{AL}$ states and transitions. States which model the OceanQA environment contain many more fluents than are shown here.

wrapped up in a predicate, $holds(<f>, <i>)$, where $<i>$ is a timestep in the video, which means that fluent $<f>$ is true at step $<i>$. Conversely, $\neg holds(<f>, <i>)$ means that $<f>$ is not true at step $<i>$. An OceanQA-specific example of a set of $\mathcal{AL}$ states and transitions between them is shown in Figure 1.3.

As mentioned above, the optimal set of actions is found by comparing the observed features of the environment with an internal $\mathcal{AL}$ model of the environment. In the rest of this section we outline the $\mathcal{AL}$ system description for the OceanQA environment. This system description can then be written in ASP using the encoding provided in Appendix A.

Firstly, we outline the types involved in the $\mathcal{AL}$ system description, which are as follows:

- Properties, including class and position, are modelled as *inertial fluents* - their values can change as a direct result of the actions taken. Notice, however, that the number of possible position values is very large - 256^4, hence, rather than allowing all possible values, we only include values that have been observed in the video, all other values are not modelled. This applied to all properties, class and position fluents.
- An additional inertial fluent, $exists(<id>)$, is also required. Intuitively, it means that an object with identifier $<id>$ is present in the current timestep.

- The close relation, which is written in ASP as $close(Id1, Id2)$ when an object with identifier $Id1$ is close to an object with identifier $Id2$, is considered a *defined fluent*. Although they are modelled as defined fluents, their thruthiness cannot altered by the events component, since relation classification is handled by the relations component. Therefore, these fluents are copied directly across from the observed data into the $\mathcal{AL}$ model.

- We add a further defined fluent, which also comes directly from the observed information. This fluent is called $disappear(<id>)$, and, naturally, means that an object with identifier $<id>$ disappears immediately after the current timestep.

- Actions are, of course, modelled by the *action* type. The action itself takes a single argument - the object's identifier. In ASP each action is written as $<action>(<id>)$.

The domain independent rules for the OceanQA environment are as outlined in Appendix A, with the following addition:

$$\neg holds(F, 0) \text{ :- } fluent(inertial, F), not\ holds(F, 0). \tag{1.3}$$

This rule ensures that the initial state of the system is complete for inertial fluents - all inertial fluents are either true or false. While this isn't a requirement for $\mathcal{AL}$ systems, it simplifies the rest of the model since there is no uncertainty about the system's state.

Finally, we outline the domain dependent $\mathcal{AL}$ statements for the OceanQA environment:

- The *move* action causes the object to move 15 pixels in the direction of rotation. This means that, in frame $i + 1$, the object is no longer in the position it was in during frame i. We use a Python function, new_pos, in the ASP program to calculate the object's next position. This function is called using the @ symbol. The $\mathcal{AL}$ *causal laws* for the move action are the following:

$$move(Id) \textbf{ causes } position(@new\_pos(P, R), Id) \textbf{ if } position(P, Id), rotation(R, Id)$$
$$move(Id) \textbf{ causes } \neg position(position(P, Id)) \textbf{ if } position(P, Id)$$

- $\mathcal{AL}$ causal laws follow the same pattern for both rotation actions. The Python function, new_rot, is used to calculate the new rotation for the object, it takes the previous rotation and the type of rotation as arguments. In the following $\mathcal{AL}$ causal laws $<d>$ is used to refer to the direction of rotation:

$$rotate\_<d>(Id) \textbf{ causes } rotation(@new\_rot(R, rotate\_<d>) \textbf{ if } rotation(R, Id)$$
$$rotate\_<d>(Id) \textbf{ causes } \neg rotation(R, Id) \textbf{ if } rotation(R, Id)$$

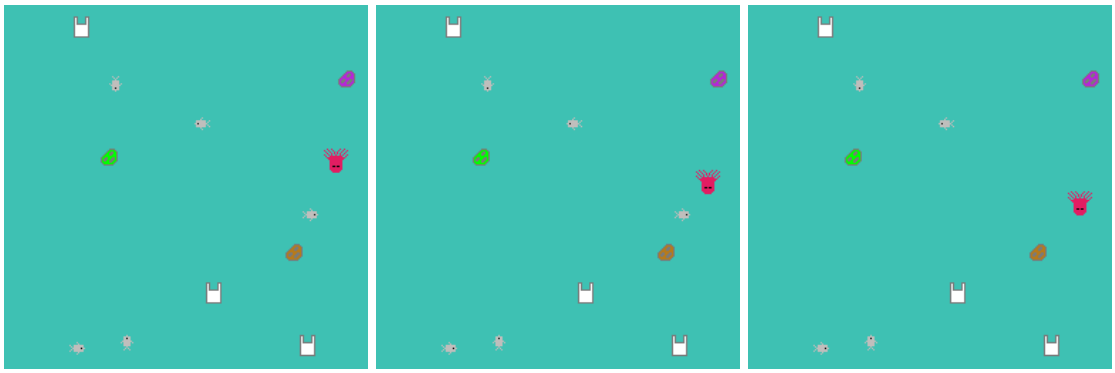

Figure 1.4: An example of the *eat a fish* effect.

- In order to ensure that objects are modelled as non-existent after they have disappeared, the following causal law is added:

$$move(Id) \textbf{ causes } \neg exists(Id) \textbf{ if } exists(Id), disappear(Id)$$

- Finally, the following *executability conditions* are added to reduce the size of the search space for the optimiser:

$$\textbf{impossible } move(Id) \textbf{ if } \neg exists(Id)$$
$$\textbf{impossible } rotate\_clockwise(Id) \textbf{ if } \neg exists(Id)$$
$$\textbf{impossible } rotate\_anticlockwise(Id) \textbf{ if } \neg exists(Id)$$

The use of the $exists(<id>)$ fluent is an intuitive way of encoding the knowledge that when an object disappears it no longer exists, and, if an object does not exist, it cannot participate in any actions. Without this predicate, encoding knowledge like this would be tricky because the $disappear(<id>)$ fluent appears (at most) once during the video.

In the remainder of this section we diverge from $\mathcal{AL}$ orthodoxy in order to model effects. Effects are defined by ASP rules with a combination of actions and $\mathcal{AL}$ fluents in the body. These rules can mostly be considered to observe the $\mathcal{AL}$ model, rather than affect it. However, an exception is made for the rules which update the octopus' colour.

Firstly, for the *eat a fish* (an example of which is shown in Figure 1.4) and *eat a bag* effects, the following ASP rules are used:

$$\begin{aligned} occurs\_effect(eat\_a\_fish(Octo), I) \text{ :- } & occurs\_action(move(Octo), I), \\ & holds(class(fish, Fish), I), \\ & holds(disappear(Fish), I). \end{aligned} \tag{1.4}$$

$$\begin{aligned} occurs\_effect(eat\_a\_bag(Octo), I) \text{ :- } & occurs\_action(move(Octo), I), \\ & holds(class(bag, Bag), I), \\ & holds(disappear(Bag), I), \\ & holds(disappear(Octo), I). \end{aligned} \tag{1.5}$$

We also need to ensure the octopus' colour is updated when it comes close to rock. To do this we use the following helper predicate:

$$\begin{aligned} change\_colour(Old, New, Id, I-1) \text{ :- } & holds(class(octopus, Id), I), \\ & holds(close(Id, IdRock), I), \\ & holds(colour(Old, Id), I-1), \\ & holds(class(rock, IdRock), I), \\ & holds(colour(New, IdRock), I), \\ & Old\,!= New, step(I-1). \end{aligned} \tag{1.6}$$

Finally, the octopus' colour is updated using the follwing rules:

$$holds(colour(New, Id), I+1) \text{ :- } change\_colour(Old, New, Id, I). \tag{1.7}$$

$$\neg holds(colour(Old, Id), I+1) \text{ :- } change\_colour(Old, New, Id, I). \tag{1.8}$$

Rules 1.7 and 1.8 alter the $\mathcal{AL}$ state, despite not being generated from $\mathcal{AL}$ statements. There are other $\mathcal{AL}$ models for the OceanQA environment (not discussed here) which do not suffer from the same problem. However, these models can be more complex, which leads to higher probability of human error. Instead we prefer to make a slight adaption to the $\mathcal{AL}$ rules in order to allow a simpler model to be defined.

1.3.2 Action Optimisation

So far the $\mathcal{AL}$ model has used the $holds(<f>, <i>)$ predicate to store its information, but the observed data uses $obs(<f>, <i>)$. This distinction is by design; it separates the information (particularly that which is modelled by inertial fluents) in the two data models. However, we need some way of comparing the two models in order to optimise the action combination.

Firstly, in order to ensure that both data models start with the same information, the fluents for the initial frame in the observed data are copied over to the $\mathcal{AL}$ model. Secondly, the ASP program generates all possible action combinations and applies each combination to the $\mathcal{AL}$ model to find the set of actions which best fits the observed data. These sets of action can be generated in ASP using the following choice rule:

$$occurs\_action(A, I) : action(A) : -step(I+1), I >= 0. \tag{1.9}$$

1.4 Error Correction

Bibliography

[1] Michael Gelfond and Yulia Kahl. *Knowledge representation, reasoning, and the design of intelligent agents: The answer-set programming approach*. Cambridge University Press, 2014.

Appendix A

ASP encoding of $\mathcal{AL}$

In this appendix we present a method for encoding an $\mathcal{AL}$ system description in ASP, as described in [1]. We need to encode three different parts of the system description: the signature, the $\mathcal{AL}$ statements and some domain-independent axioms.

We encode the signature of the system description, $sig(SD)$, as follows:

- For each constant symbol c of sort $sort\_name$ other than $fluent$, $static$ or $action$, $sig(SD)$ contains

$$sort\_name(c). \tag{A.1}$$

- For every static g of SD, $sig(SD)$ contains

$$static(g). \tag{A.2}$$

- For every intertial fluent f of SD, $sig(SD)$ contains

$$fluent(inertial, f). \tag{A.3}$$

- For every defined fluent f of SD, $sig(SD)$ contains

$$fluent(defined, f). \tag{A.4}$$

- For every action a of SD, $sig(SD)$ contains

$$action(a). \tag{A.5}$$

In the following we refer to the ASP encoding of the $\mathcal{AL}$ system description as $\Pi(SD)$, where $\Pi(SD)$ includes $sig(SD)$. We introduce a relation $holds(f, i)$ which says that fluent f is true at timepoint i. We also introduce the notation $h(l, i)$ where l is a domain literal and i is a step, which will not be used in the ASP program, but will instead be replaced by either $holds(f, i)$ if $l = f$, or by $\neg holds(f, i)$ if $l = \neg f$.

We encode the $\mathcal{AL}$ statements as follows:

- If the maximum number of steps is $< max >$, then $\Pi(SD)$ includes

$$\#const\ n = \ < max > . \tag{A.6}$$
$$step(0..n). \tag{A.7}$$

- For every causal law, a **causes** l **if** $p_0, ..., p_m$ $\Pi(SD)$ contains

$$h(l, I+1) \text{ :- } h(p_0, I), ..., h(p_m, I), occurs\_action(a, I), I < n. \tag{A.8}$$

- For every state constraint, l **if** $p_0, ..., p_m$ $\Pi(SD)$ contains

$$h(l, I) \text{ :- } h(p_0, I), ..., h(p_m, I). \tag{A.9}$$

- For every executability condition, **impossible** $a_0, ..., a_k$ **if** $p_0, ..., p_m$ $\Pi(SD)$ contains

$$\neg occurs\_action(a_0, I); ...; \neg occurs\_action(a_k, I) \text{ :- } h(p_0, I), ..., h(p_m, I). \tag{A.10}$$

The ; in the head of rule A.10 stands for logical disjunction and can be read as meaning at least one of $a_0, ..., a_k$ must not occur at timepoint I.

In order to complete our encoding of an $\mathcal{AL}$ system description we need to add a number of domain-independent axioms. These axioms are not specific to any system or task, but rather convey commonsense knowledge that should apply to many systems. It is worth noting, however, that in certain situations some or all of these axioms may not make sense and should not be used.

We encode the domain-independent knowledge as follows:

- The inertia axiom states that inertial fluents will keep their state unless explicitly changed:

$$\begin{aligned} holds(F, I+1) \text{ :- } & fluent(inertial, F), \\ & holds(F, I), \\ & not\ \neg holds(F, I+1), \\ & I < n. \end{aligned} \tag{A.11}$$

$$\begin{aligned} \neg holds(F, I+1) \text{ :- } & fluent(inertial, F), \\ & \neg holds(F, I), \\ & not\ holds(F, I+1), \\ & I < n. \end{aligned} \tag{A.12}$$

- The closed-world assumption (CWA) for defined fluents states that defined fluents which are not known to be true are assumed to be false.

$$\neg holds(F, I) \text{ :- } fluent(defined, F), step(I), not\ holds(F, I). \tag{A.13}$$

- The CWA for actions states that actions that are not known to occur are assumed to not occur.

$$\neg occurs\_action(A, I) \text{ :- } action(A), step(I), not\ occurs\_action(A, I). \quad \text{(A.14)}$$

Finally, in order to make use of the system description encoding we would include information on which events occured, using the $occurs\_action(A, I)$ predicate, and information on the different states of the system using $holds(F, I)$ and $\neg holds(F, I)$.

Briefcase Example

We now consider a simple example domain: a briefcase with two clasps (from [1]). There is a single action, toggle, which moves a given clasp into the up position if it is down, and vice-versa. If both clasp are up the briefcase is open, otherwise, it is closed.

The signature of the domain consists of sort clasp, which can take value 1 or 2, inertial fluent $up(C)$, defined fluent $open$ and action $toggle(C)$, where C is one of the clasps.

A schema for the system description is as follows:

$$toggle(C) \textbf{ causes } up(C) \textbf{ if } \neg up(C) \quad \text{(A.15)}$$
$$toggle(C) \textbf{ causes } \neg up(C) \textbf{ if } up(C) \quad \text{(A.16)}$$
$$open \textbf{ if } up(1), up(2) \quad \text{(A.17)}$$

This is known as a schema for a system description since it contains variables. Individual rules can be obtained by grounding the variables. So each of the first two schema rules would produce two ground rules, one where $C = 1$ and another where $C = 2$.

Following the procedure outlined above, we would encode this system description into the following ASP program:

```
% Possible timesteps
#const n = 1.
step(0..n).

% Signature
clasp(1).
clasp(2).

fluent(inertial, up(C)) :- clasp(C).
fluent(defined, open).
action(toggle(C)) :- clasp(C).
```

```
% Domain dependent rules
holds(up(C), I+1) :- occurs_action(toggle(C), I),
                     -holds(up(C), I),
                     I<n.

-holds(up(C), I+1) :- occurs_action(toggle(C), I),
                      holds(up(C), I),
                      I<n.

holds(open, I) :- holds(up(C), I), holds(up(2), I).

% Domain independent rules
holds(F, I+1) :- fluent(inertial, F),
                 holds(F, I),
                 not -holds(F, I+1),
                 I<n.

-holds(F, I+1) :- fluent(inertial, F),
                  -holds(F, I),
                  not holds(F, I+1),
                  I<n.

-holds(F, I) :- fluent(defined, F), step(I),
                not holds(F, I).

-occurs_action(A, I) :- action(A), step(I),
                        not occurs_action(A, I).
```

In a similar fashion to system description schemas, ASP programs allow variables. ASP solvers will first ground the program by computing the the *relevant grounding* - the set of ground rules which could be used by the program. The relevant grounding can be infinite so care needs to taken to ensure that it can be computed.

Appendix B

Dataset Examples

TODO