Traversing the Linguistic Divide: Aligning Semantically Equivalent Fluents Through Model Refinement*

Kelsey Sikes,[1] **Morgan Fine-Morris,** [2] **Sarath Sreedharan** [1] **Mark Roberts,** [2]

[1] Colorado State University, Fort Collins, Colorado, USA

[2] Navy Center for Applied Research in AI, Naval Research Laboratory, Washington, DC, USA

kelsey.sikes@colostate.edu, morgan.f.fine-morris.ctr@us.navy.mil, sarath.sreedharan@colostate.edu, mark.c.roberts20.civ@us.navy.mil

Abstract

Recent advances in natural language processing have resulted in the development of powerful semantic parsers that could, in theory, extract planning models from text. Despite their proficiency, these approaches remain poor at extracting model information from diverse sources, as they are unable to handle cases where the same entity may be referred to in different terms in different documents. In this work, we address this problem for the first time by applying meta-actions to a planning model to create bridges between semantically equivalent state variables with different labels. We then iteratively test and refine this model until it is able to produce a valid plan by correctly identifying the relationship between syntactically different fluents like this. Our approach is then evaluated on two standard IPC domains, where our methods are able to identify the correct plans.

Introduction

The recent developments in large language models have led to the development of powerful semantic parsers that could be used out of the box to extract planning model descriptions directly from natural language (Guan et al. 2023). However, we still see such systems failing in scenarios where information about the model is coming from heterogeneous sources.

Consider the task of extracting a planning model to capture the possible actions a cybersecurity attacker can take from a database of vulnerability descriptions, more commonly referred to as Common Vulnerabilities and Exposures (CVE) (Yosifova 2021; Santiago and Mendez 2023). From a conceptual point of view, this is a reasonably easy task; after all, each vulnerability here corresponds to a specific action the attacker can take, and the descriptions themselves are explicitly specified in terms of pre-conditions that need to hold for a vulnerability to be exploited and post-conditions (i.e. effects) resulting from that exploitation (and as such can be mapped over to the effects).

Once such planning models are extracted, they can be used to model attackers and perform analysis like pentesting (Bezawada, Ray, and Tiwary 2019). But different vulnerability authors might describe the same phenomena in different terms. Thus, one of the greatest strengths of such databases, namely the fact that they are collected from different sources, also becomes a weakness. For example, while one author might list a service being down as a post-condition for exploiting a vulnerability, another author might use the words service unavailable to describe the preconditions for a different vulnerability. A semantic parser might ascribe different propositions to each of these descriptions and, as such, result in an incorrect domain description.

Beyond cybersecurity, this issue applies to any application domain where a model may need to be learned from documents from different sources, including application areas for planning like web-service composition (Hoffmann 2015) and business process automation (Carman, Serafini, and Traverso 2003; Marrella and Chakraborti 2021). In each of these cases, one may end up with a domain with a set of semantically equivalent state variables, which are syntactically different in so far that they are provided with different labels. We will refer to such models as partial models in so far that it is missing information, particularly equivalence between the fluents.

Our work represents the first attempt at providing a systematic solution to working with domains with such state variables. In this work, we will focus on refining such a planning model (assumed to be given) through interaction with a simulator until it can generate a valid plan for a given problem instance. At the core of our approach lies the use of a set of meta-actions, referred to as 'replace' actions. These actions hypothesize potential equivalence between pairs of state variables. Plans generated from domains augmented with such replace actions are then tested on simulators for the domain, and the feedback received (in terms of what actions failed and what preconditions were unsatisfied) is then used to update the replace action list.

We demonstrate our approach's effectiveness on two IPC domains modified to contain semantically equivalent state variables with distinct labels. Our method identified valid plans in each case and uncovered correct fluent relationships.

Related Work

In previous work that looked at the mapping between state variables, most have focused on the relationship between variables from different models. Planning has a long history of utilizing state abstractions (e.g., (Bäckström and Jonsson 2013; Sacerdott 1973; Srivastava, Russell, and Pinto 2016)).

*Longer version can be found at: https://bit.ly/4aAJYnp

Here, a specific state space is mapped to a different, possibly smaller set. If we use a factored representation for the target, one could have mappings between state variable subsets from the source model to the target. In this vein, another set of related works is the problem of learning interpretable representations for inscrutable models (Sreedharan and Kambhampati 2021). The underlying assumption here is that the state variables in the learned representation correspond to information in the base model. However, given the inscrutable nature of the base model, the exact variables may not be known.

Most previous work on model learning for classical planning side-steps this issue, given their focus on learning from plans or plan traces (Wu, Yang, and Jiang 2007; Aineto, Jiménez Celorrio, and Onaindia 2019; Cresswell, Mccluskey, and West 2013). Many of these methods assume access to the set of state fluents upfront. While some of these assume access to intermediate states (cf. (Stern and Juba 2017)), the others assume access to only the initial and final state (Aineto, Jiménez Celorrio, and Onaindia 2019). Some methods try to generate state variables as part of the model learning process (Verma and Srivastava 2020; Guan et al. 2023). We start running into the problem described in the paper when considering learning models from a natural language description. While there are previous works on learning models from natural language descriptions (e.g., (Lindsay et al. 2017; Addis and Borrajo 2011; Guan et al. 2023), to the best of our knowledge, we are the first to identify and solve the specific problem of linking semantically equivalent fluents. It is also worth noting that our approach of refining models using simulators also makes it an instance of online model learning (Lamanna et al. 2021).

Running Example

As a motivating example, consider a robot that learned to cook from a set of recipes. Now, the robot is tasked with making hummus, whose recipe calls for the use of chickpeas. Searching through the kitchen cabinet it only finds a package of garbanzo beans. Now the robot can only achieve its goal if it knows that `(has-chickpeas)` are the same as `(has-garbanzo-beans)`. Now, our method will, in fact, try to hypothesize such potential equivalence and try out plans that are supported by them. In this case, the robot will find out that the equivalence holds, as using garbanzo beans still results in a tasty hummus.

Background

In this section, we introduce the planning formalisms used in this paper. A planning model is the tuple $\mathcal{M} = \langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle$, where $\mathcal{I}$ is an initial state and $\mathcal{G}$ is a goal description. $\mathcal{D}$ is then the domain associated with the model and is defined by $\mathcal{D} = \langle \mathcal{F}, \mathcal{A} \rangle$. $\mathcal{F}$ corresponds to a set of boolean variables that can uniquely describe any state space s for a given planning problem. An action $a \in \mathcal{A}$ is a tuple where $a = \langle pre_+(a), pre_-(a), add(a), del(a) \rangle$, which correspond to the positive or negative preconditions that must be satisfied for each action to be executed, while $add(a)$ and $del(a)$ are add and delete effects produced when an action is executed. A solution to this is a plan π, which is a sequence of actions that takes an agent from an initial state $\mathcal{I}$ to a goal state $\mathcal{G}$. Each $a \in \mathcal{A}$ has a cost c. The sum of these costs is the total cost of a plan.

Problem Formulation

To formalize the notion, we create a mapping function between two models. The ground truth model $\mathcal{M}^{\mathcal{T}} = \langle \mathcal{D}^{\mathcal{T}}, \mathcal{I}^{\mathcal{T}}, \mathcal{G}^{\mathcal{T}} \rangle$ represents the ideal situation where everything is known and all actions and states are accurately represented. Instead of the true model, the planner instead has access to a partial model $\mathcal{M}^{\mathcal{P}} = \langle \mathcal{D}^{\mathcal{P}}, \mathcal{I}^{\mathcal{P}}, \mathcal{G}^{\mathcal{P}} \rangle$, which is defined in terms of a new fluent set $\mathcal{F}^{\mathcal{P}}$. We assume that there exists an unknown surjective $\Lambda : \mathcal{F}^{\mathcal{P}} \rightarrow \mathcal{F}^{\mathcal{T}}$, that maps fluents from $\mathcal{F}^{\mathcal{P}}$ to fluents in $\mathcal{F}^{\mathcal{T}}$, which may include fluents that were not originally a part of the true model.

We will state that $\mathcal{M}^{\mathcal{P}}$ is a partial model that corresponds to $\mathcal{M}^{\mathcal{T}}$ if the application of the function Λ over the model components results in the corresponding model component for $\mathcal{M}^{\mathcal{T}}$. Model components here include action definitions, initial state, and the goal description. For example, the initial state and goal description of $\mathcal{M}^{\mathcal{P}}$ is given as follows, $\mathcal{I}^{\mathcal{T}} = \Lambda(\mathcal{I}^{\mathcal{P}})$ and $\mathcal{G}^{\mathcal{T}} = \Lambda(\mathcal{G}^{\mathcal{P}})$. Now, we can state that the partial model $\mathcal{M}^{\mathcal{P}}$ contains semantically equal but syntactically distinct fluents if there exists $f_1, f_2 \in \mathcal{F}^{\mathcal{P}}$, such that $f_1 \neq f_2$, but $\Lambda(f_1) = \Lambda(f_2)$.

In the running example, the true model corresponds to the unknown true model with a single unified fluent for having access to the legume needed for making hummus. Meanwhile, the partial model includes separate fluents for having access to chickpeas and garbanzo beans.

Our goal here is to use $\mathcal{M}^{\mathcal{P}}$ to identify a plan π that is valid in the unknown model $\mathcal{M}^{\mathcal{T}}$. Given the fact that we consider positive precondition models and we have a surjective mapping from $\mathcal{M}^{\mathcal{P}}$ action definitions to $\mathcal{M}^{\mathcal{T}}$, the only reason a plan π that is valid in $\mathcal{M}^{\mathcal{T}}$ would be invalid in $\mathcal{M}^{\mathcal{P}}$ is if there is at least one *broken* causal link in the plan for $\mathcal{M}^{\mathcal{P}}$ because of a semantically equivalent but syntactically distinct fluent pair. In the running example, the action `make_puree` has a precondition `(has-chickpeas)`, but there are no actions that would make it true (on the other hand, you have actions that will make `(has-garbanzo-beans)` true).

Generating Potential Plans

As discussed earlier, we will allow for the model to generate such plans by the use of additional "replace" actions which try to connect different fluent pairs. To do this, we will extend the model $\mathcal{M}^{\mathcal{P}}$ to a new model $\mathcal{M}'$. To start with, the fluents for this new model are given as $\mathcal{F}' = \mathcal{F}^{\mathcal{P}} \cup \mathcal{F}^{\mathcal{K}}$, where $\mathcal{F}^{\mathcal{K}}$ is a set of fluents that is used to track whether the value of some fluent in $\mathcal{F}^{\mathcal{P}}$ is known to be true (as such $|\mathcal{F}^{\mathcal{K}}| = |\mathcal{F}^{\mathcal{P}}|$). We will refer to this new set of fluents as the known status of the original fluents. The new actions are given as $\mathcal{A}' = \mathcal{A}^{\mathcal{R}} \cup \mathcal{A}^{\mathcal{P}'}$, where $\mathcal{A}^{\mathcal{R}}$ is the set of replace actions and $\mathcal{A}^{\mathcal{P}'}$ is obtained by updating the original action set $\mathcal{A}^{\mathcal{P}}$ by including the known status fluents.

In the new model, an action in $\mathcal{A}^{\mathcal{P}'}$ will now include a known status fluent in its add effect if the original fluent was part of the add effect or the precondition of the corresponding action definition in $\mathcal{A}^{\mathcal{P}}$ (and isn't part of the delete effect). We set the known status of the precondition fluents as true because the successful execution of the action confirms that the fluents in the precondition must have been true at the time of the action execution. The known status of a fluent is deleted whenever that fluent is part of the delete effect in the original action definition. Additionally, as we will see the known status is not made true by the replace actions. This means that even if a replace action added a fluent, this is to be treated as a possibility and is only known to be true if a future action uses it in a precondition.

Now, we will have a replace action for every pair of fluents in the original fluent set $\mathcal{F}^{\mathcal{P}}$. For the pair, f_1, f_2, the corresponding replace action will delete f_1 from the state and make f_2 true, provided the f_1 fluent has a true value in the state, and f_1 was known to be true. More formally, the action definition for $a^{\mathcal{R}}_{(f_1,f_2)}$is given as:

$$pre_{+}(a^{\mathcal{R}}_{(f_1,f_2)}) = \{f_1, f_1^{\mathcal{K}}\},$$
$$add(a^{\mathcal{R}}_{(f_1,f_2)}) = \{f_2\},$$
$$del(a^{\mathcal{R}}_{(f_1,f_2)}) = \{f_1, f_1^{\mathcal{K}}\}.$$

The cost of the actions in $\mathcal{A}^{\mathcal{P}'}$ is set the same as the original action in $\mathcal{A}^{\mathcal{P}}$, except that we place an extremely high penalty on the replace action, so it is only used when necessary. We set the cost of the replace action to be higher than the cost of the costliest plan possible under $\mathcal{A}^{\mathcal{P}}$. In future work, we will investigate assigning variable costs to different replace actions based on the semantic relevance of their fluents. If a fluent is part of the initial state, the corresponding known status fluent is also set to true in the initial state. The goal is kept the same as the partial model.

With the replace action in place, the new model $\mathcal{M}'$ can generate new plans by connecting an add effect and a precondition specified using different fluents. If we are using an optimal planner, it will only use a replace action if no other plans exist.

Inner Loop for Identifying a Valid Plan

Using this updated model with replace actions, $\mathcal{M}'$, we generate a plan, π'. All replace actions from this plan are then filtered out, and the remaining plan is checked against a simulator that captures the true model $\mathcal{M}^{\mathcal{T}}$, to see whether it is valid or not. If the filtered plan is valid for this domain, the algorithm stops. However, if the plan is not valid, the replace actions that made this plan invalid are identified (this corresponds to the last replace action that tried to add the failed precondition) and removed from the $\mathcal{M}'$ model. This loop then repeats until either a valid plan is found or the problem is determined to be unsolvable. The pseudo-code for this procedure is given in Algorithm 1.

Algorithm 1: Fluent Mapping Algorithm

Input: $\mathcal{M}^{\mathcal{T}}, \mathcal{M}'$
Output: A plan π' that is valid in $\mathcal{M}^{\mathcal{T}}$

```
is_valid_plan ← False
while is_valid_plan is false do
    for replace_action in ActionsToRemove do
        M' ← RemoveAction(M')
        π' ← GetPlan(M')
        π' ← FilterReplaceActions(π')
        if π' is valid in M^T then
            is_valid_plan ← True
        else
            π' ← AddBackReplaceActions(π')
            ActionsToRemove[] ← BadActions(π')
return π'
```

Evaluation

Our evaluation objective was to show that our approach could be used to identify relationships between semantically equivalent fluents and identify plans that are valid in the true model.

Datasets We evaluated our method on the standard Blocksworld and Gripper IPC domains [1], by creating variations of each original problem instance for use in our fluent mapping problem. Both domains were chosen for their size and simplicity, as the execution of each depends on a finite number of actions whose connection to a final goal state is clear and therefore easy to manipulate. This allowed us to test our method on more basic, controllable domains before trying more complex ones.

Blocksworld: This domain tests the situation when a *single* link between actions is broken. In Blocksworld, an agent is tasked with stacking blocks using four actions: pick-up, put-down, stack, and unstack. The fluent `(handempty)` tracks whether an agent is holding a block. When this fluent is true, the agent can pick up or unstack a block, resulting in `(not (handempty))`, which can only reverse if the agent either stacks or puts a block down. This variation is the total model.

To create the partial problem, a copy of the original Blocksworld problem adds the fluent `(not-holding)` to its list of predicates, and replaces `(handempty)` in the effect of the stack action with `(not-holding)`. Thus, once an agent stacks a block, the `(handempty)` fluent remains false while the `(not-holding)` fluent becames true, preventing the agent from picking up any additional blocks or fully executing its plan.

Gripper: This domain tests a situation where *two* links between actions are broken. In Gripper, an agent uses grippers to transport balls between two rooms with three actions: move, pick, and drop. The fluent `(free ?g - gripper)` tracks if an agent is holding a ball in one of its grippers. When `free` is false, the agent cannot pick up a ball with a given gripper, and when true, it can. The fluent `(at ?b - ball ?r - room)` tracks the location of balls in rooms. Once false, `(free ?g - gripper)`

[1] 2nd International Planning Competition, 2000. //github.com/potassco/pddl-instances/tree/master/ipc-2000 //github.com/potassco/pddl-instances/tree/master/ipc-1998

and (at ?b - ball ?r - room) can only become true again once the agent has executed a drop action. This variation is the total model.

To create the partial problem, a copy of the original problem adds the fluents (in ?b - ball ?r room) and (not-holding ?g - gripper) and replaces (at ?b - ball ?r - room) with (in ?b - ball ?r room) and (free ?g - gripper) with (not-holding ?g - gripper) in the effects the drop action. Thus, once an agent dropps a ball, the (at ?b - ball ?r - room) and (free ?g - gripper) remain false, while the fluents (in ?b - ball ?r room) and (not-holding ?g - gripper) become true. Because the Gripper move action requires the (at ?b - ball ?r - room) to be true, and the pick action requires (at ?b - ball ?r - room) and (free ?g - gripper), these changes disrupt the agent's ability to execute its plan by preventing it from moving or picking any more balls up.

Domain Replace Actions: These partial Blocksworld and Gripper domains were grounded, and replace actions composed of every fluent pair possible (except for the known fluents) were added to them. For Blocksworld, this totaled 900 replace actions, constructed from four actions and a mixture of six propositional and parameterized fluents. For Gripper, this also totaled 900 replace actions, from three actions, and six parameterized fluents. For each domain, a cost of 10,000 was assigned to all replace actions, while all actions native to these domains were assigned a cost of one. This was to force the planner to be selective about which replace actions it chose to use. Using the method outlined in Algorithm 1, these replace actions were then iteratively removed from each domain until the planner found a subset of them which allowed it to produce a valid plan.

Setup The original and partial Blocksworld and Gripper PDDL domain and problem files were inputted using our method. The original files were taken from a repository of benchmark PDDL problem instances from past years used in the International Planning (IPC) Competition. To create a partial version, a duplicate copy of these files was made and modified according to the methodology outlined in the previous section, to which replace actions were then added. Here, the partial files were used to generate a plan, and the original files were used to validate it with VAL. We used the Fast Downward Planner to run $\mathcal{A}^*$ search with the Landmark Cut heuristic (Helmert 2006). All experiments were performed on a Mac with a 2.6GHz 6-Core i7 processor and 16 GB Ram.

Results To create an initial benchmark, we tested our method on a single version of the Blocksworld and Gripper domains. These results are presented in Table 1 where the primary metrics tracked were the total time in seconds and number of iterations taken for a valid plan to be found, and the number of replace actions in the domain before and after a valid plan was found. Recall, during each iteration a single replace action was removed from the domain, meaning this number corresponds to the total number of replace actions removed *and* iterations taken for our problem to be solved. As such, we see that our method was able to find a valid plan by both leveraging and only removing a fraction of the replace actions available to it.

Domain	Blocksworld	Gripper
Total Time (sec)	11470.00	171394.05
Starting Replace Actions	900	900
Ending Replace Actions	853	709
Iterations / Removed Actions	47	191

Table 1: The first row lists the domains used in our experiment, Blocksworld and Gripper. The remaining rows show the total number of iterations/replace actions deleted, total time taken in seconds, and the starting/ending number of replace actions in each domain to find a valid plan.

Discussion From our empirical evaluation, we see our method successfully allowed a valid plan to be found in each domain. One shortcoming is that allowing replace actions to be created between every possible fluent increases the number of grounded actions significantly. In fact, for the problems we considered, we ended up with a total of 900 actions. This increase in grounded actions affected our problem runtime significantly. In our case, the planner spent substantial amounts of time reasoning about plans with replace actions that would never be executable in the real world. This was particularly evident in the Gripper domain.

Conclusion and Future Work

In this paper, we showed one possible method for generating a plan using a partial domain, that was valid given a total domain. Specifically, we looked at scenarios where the partial domain contained fluents that were semantically but not syntactically equivalent, resulting in broken links between actions and inexecutable agent plans. The purpose of this work was then to avoid these unwanted outcomes by using replace actions to map these fluents to each other. We tested this method on modified versions of the Blocksworld and Gripper domains, achieving promising results for each, where the agent was able to use a minimal set of replace actions to find a valid plan. In the future, we hope to extend this work by applying our method to more complex domains that contain multiple fluent pairs like this, as well as different types of actions and broken action links yet to be explored. We also hope to investigate the use of domain knowledge and the use of domain structural clues to identify more promising replace actions. Other possible venues of investigation include the use of more complex and realistic simulators. The use of simulators that provide more detailed feedback about the identification of specific failures could even be used to improve our current methods. Finally, our current paper focuses on the use of actions as a way to connect different fluents. However, this is one of many ways to achieve this. Other possibilities include the use of axioms, conditional effects, etc. Each of these methods could have computational trade-offs Investigating such trade-offs could be a fruitful future next step.

Acknowledgements

We thank NRL and ONR for funding this research. Sarath Sreedharan's research is supported in part by grant NSF 2303019.

References

Addis, A.; and Borrajo, D. 2011. From Unstructured Web Knowledge to Plan Descriptions. In *Information Retrieval and Mining in Distributed Environments*, volume 324, 41–59. Springer Berlin Heidelberg.

Aineto, D.; Jiménez Celorrio, S.; and Onaindia, E. 2019. Learning action models with minimal observability. *Artificial Intelligence*, 275(C): 104–137.

Bäckström, C.; and Jonsson, P. 2013. Bridging the Gap Between Refinement and Heuristics in Abstraction. In *Proc. of the Twenty-Third International Joint Conference on Artificial Intelligence*, 2261–2267. AAAI Press.

Bezawada, B.; Ray, I.; and Tiwary, K. 2019. AGBuilder: An AI Tool for Automated Attack Graph Building, Analysis, and Refinement. In *33th IFIP Annual Conference on Data and Applications Security and Privacy*, volume LNCS-11559 of *Data and Applications Security and Privacy XXXIII*, 23–42. Springer International Publishing.

Carman, M.; Serafini, L.; and Traverso, P. 2003. Web service composition as planning. In *Working notes of the Workshop on Planning for Web Services at ICAPS 2003*.

Cresswell, S.; Mccluskey, T.; and West, M. 2013. Acquiring planning domain models using LOCM. *The Knowledge Engineering Review*, 28.

Guan, L.; Valmeekam, K.; Sreedharan, S.; and Kambhampati, S. 2023. Leveraging Pre-trained Large Language Models to Construct and Utilize World Models for Model-based Task Planning.

Helmert, M. 2006. The Fast Downward Planning System. *The Journal of Artificial Intelligence Research*, 26: 191–246.

Hoffmann, J. 2015. Simulated Penetration Testing: From Dijkstra to Turing Test++. *Proc. of the International Conference on Automated Planning and Scheduling*, 25(1): 364–372.

Lamanna, L.; Saetti, A.; Serafini, L.; Gerevini, A.; and Traverso, P. 2021. Online Learning of Action Models for PDDL Planning. In *Proc. of the Thirtieth International Joint Conference on Artificial Intelligence*, 4112–4118. International Joint Conferences on Artificial Intelligence Organization.

Lindsay, A.; Read, J.; Ferreira, J.; Hayton, T.; Porteous, J.; and Gregory, P. 2017. Framer: Planning Models from Natural Language Action Descriptions. *Proc. of the International Conference on Automated Planning and Scheduling*, 27(1): 434–442.

Marrella, A.; and Chakraborti, T. 2021. Applications of Automated Planning for Business Process Management. In *Business Process Management*, 30–36. Springer International Publishing.

Sacerdott, E. D. 1973. Planning in a hierarchy of abstraction spaces. In *Proc. of the 3rd International Joint Conference on Artificial Intelligence*, 412–422. Morgan Kaufmann Publishers Inc.

Santiago, N.; and Mendez, J. 2023. Analysis of Common Vulnerabilities and Exposures to Produce Security Trends. In *Proc. of the 2022 International Conference on Cyber Security*, 16–19. Association for Computing Machinery.

Sreedharan, S.; and Kambhampati, S. 2021. Leveraging PDDL to Make Inscrutable Agents Interpretable: A Case for Post Hoc Symbolic Explanations for Sequential-Decision Making Problems. In *Working notes of the Workshop on Explainable AI Planning at ICAPS 2021*.

Srivastava, S.; Russell, S.; and Pinto, A. 2016. Metaphysics of planning domain descriptions. In *Proc. of the Thirtieth AAAI Conference on Artificial Intelligence*, 1074–1080. AAAI Press.

Stern, R.; and Juba, B. 2017. Efficient, safe, and probably approximately complete learning of action models. In *Proc. of the 26th International Joint Conference on Artificial Intelligence*, 4405–4411. AAAI Press.

Verma, P.; and Srivastava, S. 2020. Learning Generalized Models by Interrogating Black-Box Autonomous Agents. In *Working notes of the Workshop on Generalization in Planning at AAAI 2020*.

Wu, K.; Yang, Q.; and Jiang, Y. 2007. ARMS: an automatic knowledge engineering tool for learning action models for AI planning. *The Knowledge Engineering Review*, 22(2): 135–152.

Yosifova, V. 2021. Vulnerability Type Prediction in Common Vulnerabilities and Exposures Database with Ensemble Machine Learning. In *2021 International Conference Automatics and Informatics*, 146–149.