Theory Alignment via a Classical Encoding of Regular Bisimulation

Alex Coulter*, **Teo Ilie***, **Renee Tibando***, **Christian Muise**

Queen's University, Kingston, On, Canada
{alex.coulter, 17ti5, 17rat3, christian.muise}@queensu.ca

Abstract

Bisimulation, at its core, is a means of studying the alignment between two dynamical systems. It has been used to great effect in the planning community for heuristic computation; simulating the full state space in the space of abstractions (merge-and-shrink heuristics). Here, we consider the direct task of theory alignment – assessing if two planning problems are "equivalent" – through the lens of *regular bisimulation*. We accomplish the task through a novel encoding that merges the two theories as a new planning problem, where the encoded problem is unsolvable if and only if the two theories are a regular bisimulation. We demonstrate that modern planners are capable of solving many of these encodings, and the solutions (if plans exist) provide a rich explanation as to why two models differ. The work has already had a direct and practical impact, being deployed in a classroom setting to assess the correctness of student-authored planning models as compared against a reference solution. Our solution has a direct impact on being able to verify if a candidate planning model matches a known specification, and opens the door to model verification through planning techniques.

1 Introduction

Aligning two planning theories – the task of seeing how two models relate – can have many applications. Notable examples include continual refinement to improve human-robot interaction (Chakraborti et al. 2017), model acquisition of action theories (Fervari, Velázquez-Quesada, and Wang 2021), and heuristic computation for classical planning (Helmert et al. 2014). Here, we consider the direct task of aligning two planning theories that share a number of elements from the high-level PDDL specification: types, objects, constants, and action names + parameters. The main question we seek to answer is how we can test if two models align (where the fluents and action implementations may differ), and if not, where that mis-alignment occurs. While our encoding works directly at the PDDL model, our work is built on a foundation of *regular bisimulation*.

The area of bisimulation has been mined by the planning community for several aspects. Most notably, it has lead to the extremely impressive and popular line of merge-and-shrink heuristics (Katz, Hoffmann, and Helmert 2012; Helmert et al. 2014) found in the state-of-the-art Fast Downward planning system (Helmert 2006). Loosely defined, a bisimulation is a correspondence between two transition systems such that labeled transitions between the two systems coincide: if two nodes (one from each system) coincide, then the reachable nodes in each system also coincide (following the labeled transitions). We focus on *regular* bisimulation, where the labels of the transitions are presumed to be the same, as well as the models' reachable state space.

Due to the ultimate application of comparing PDDL models, our approach is grounded firmly in the manipulation of PDDL. Taking two planning models (represented in domain+problem PDDL files), we *merge* them through a novel encoding to a new domain+problem file. This merged domain retains the original types, action names/parameters, objects, and constants (assumed to be the equivalent in both of the original models), and combines the action specifications to progress through both models simultaneously. Similarly, fluents and initial states are merged. Finally, "failure actions" are introduced that represent a potential misalignment between the two models. As long as one of these failure actions can be executed, the two theories do not align.

The encoding was implemented, and embedded as a core analytic in an undergraduate AI class where one of the assignments was to create a planning model (fluents, action preconditions/effects, initial/goal states) according to a text-based specification. Two reference models were created to compare against, and the proposed alignment was an integral part of the teaching staff's analysis of student submissions. We found that our proposed alignment was not only viable, with many submissions having "solutions" to the merged model showing where a modeling error occurs, but several cases demonstrated errors with the submitted domains that were subtle and detected only by this added approach.

Next, we cover some of the preliminary material required to understand the approach we take. We then describe our main approach in Section 3, grounded primarily in the PDDL processing that was required to realize the ideas we put forward. We discuss the (mainly qualitative) results in Section 4 and conclude with a brief discussion in Section 5.

2 Background

Our approach operates entirely on the Planning Domain Definition Language (PDDL) and for a more complete treat-

*These authors contributed equally.

ment, the interested reader can refer to (Haslum et al. 2019). Here, we detail some of the core functionality and notation used through the paper.

Figure 1 shows a complete example in PDDL, and the specific features we make use of mostly surround actions. Specifically, we use the following notation:

- $parameters(\texttt{act})$: The typed parameters of the action `act`. Referring to Figure 1, $parameters(\texttt{turnon})$ would be just $?l - light$.
- $precond(\texttt{act})$: The precondition of action `act`. We assume simple effects as a conjunction of predicates or their negation.
- $effects(\texttt{act})$: The effects of action `act`. We allow arbitrary ADL effects, but primarily focus on those theories with a conjunction of conditional, add, and delete effects.

We assume that the input planning models are potentially in lifted form (so ungrounded actions), and our compilation retains this assumption.

3 Approach

Here, we describe the high-level approach we take to aligning two planning models. Specifically, we start by detailing the core assumptions under which alignment can take place, then describe how two models can be merged to create a third that will allow us to test for alignment. Finally, we discuss how the merged model and its solutions should be interpreted, and discuss the connection to regular bisimulation.

3.1 Specification Assumptions

There is a wide space of ways one might interpret the problem of "aligning two planning models". Finding an isomorphism between action names, focusing on fluent-defined reachable state spaces, etc. Here, we detail precisely the setting we are interested in, motivated (in large part) by the application of our work in the classroom setting.

Types We assume that the typing for each of the two models are precisely the same. This is required in order to adequately handle the action parameter space. While slight deviations may be possible[1], we simply assume that the set of types in both models are precisely the same.

Objects / Constants Similarly, we assume that the objects and constants, including the types they adopt, are precisely the same between the two models. Moreover, we assume that both of the models use these objects in the same way. For example, imagine we have two models to align that capture the mechanics of flipping a pair of switches in a real physical room (one left and one right). If the models share the objects or constants `switch1` and `switch2`, then those objects must consistently refer to the left and right switches (or vice versa) across two models. This is necessary, as a renaming of the objects could lead to a misalignment between the two models unnecessarily. It is an open question for future work to consider encodings that relax this assumption.

[1]One example would be having some types that do not play a role in the action parameter specification, but only on constants and fluents mentioned in the action implementation.

Action Names & Parameters Because we are looking for an alignment of the state spaces, the space of full ground action specifications must coincide. We can ensure this by making sure the same action names and `:paramaters` definition is used in both of the models under consideration. This will mean that the set of ground actions will have the same reference in both of the models.

Initial States We assume that each model captures the same initial state semantically. Note, however, that this is not something we can check through syntactic analysis. But, rather, it is a property we assume in even deciding to make this alignment. If the two models captured different initial states, then we wouldn't expect them to align.

Fluents We *do not* assume that the fluents are aligned. This includes both the naming and parameterization. This opens the door to a wide-range of alignment possibilities from semantically equivalent (as defined by regular bisimulation) models. If the two models use the same name/symbol for a fluent, we will *not* assume them to be equivalent (as they may be used in different ways).

Action Preconditions & Effects Finally, we *do not* assume there to be any syntactic equivalence between the action preconditions / effects of the two models. This allows us to compare wildly different encodings of the same real-world system. It is also an essential element, given that the fluents are not assumed to coincide. The effective application of the action will be analyzed through the alignment process, but the actual implementation (using each model's custom set of fluents) must not coincide syntactically in any way.

3.2 Encoding

The encoding process creates a domain+problem PDDL pair that corresponds to the merged domains that will be aligned. Integral to this encoding, every shared action has a pair of "failure actions" introduced that correspond to the situation where the action can be executed in one model, but not the other.

Definition 1 (Failure Action)**.** For a shared action `act` with parameters $parameters(\texttt{act})$, preconditions $precond_1(\texttt{act})$ / $precond_2(\texttt{act})$, and effects $effects_1(\texttt{act})$ / $effects_2(\texttt{act})$, we can define `failure_act_1` as follows [2]:

$$parameters(\texttt{failure\_act\_1}) = parameters(\texttt{act})$$

$$precond(\texttt{failure\_act\_1}) = (\texttt{and}\ precond_1(\texttt{act})\ (\texttt{not}\ precond_2(\texttt{act})))$$

$$effects(\texttt{failure\_act\_1}) = (\texttt{failed})$$

Note that we do not require the effects from either version of the action: *the applicability of a failure action is enough to ensure that a transition can occur in one input planning model but not the other*. Each of the original action specifications will have a pair of "failure actions" created, and

[2]The dual action is defined analogously.

these correspond to the two options of progression in one domain but not the other. One final note on practicality – the planning system used must be able to handle nested preconditions such as those created by the scheme above. This was true of the FastDownward planning system used to solve these problems (Helmert 2006).

We can now describe the full encoding in detail. The process is as follows:

1. Read in both of the input planning models. We use Tarski for this part and further manipulation (Francés, Ramirez, and Collaborators 2018).
2. Rename all of the predicates/fluents throughout both models so they are unique.[3].
3. Include all of the syntactically shared elements: types, objects, and constants.
4. Create a merged initial state `(:init ...)` by combining the initial states from both input models.
5. Create a merged section, `(:predicates ...)`, by combining the uniquely defined (and now renamed) predicates from both input models.
6. Add `(failed)` as a predicate, and set the goal to just be achieving this fact.[4]
7. Create a single merged action for shared action in the original specifications: the preconditions and effects are simply concatenated with an `(and ...)` term.
8. For every action, created a pair of "failure actions".

We illustrate the procedure by using a simple light-switch example. Figures 1 and 2 demonstrate the two planning models to be aligned, while Figure 3 shows the merged model in its entirety.

3.3 Interpreting Solutions

If no solution exists, it means that there is no reachable state where an action is executable in one model but not the other. Thus, if a planner determines the merged model to be unsolvable, the two original models align. While showing a model to be unsolvable is arguably harder than finding a plan (Muise and Lipovetzky 2015), we found that the majority of tested instances were indeed manageable. Further, if the planner struggles to find a solution within resource limits, it is strongly suggestive that no solution exists. While this may not be useful when guarantees are required, it is indeed useful for our use case, where plan existence is simply an aid to understand errors in a model.

With the merged domain, the only way to achieve the goal is for one of the "fail actions" to be executed. If this happens, it means that the plan represents a sequence of actions executable in both original models that arrives at a pair of states s_1 and s_2 in each model respectively. From that point, if a

[3]We prepend a prefix to every occurrence of a fluent (e.g., adding **domain1_** or **domain2_** to the start of every fluent name)

[4]Any predicate sharing this name from the input models would be renamed already).

```
(define (domain lights1)

    (:requirements
        :negative-preconditions :typing)

    (:types light)

    (:predicates (on ?l - light))

    (:action turnon
        :parameters (?l - light)
        :precondition (and (not (on ?l)))
        :effect (and (on ?l))
    )

    (:action turnoff
        :parameters (?l - light)
        :precondition (and (on ?l))
        :effect (and (not (on ?l)))
    )
)

(define (problem lighttprob1)
   (:domain lights1)
   (:objects light1 light2 - light)
   (:init (on light1))
   (:goal (and (on light1)
               (on light2))))
```

Figure 1: Simple planning model for switching lights

"fail action" is executable (and thus the merged model solvable), an action is applicable in one model but not the other. We have thus found an candidate for why the two original models *do not align*. Many such examples may exist, and the planner will naturally find one of shorter length. We found the when models misalign, the planning process to demonstrate as such was very quick (sub-second solve time).

As an example, consider the "broken" model in Figure 4 for the light switch example discussed previously (only the broken action is shown, and the rest corresponds to Figure 2). The error introduced mirrors the common mistake of forgetting to delete a fluent. The found plan for the merged domain is as follows:

```
1: (turnon light2)
2: (fail_turnon2 light2)
```

Notice that the final action, which achieves our `(failed)` predicate, indicates that the action was allowed to execute in the second domain (with the failed action), but not the first. Without deleting the `(off ?l)` fluent in the action effects, it erroneously allows the light to be both on and off simultaneously. Also note that the solutions of the two domains coincide – both, if given to a planner, would come up with a correct solution to just turn on `light2`. It is only through the analysis we present here that the misalignment between the two models is detected.

Given the assumptions we outline in Section 3.1, there are

```
(define (domain lights2)

    ; Same requirements and types

    (:predicates
        (on ?l - light)
        (off ?l - light)
    )

    (:action turnon
        :parameters (?l - light)
        :precondition (and (off ?l))
        :effect (and (on ?l) (not (off ?l)))
    )

    (:action turnoff
        :parameters (?l - light)
        :precondition (and (on ?l))
        :effect (and (not (on ?l)) (off ?l))
    )
)

(define (problem lighttprob2)
   (:domain lights2)
   (:objects light1 light2 - light)
   (:init (on light1) (off light2))
   (:goal (and (on light1)
               (on light2))))
```

Figure 2: An alternative model for light switching

three potential sources of failure for model alignment:

1. Initial state.
2. Preconditions of an action.
3. Effects of an action.

A final "failure action" `failure_act` may be due to a mismatch of the `act` preconditions or effects, but it also may be due to *another* action's effects or differences in initial state implementation. It is thus interesting to consider how to use a found plan in order to diagnose a misalignment between models.

Failed Action Precondition The most common source of error found is an error in precondition of the failed action. Consequently, the first step in analyzing a plan is to contrast the preconditions of `act` in both of the models, and see if there is a mismatch in what is implemented.

Previous Action Effect If the two models capture `act` similarly, then the next most common issue leading to misalignment is a previous action in the plan. Typically, candidate solutions that lead to a failure are short, and so the space of actions that must be considered is limited. Further, it would need to be an action that directly influences the failed action `act` (through its preconditions), and so that restricts the analysis further. It is worth noting, however, that because we have generalized to actions with conditional effects, it may not be as simple as exploring the actions that correspond to those achieving `act`'s preconditions – rather, it may be actions further back in the plan that lead to different conditional effects firing.

```
(define (domain lights3)

    ; Same requirements and types

    (:predicates
        (domain1_on ?x1 - light)
        (domain2_on ?x1 - light)
        (domain2_off ?x1 - light)
        (failed))

    ; For space, only turnon actions shown

    (:action turnon
        :parameters (?l - light)
        :precondition (and
            (not (domain1_on ?l))
            (domain2_off ?l))
        :effect (and
            (domain1_on ?l)
            (domain2_on ?l)
            (not (domain2_off ?l))))

    (:action fail_turnon1
        :parameters (?l - light)
        :precondition (and
            (not (domain2_off ?l))
            (not (domain1_on ?l)))
        :effect (and (failed)))

    (:action fail_turnon2
        :parameters (?l - light)
        :precondition (and
            (domain1_on ?l)
            (domain2_off ?l))
        :effect (and (failed)))
)

(define (problem lighttprob3)
    (:domain lights3)
    (:objects light1 light2 - light)
    (:init
        (domain1_on light1)
        (domain2_on light1)
        (domain2_off light2))
    (:goal (failed)))
```

Figure 3: Merged light switching domain

Initial State The most rare source of errors is a misalignment in the initial state implementation. Depending on the planning problem being modeled, plans that demonstrate this error may range in size (depending on how hard it is to get to a point in the state space that relies on the erroneous initial state). In a sense, this source of error can be seen as

```
(:action turnon
    :parameters (?l - light)
    :precondition (and (off ?l))
    :effect (and (on ?l))
)
```

Figure 4: An erroneous action definition for light switching

analogous to the "Previous Action Effect" errors, when one views the initial state as the effects of a single action at the start of a plan.

3.4 Potential Extensions

While the approach described above has led to a powerful and usable system for model alignment already, there are several key avenues for potential extensions to provide even better analysis of the model's misalignment. We detail some of them here.

Precondition Analysis The only true failure mode of the encoding above is for an action to be applicable in one model, but not the other. Applicability, by its very definition, is concerned with precondition satisfaction. *Knowing which aspect of the precondition has failed can lead to greater insight.* Currently, this is not offered, and users of the system must infer such details manually.

Goal Analysis There is nothing in the above approach that speaks to the implemented goal of the pair of input models. If there is a misalignment between them, this will not be detected. One simple work-around would be to include a "goal-achieving action" that can be executed only if the goal is satisfied. Then, we would expect a plan for misalignment to exist that finishes with this goal-achieving action (meaning the goal holds in one model but not the other).

Plan Diversity We found that a simple failure may be detected early on, and other issues with the models go unchecked. One way to mitigate this would be to generate a diversity of plans for the merged model. This may both (1) surface multiple sources of error; and (2) demonstrate commonalities between the plans that point to a source of error (e.g., all sharing an early action that has a misalignment on the action's effect).

Iterative Model Refinement Perhaps the most useful next step would be to enable the iterative refinement of models. In the spirit of (Chakraborti et al. 2017), we could (1) detect an error using everything described above; (2) "fix" the model that appears to have the error; and (3) re-start the process to look for further sources of mis-alignment. Currently, this is not offered as a possibility of the implementation.

3.5 Connection to Regular Bisimulation

Our work is largely motivated by, and intimately tied to, the setting of PDDL. Despite the practical nature of the work, there is a rich theory behind what is being analyzed – plans in the merged model effectively provide us with a proof of contradiction to the two models being a regular bisimulation (Milner 1990). Regular bisimulation captures the notion that there is a mapping between two transitions systems such that labeled edges between states of the system coincide: every sequence of labels in one must be mirrored in the other. For two models that are regular bisimilar, the reachable state space of our merged model represents this bijective mapping.

Unlike the common characterization of determining if two transition systems are bisimilar, our setting begins with a shared mapping on the initial state, and further benefits from the labeled edges being uniquely defined (based on action applicability). This means that we know precisely how the two transition systems (coming from the reachable state spaces of each model) would align. What is left for the planner to decide is (1) if they are indeed regular bisimilar (the case when no plan exists); or (2) if they are not, a candidate explanation as to why in the form of a sequence of actions.

While this connection is straight-forward at the outset, we feel it is an important one to highlight: if we consider relaxed or different forms of bisimulation, this may yield interesting new encodings and model reconciliation settings beyond what is considered here.

4 Evaluation

The method above was implemented using the Tarski library for reading, synthesizing, and writing the PDDL involved (Francés, Ramirez, and Collaborators 2018). The expressivity of PDDL handled is generally STRIPS-like domains with negative preconditions and conditional effects. Richer forms of effects would also be readily handled, since our method does not require advanced manipulation of them: they are simply merged. The preconditions of the actions need to be negated for the "fail actions", and this was readily handled by the planner used for the compiled domain (Fast Downward).

As a very grounded and useful application of the methods, we deployed the system as a primary driver for grading a PDDL authoring assignment in an undergraduate AI class: 86 groups of 1-3 students were tasked with creating PDDL models from a given specification, and the assignment included assumptions of the form found in Section 3.1. Here, we detail some of the results of this usage.

4.1 Assignment Setup

The assignment asked students to implement a model that captured a hero going through a dungeon: locked corridors, colour-coded keys/locks, limited-use keys, collapseable corridors, and final treasure chest location were all features of the setting. The implementation involved devising (1) the fluents for the domain; (2) the preconditions and effects for 4 actions (move, pickup, drop, unlock); (3) the initial and goal states for 3 described problems; and (4) a novel problem with limited rooms and a minimal shortest plan length (requiring students to come up with a nesting of keys & locked doors that forced long plans). Crucially, the students were provided with a skeleton of the domain and problem files that correspond precisely to the assumptions in Section 3.1. Starting PDDL templates can be found in the session,

http://editor.planning.domains/#read_session=6Xczw7tbFC

Problem	Solve	St-Val	Ref-Val	Aligns Orig	Aligns Move
p01	✓	✓	✓	✗	✗
p02	✓	✓	✓	✗	✗
p03	✓	✓	✓	✗	✗

Figure 5: Single Problem Analysis

4.2 Other Grading Indicators

The alignment to a reference model is only one aspect used to support the grading of assignments. Other planning-oriented functionality included,

1. Running student domain/problem files to generate plans.
2. Validating student-found plans with the reference model.
3. Validating reference plans with the student model.
4. Align to a model which included the most common error.

Typically, grading would stop at the first (i.e., running the student's models and manually inspecting). Steps 2 and 3 are made possible as a result of the set of assumptions we put in place. Specifically, the plans (grounded and parameterized actions) should be valid in both models since the action names and objects are the same. The final check permits further analysis using the alignment technique described above. This is particularly useful when a common error causes the alignment to the original reference model to fail in a predictable and consistent way.

4.3 Summary of Usage

For the given assignment, teaching assistants were instructed to document where errors occurred (e.g., `pre(move)`) and further include which analysis led to the errors found. The follow statistics summarize the findings:

Total # of Assignments	86
Assignments With Plan-based Errors	11
Assignments With Validation Errors	31
Assignments With Alignment Errors	67
Those With Multiple Alignment Errors	9

The introduction of this assignment grading methodology demonstrated the clear advantage of using stronger planning-based mechanisms in this setting. Nearly 3x more errors were found with validation techniques (i.e., running the found plans on another model) compared to just analyzing the plans produced, and over 6x more errors were found with the alignment analysis following the work presented in this paper. Further, in several of the cases we had assignments that would validate just fine (student plans work for the reference model and vice versa), but the errors only manifested in the alignment analysis. These subtle bugs are the ones that historically go unchecked for PDDL assignments.

4.4 Example Application

As an example beyond the simple light switch, here we show the analysis of a single submission. Figure 5 shows the information presented to the grader. Failures in "Aligns Move" are due to the group *not* making the common error (alignment should only work for at most one reference model). Failures for "Aligns Orig" can be seen in the example plan:

```
Mis-alignment plan for p01 (version: orig):
(move loc12 loc22 c1222)
(pick-up loc22 key1)
(move loc22 loc23 c2223)
(unlock loc23 c2324 red key1)
(fail_unlock2 loc23 c2324 red key1)
; cost = 5 (unit cost)
```

Upon inspecting the suspicious unlock example, it is clear that the team neglected to change the locked status of the corridor (it should be deleted as an effect):

```
(:action unlock

    :parameters (...)

    :precondition (and
        ...
        (cor-locked ?cor ?col)
        ...)

    :effect (and
        (cor-unlocked ?cor)
        (when (key-two-use ?k)
              (key-one-use ?k))
        (when (key-one-use ?k)
              (key-used-up ?k))))
```

This is a common modeling error, and one that does not affect any of the found plans for the setting. It is only through the alignment that such errors become readily detectable.

5 Summary

In this paper, we have introduced a novel encoding for the task of aligning two planning models that share some key aspects: objects, constants, types, and action signatures. The models are free to define the predicates and action preconditions/effects in any way and the alignment is conducted through the systematic search in a merged state space for a state where one model diverges from another. In our setting, this means that a particular (shared) action is applicable in one model for a found state, but not in the other model. The fundamental principle captured by the merged model is *regular bisimulation*, and we found that modern planners are very capable at finding counterexamples to two models being regular bisimular. Our approach has been successfully deployed in a classroom setting, and has led to substantial improvements in analytic support for the teaching staff marking a PDDL authoring assignment. So not only does our proposed method address a fundamental question in how to provide certificates is misalignment for regular bisimulation, but it also offers a powerful modeling or assessment tool for comparing two planning models.

References

Chakraborti, T.; Sreedharan, S.; Zhang, Y.; and Kambhampati, S. 2017. Plan Explanations as Model Reconciliation: Moving Beyond Explanation as Soliloquy. In *IJCAI*.

Fervari, R.; Velázquez-Quesada, F. R.; and Wang, Y. 2021. Bisimulations for knowing how logics. *The Review of Symbolic Logic*, 1–37.

Francés, G.; Ramirez, M.; and Collaborators. 2018. Tarski: An AI Planning Modeling Framework. https://github.com/aig-upf/tarski.

Haslum, P.; Lipovetzky, N.; Magazzeni, D.; and Muise, C. 2019. *An Introduction to the Planning Domain Definition Language*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers.

Helmert, M. 2006. The Fast Downward Planning System. *J. Artif. Intell. Res.*, 26: 191–246.

Helmert, M.; Haslum, P.; Hoffmann, J.; and Nissim, R. 2014. Merge-and-Shrink Abstraction: A Method for Generating Lower Bounds in Factored State Spaces. *J. ACM*, 61(3): 16:1–16:63.

Katz, M.; Hoffmann, J.; and Helmert, M. 2012. How to Relax a Bisimulation? In McCluskey, L.; Williams, B. C.; Silva, J. R.; and Bonet, B., eds., *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling, ICAPS*. AAAI.

Milner, R. 1990. Operational and Algebraic Semantics of Concurrent Processes. In van Leeuwen, J., ed., *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, 1201–1242. Elsevier and MIT Press.

Muise, C.; and Lipovetzky, N. 2015. Unplannability IPC track. *Proc. of the 2015 Works. on the IPC (WIPC 2015)*.