DOVETAIL - An abstraction for Classical Planning using a Visual Metaphor

Maurício Cecílio Magnaguagno and **Ramon Fraga Pereira** and **Felipe Meneguzzi**
School of Informatics
Pontifical Catholic University of Rio Grande do Sul
Porto Alegre - RS, Brazil
{mauricio.magnaguagno, ramon.pereira}@acad.pucrs.br
felipe.meneguzzi@pucrs.br

Abstract

While domain descriptions are often shared and manipulated through diagrams, most complex domains are still described using text-based languages. Code becomes an intermediary between the real-world and an abstract idea, and the programmer is merely a converter of diagrams into code. For automated planning this is no different. The state transition function is described in terms of a textual representation of actions and, although simple actions require little effort to define by the user, the learning process is often slow. New users have no metaphor to help them to visualize the domain description that they are working on and little information about why a planner fails due to formalization errors. In this paper, we propose a visual abstraction for both the planning domain actions and the planning process itself, to facilitate the design of classical planning domains. Using this abstraction, we expect to improve the learning curve for defining and subsequently diagnosing problems with new planning domains.

1 Introduction

Automated planner implementation is often concerned with speed, resources, and the expressive power their description and subsequent algorithms can deal with. Thus, the process through which domains are encoded for such systems is usually a secondary concern. Planning domains are described within a text file, following a standard, such as the *Planning Domain Definition Language* (PDDL) (McDermott et al. 1998). Planning systems parse planning domains into their internal representation and provide either successful plans or error messages regarding the impossibility to generate a plan from the domain as feedback. Users are left to their own devices to determine whether the input is correct but the problem is not solvable, or worse, whether the problem is erroneously solved. Few development tools for planning languages exist and do not provide any help regarding operator inconsistency. Planning systems provide little additional information in case of failure. Nevertheless, information originating directly from the internal data structures of the planning system is often helpful to *debug* planning domains. However, for more complex domains, such information is likely to be too much for a human to successfully understand the reasons behind a planner failure. A common practice to learning in science is to draw the problem in an informal and subjective way, usually targeting the specific domain and giving an explanation of the diagram to others for help (Ainsworth, Prain, and Tytler 2011). In our experience, we have observed similar behavior in students of automated planning as one of their strategies to understand planning domain descriptions. Without a clear metaphor and interface, new users of planning systems struggle to understand how to describe planning domains.

In this paper, we describe a metaphor to visualize and manipulate the only part that is not automatic in an automated planner, the formalization of planning domains. Our approach is inspired by a tool created by Brett Victor's[1] user interface to edit both input (code) and output (result). We have found this tool to be intuitive and thus beneficial to the learning and understanding process. Our primary contribution is a graphical representation for PDDL that aims to help new users of automated planners to understand complex planning domains by allowing them to visualize how actions interconnect to form a plan We begin by reviewing the background on concepts of visual representation and planning before describing our graphical representation for PDDL. We then demonstrate with a use case, the modelling of a problem from scratch in our notation, showing how details of planning can be visualized. We finish with a model validation of our notation, and finally, we discuss our conclusions and future directions.

2 Background

Planning Representation

We represent planning problems using the following terminology (Ghallab, Nau, and Traverso 2004, Chapter 1 – page 17).

Definition 2.1 (**Predicates and State**). *A predicate is denoted by an n-ary predicate symbol $\mathcal{P}$ applied to a sequence of zero or more terms (τ_0, τ_2, ..., τ_n). For example, a predicate is any construct of the form $\mathcal{P}(\tau_0, \tau_2, ..., \tau_n)$.* Terms are either constants or variables, and when all terms of a predi-

[1] http://worrydream.com/

cate are constants we call it a ground predicate. A state is a finite set of ground predicates (effectively, propositions).

Definition 2.2 (**Operator**). *An operator is represented by a 3-tuple $\mathcal{A}$ = $\langle$name($\mathcal{A}$), pre($\mathcal{A}$), eff($\mathcal{A}$)$\rangle$: name($\mathcal{A}$) represents the description or signature of $\mathcal{A}$; pre($\mathcal{A}$) describes the preconditions of $\mathcal{A}$, a set of predicates that must exist in the current state for $\mathcal{A}$ to be executed; eff($\mathcal{A}$) represents the effects of $\mathcal{A}$. These effects are divided into eff($\mathcal{A}$)$^+$ (i.e, an add-list containing the positive predicates) and eff($\mathcal{A}$)$^-$ (i.e, a delete-list containing negated predicates).*

When an operator is variable free (i.e. all predicate terms are constants) it is called an action. Such actions can be applied to a state if all preconditions are satisfied, resulting in a new state containing all predicates from eff($\mathcal{A}$)$^+$ none of the predicates of eff($\mathcal{A}$)$^-$.

Definition 2.3 (**Planning Instance**). *A planning instance is represented by a tuple $\Pi = \langle \Xi, \mathcal{I}, \mathcal{G} \rangle$, in which $\Xi = \langle \Sigma, \mathcal{A} \rangle$ is the domain definition, which specifies the domain knowledge comprising a finite set of predicates Σ, and a finite set of actions $\mathcal{A}$; $\mathcal{I} \subseteq \Sigma$ is the initial state specification; and $\mathcal{G} \subseteq \Sigma$ is the goal state specification.*

Classical planning representations often separate the definition of $\mathcal{I}$ and $\mathcal{G}$ as part of a planning problem to be used together with a domain Ξ.

Definition 2.4 (**Plan**). *A plan is a sequence of actions π = $\langle a_1, a_2, ..., a_n \rangle$ that modifies the initial state $\mathcal{I}$ into one where the goal state $\mathcal{G}$ holds by the sequential execution of actions $a_i \in \pi$.*

Graphical Representations

Graphical representations are commonly used to convey complex ideas through symbols. These representations are meant to be easily interpreted by people, by creating an analogy with previous experiences. Notations that use a consistent and clear set of symbols as primitives are often easier to grasp by users and facilitate conveying ideas graphically. For example, *Scratch* (Malan and Leitner 2007) is a visual language aimed at school-level pupils that uses blocks to represent different programming statements. The connections between blocks indicate the possible connections between programming statements, allowing users to experiment with different combinations of program statements. This idea of high level blocks exposed for experimentation is part of *Kodu*[2], but instead of recreating imperative languages structure the blocks are related to sensing and action. Outside programming languages, graphical languages are often used to visualize other processes, such as the language *Gantt charts* (Wilson 2003) uses to show how tasks are correlated and how much time is expected to complete them, while *Waveforms* (Ha 2010, Chapter 1 – page 2) are used to express the behavior of analog or digital data through time.

Learning Planning Languages

The key objective of planning languages is to describe a set of valid operations and properties of a specific domain in order to solve a planning problem (Gelfond and Lifschitz 1998). After a succession of planning languages starting with the *Stanford Research Institute Problem Solver* (STRIPS) (Fikes and Nilsson 1971), PDDL became a standard formalization language aiming at research and objective comparisons between planners. PDDL uses a layered structure for language features in order to facilitate its use and dissemination, though only a few planners support more than STRIPS

The key challenge of learning planning languages is dealing with a declarative specification and overcoming the thinking patterns associated with programming using imperative languages. Passing arguments and unification are different concepts that may look similar in source code, but mean completely different things. Whereas in imperative programming a user tells which variable assignment is required for a function to work, a planner aims to find, on its own, the correct values for the action parameters that solves the planning problem. A simple movement action, in which an object that occupies a position is moved to occupy a different position is enough to illustrate some common problems. For example, if an object occupies a new position, it does not mean that the old position is cleared. The need to declaratively express all real and implied changes to a state upsets new users, especially when these changes result in the planner determining whether some solutions are possible or not due to user mistakes in the domain specification. Understanding individual actions is not particularly challenging, since a visualization of the add and delete lists is often sufficient. The main challenge is understanding, in complex domains, the connection between multiple states and actions applied to these states. New users often benefit from diagrams to understand the simple act of movement with few lines of code, looking for a bug that remains invisible for the user following common sense (that was not explicitly formalized) about being in one place at a time. Here, position is a predicate that should be modelled as mutually exclusive for the same object, since the planner considers this relation between object and place as a feature, and an object may have an infinite number of features. This is not necessarily a problem, since a ball may have the color red and blue, and both features co-exist.

Listing 1 shows a PDDL specification for a move action that illustrates a common mistake made by inexperienced PDDL programmers. The highlighted code is an example of forgotten lines, replicating the moving object *?obj* at the destination *?to*, instead of clearing the position *?from*.

```
(:action move
 :parameters (?obj ?from ?to)
 :precondition (and (clear ?to)
                    (at ?obj ?from) )
 :effect (and (clear ?from)
              (at ?obj ?to)
(not (clear ?to)) (not (at ?obj ?from))
))
```

Listing 1: *Move operator with commonly missing effects highlighted.*.

[2]http://www.kodugamelab.com/about/

3 A Visual Abstraction for Classical Planning

Several metaphors are used to simplify our understanding of abstract ideas. Imperative code is an abstraction that simplifies our understanding of a computer's behavior that focuses on writing instructions quickly, but without a clear understanding of the underlying data. The programmer must mentally maintain a model of the behavior of the data being modified throughout its execution. Languages with an interactive shell have the results of each instruction shown and exhibit such model through experimentation. PDDL has a specific group of users that do not focus on the language or data itself, but on the planner. Although few PDDL editors with code completion are available, code completion by itself does not facilitate understanding the relations defined in the preconditions and effects of an operator. For example, it may be useful to visualize the mutual exclusion relations used by GraphPlan (Blum and Furst 1997) during the planning process, but as the number of possible cases grows, these relations become extremely hard to visualize graphically. In this paper, we argue that a concrete representation can help new users to learn by association as they create a uniform representation for research. In order to create a consistent visual language, we take the representation of a boolean function as the starting point from which we develop a visual representation of planning operators.

Logic

Boolean functions in propositional and predicate logic are deterministic and easy to understand through several representations. Complex boolean functions are usually represented by a description of the input and the output separately, without a visualization of how the input is transformed into the output. To better understand the effects of each input different values must be propagated through a circuit-like diagram. Since we are more interested in the actual inputs, instead of the circuit, we need a grounded equivalent, where actual values are made explicit in the notation. Complex operations should be decomposable and may require several inputs. We need a representation that: can be broken into smaller pieces; is able to scale up to n inputs; and represent a metaphor that most users can relate to from previous experience. Metaphors require familiarity, and toy mechanics are easy to relate to different real problems being part of the common childhood experience. The connections and desired scalability are natural parts of jigsaw puzzles, and we use this idea as the main element of our graphical representation. By using one side of the piece as the input and other side as the output we represent all the combinations that match a truth-table. Figure 1 illustrates the similarity between a jigsaw piece and a circuit, by showing the equivalence between $\neg A$ and $\neg(\neg A)$ using the convex and concave joints to explicitly show the logic value. One side is the input or the expected value that allows an operator to be executable, while the other side is the output, limiting which other jigsaw pieces can be connected to it. This notation requires 2^n pieces for n inputs, which limits the technique to small or a partial set of pieces.

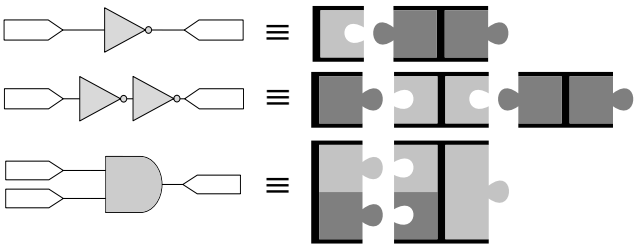

Figure 1: *Equivalence between circuits and joints.*

Planning

Preconditions and effects can be seen as the input and output of actions, but instead of changing the output value based on the input, an action uses the preconditions as constraints that must be satisfied for the effects to be applied. We now present a notation that represents possible values based on a three state logic to account for unreferenced propositions whose value is unknown. Under the *closed world assumption*[3] we assume that such values are not modified by the actions. Our notation is much easier to describe in this context, since the entire jigsaw piece must be connected to trigger an output, this means that no missing values must be present in the state in order to execute an action. We assume that all operators have already been grounded within the current problem, and each proposition is known, thus, we must consider the following cases when dealing with a representation that allows negative-preconditions. We illustrate each of these cases graphically in Figure 2, we consider each case from top to bottom.

- **Unspecified precondition - unspecified effect**: predicate value unchanged by this action, as seen in case 1.
- **Specified precondition - unspecified effect**: we assume that the value expected by the precondition is maintained, as seen in cases 2 and 3.
- **Unspecified precondition - specified effect**: the value becomes the effect, the previous value is discarded, as seen cases 4 and 5.
- **Precondition $=$ effect**: the value expected by the precondition is maintained, as seen cases 6 and 7.
- **Precondition $\neq$ effect**: the value expected by the precondition is toggled, as seen in cases 8 and 9.

Each proposition is always placed in the same fixed line in the graphical representation, and each line represents the dynamics of a single proposition. We can apply colors to better differentiate each element. Highlighting the action scheme with the grounded version using the same color for the propositions creates the visual link between textual and visual representation. Of course that this representation is not perfect, as most visual representations require a large area to show the diagram. A possible solution is to create

[3]Using *Closed World Assumption* it is possible to solve deterministic and totally observable problems, ignoring external effects changing the world.

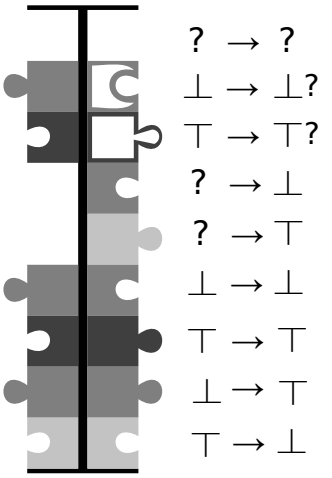

Figure 2: *Possible cases of precondition and effects using* DOVETAIL *notation.*

small problems to teach minimal scenarios in order to explain patterns. The second problem is how to illustrate the consequence of an effect. Some actions do not reference a proposition, which does not mean that the value is lost. The value is maintained and the puzzle joint that created the last value for this proposition must be stretched to connect with further preconditions.

4 Case Study

A concrete representation should be help to connect abstract ideas, in the same way of a dovetail joint is used in the carpentry. Our tool, called DOVETAIL, is based on the idea of abstracting the state description with joints representing each proposition. Actions are represented in the same way, making a consistent relation between preconditions to satisfy, and effects that modify the world. The task of DOVETAIL is to represent the entire plan, making explicit the duration of each proposition. The joints are representing the values in the same sense as the waveforms. Once all those features are represented we can observe the patterns and discuss about features without complex formulas.

We generate the possible propositional state variables from a modified planner that processes PDDL input and exposes its internal data structures. From these possible propositional state variables, we remove those that remain unchanged throughout the planning process to save screen space. We represent the initial state as an action without preconditions, with the effects defining the state, and the goal as an action without effects so that the goal is reached when the action becomes executable. Propositions present in the preconditions and not in the effects are expected to be maintained after the action is applied. When one or more preconditions of an action are not satisfied by the previous state the shapes do not match representing a failure. DOVETAIL supports only actions for this reason, to show how grounded predicates relate in the plan.

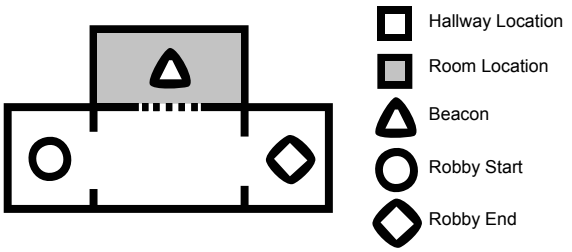

Figure 3: *Rescue Robot Robby problem example.*

Rescue Robot Robby Domain

In order to test the intuitiveness of our planning representation, we modeled a specific domain specification from scratch. We selected a domain called Rescue Robot Robby[4] *Robby* is a robot that walks towards beacons to report the status at the current position. As *Robby* moves through hallway-locations (from/to) it may have to enter and exit room-locations to reach the beacons to fulfil the goal, reporting every beacon, and sometimes reach a safe position for retrieval. Thus, *Robby* can be *at* a particular location at a given point, and only at that location (hallway or room). The beacons can be either in hallway-locations or room-locations.

This domain requires four actions: *move* from hallway to hallway; *enter* in rooms that are connected to hallways; *exit* those rooms to reach a hallway; and *report* beacons at the same location as *Robby*. The problem that we want to solve is represented graphically in Figure 3. This particular problem is small, and there is only one *Robby* that starts at the left hallway, with the beacon at the room connect to the middle hallway. The three movement actions (*move*, *enter*, and *exit*) are similar to the generic one of Listing 1. We use different movements just to differentiate between the possible interactions between rooms and hallways. The *report* action is the only action without a negative effect, as the beacon being reported is the only effect. Although it is straightforward to construct a mental model (i.e, go to each beacon and report, and go to final position) a few details may be lost during modelling. Once *Robby* reports the beacon inside a room, two actions are applicable: *exit* the room and *report* again. Such repeated applications of report would lead to the same state, but the only way for the planner to avoid would be testing the preconditions, applying the effects and checking if this is a visited state. One solution requires adding (*not* (*reported* ?*beacon*)) to the preconditions. This modification describes to the planner how to avoid repeating reported, does not make this description more correct. We are drawing attention to the preconditions to affect performance, which is easier when precondition and effects are side-by-side.

We illustrate one possible plan for the problem of Figure 3 in our representation in Figure 5. If we stretch the effects until their next connection we see the state maintaining each value not explicitly modified by an action as in a *Gantt chart*, this is shown graphically in Figure 6. From the seven

[4]The *Robby* domain was created by Subbarao Kambhampati and Kartik Talamadupula from Arizona State University. http://rakaposhi.eas.asu.edu/cse471/

ground actions (A1, A2, ..., A7) in the Figure 4, note that only two were not used in the plan. At the side of the figures a caption describes the proposition of the line, Listing 2, and at the bottom of each action piece the index according to the grounding, Listing 3.

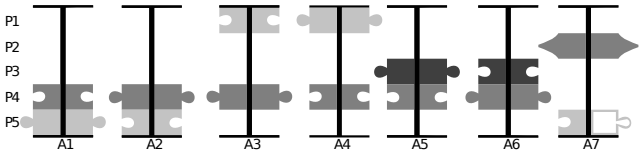

Figure 4: *Set of actions for Robby in this problem.*

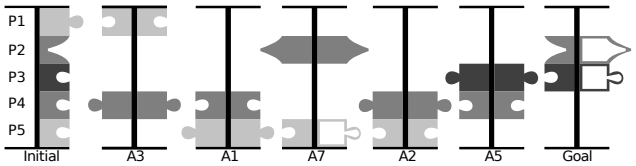

Figure 5: *A possible plan for Robby problem.*

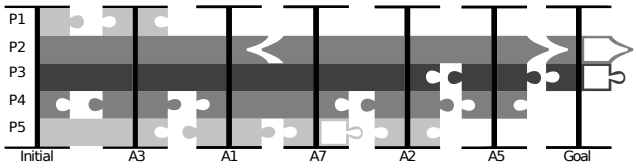

Figure 6: *Plan with effects stretched in time.*

```
P1 - (at robby left)
P2 - (reported robby beacon1)
P3 - (at robby right)
P4 - (at robby middle)
P5 - (at robby room1)
```

Listing 2: *Robby problem grounded propositions*.

```
A1 - (enter robby middle room1)
A2 - (exit robby room1 middle)
A3 - (move robby left middle)
A4 - (move robby middle left)
A5 - (move robby middle right)
A6 - (move robby right middle)
A7 - (report robby room1 beacon1)
```

Listing 3: *Robby problem grounded actions*.

5 Model Validation

In order to validate the model, we emphasize how patterns can be seen and explained through DOVETAIL. These patterns are detected by an internal analyzer and shown by the interface with hints such as a warning, error, or more information about the behavior itself. Below, we detail two specific formalization issues that our tool helps identify:

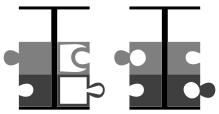

Figure 7: *Useless and Inconsistent action pieces.*

1. A poorly defined operator may generate useless instances (ground actions), which occur if all the effects of one action are explicitly defined in the preconditions of the same action, that is if $(\text{eff}(\mathcal{A})^+ \subseteq \text{pre}(\mathcal{A})^+) \wedge (\text{eff}(\mathcal{A})^- \subseteq \text{pre}(\mathcal{A})^-)$. For example, in the simple operator of Listing 4, when $?a = ?b$ the action becomes useless.

```
(:action make
  :parameters (?a ?b)
  :precondition (and (p ?a))
  :effect (and (p ?b)) )
```

Listing 4: *Useless action*.

Useless actions can only be found with a testing problem for complex cases, where $?a$ and $?b$ are related to more predicates. The inconsistency does not require the problem, it follows a regular pattern that could find and hint the user about the mistake.

2. We consider that an action is inconsistent if any preconditions and effects contain, respectively the same instance of a predicate in positive and negated from, namely, if $((\text{pre}(\mathcal{A})^+ \cap \text{pre}(\mathcal{A})^-) \neq \emptyset) \vee (\text{eff}(\mathcal{A})^+ \cap \text{eff}(\mathcal{A})^-) \neq \emptyset))$. We illustrate a minimal example in Listing 5, which, like the previous example, creates a problem when $?a = ?b$. Since a domain containing this kind of error is impossible to draw using the DOVETAIL notation, it is easy to highlight it for a user. The grounded action would render both joints (convex and concave) for the same proposition. The corresponding dovetail for both cases can be seen at Figure 7.

```
(:action make
  :parameters (?a ?b)
  :precondition (and (p ?a))
  :effect (and (p ?a) (not (p ?b)) ))
```

Listing 5: *Inconsistent action by effect*.

6 Related Work

Most tools for PDDL edition and verification are interested in building a richer domain and problem description. Below, we list some of these tools.

Graphical Interface for Planning with Objects (GIPO) (Simpson, Kitchin, and McCluskey 2007) is a tool for planning domain knowledge engineering that allows the specification of domains in PDDL and *Hierarchical Task Network* (HTN). Besides the domain specification, GIPO provides an animator tool to graphically inspect the output plans produced by the internal planner, from

a domain and problem specification. Unlike DOVETAIL, GIPO checks a set of plans to validate a certain domain and problem specification, indicating whether the domain and problem specification do support the given plans. Similar to DOVETAIL, GIPO also provides an animator tool to visualize how a sequence of actions (i.e, a plan) connects to form a plan that achieves a goal state from an initial state.

VisPlan (Glinskỳ and Barták 2011) is an interactive tool to visualize and verify plans' correctness. This tool is closely related to DOVETAIL in the sense of helping planning users to better understand how a sequences of actions achieve a goal from an initial state. Unlike DOVETAIL, *VisPlan* identifies possible flaws (i.e, incorrect actions) in a plan, allowing users to manually modify this plan by repairing these identified flawed actions.

PDVer (Raimondi, Pecheur, and Brat 2009) is a methodology and tool that verifies if a PDDL domain satisfies a set of requirements (i.e, planning goals). This tool allows an automatic generation of these requirements from a *Linear Temporal Logic* (LTL) specification into a PDDL description. More specifically, this tool is concerned with how the corresponding PDDL action constraints are translated from an LTL specification. Whereas *PDVer* provides a summary of test cases (positive and negative) indicating why a PDDL domain specification does not satisfy a set of requirements to achieve a goal, DOVETAIL graphically shows how a plan achieves (or does not achieve) a goal through state transition.

PDDL *Studio* (Plch et al. 2012) is a PDDL editor in the same sense as imperative language editors with syntax highlighting, code completion, and context sensitive hints specifically designed for PDDL. It is possible to integrate the editor to an external planner to easily create an integrated environment. PDDL *Studio* does not provide any model verification, it provides an IDE, while DOVETAIL aims to aid the learning curve for defining and subsequently diagnosing problems with new planning domains.

itSimple (Vaquero et al. 2012) is concerned with domain modeling, using steps to guide the user from the informal requirements (UML) to the objective representation (Petri Nets). The itSimple features provide a visualization and simulation tool to help understanding planning domains through diagrams. Whereas DOVETAIL uses a set of jigsaw pieces to model a planning instance, *itSimple* uses UML diagrams to model planning instances and Petri Nets for validating planning instances.

7 Conclusion

Planning can be compared to a jigsaw puzzle in which there is no picture and which can either have more pieces than are required for it to be solved, or have a few missing pieces. In fact, the drawing we usually see on top of each piece is the heuristic, guiding our thought process of which pieces make sense to use in order to complete the image without testing the connectivity of each piece. The style of construction is the search itself, either forward (middle to border), backward (border to middle) or bi-directional. This metaphor explains how hard planning is, a puzzle without a guiding image, requiring the user to test if connections are possible all the time until the final piece (the goal) is placed. We can go further and say that it is harder, as some pieces may be repeatedly used. Rather than deal just with syntax highlighting and code completion, DOVETAIL provides syntactic checks of the domain specification, and allows users to verify why their specification does not generate a plan (e.g due to useless predicates or actions). Yet, there are several theoretical and technical challenges to be solved in order to improve DOVETAIL. So far, DOVETAIL is an exploratory tool, making the process of learning planning easier. Our goal is not to obviate the use of PDDL, but rather to relate it to something that we can visualize. Being able to modify both the textual PDDL and the pieces of DOVETAIL is the key to share and play with ideas. DOVETAIL currently accepts only the STRIPS fragment of PDDL with negative preconditions and typing, but as future work we aim to add an editor to modify, visualize and export new descriptions.

Acknowledgement: We thank *Conselho Nacional de Desenvolvimento Científico e Tecnológico* (CNPq) for partial support under process number 400316/2014-5.

References

Ainsworth, S.; Prain, V.; and Tytler, R. 2011. Drawing to Learn in Science. *Science Magazine - Education* 3:5.

Blum, A., and Furst, M. L. 1997. Fast planning through planning graph analysis. *Artificial intelligence* 90(1-2):281–300.

Fikes, R. E., and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial intelligence* 2(3):189–208.

Gelfond, M., and Lifschitz, V. 1998. Action Languages. *Electronic Transactions on AI* 3:195–210.

Ghallab, M.; Nau, D. S.; and Traverso, P. 2004. *Automated Planning - Theory and Practice.* Elsevier.

Glinskỳ, R., and Barták, R. 2011. Visplan–interactive visualisation and verification of plans. *Proceedings of the Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)* 134–138.

Ha, T. T. 2010. *Theory and design of digital communication systems*. Cambridge University Press.

Malan, D. J., and Leitner, H. H. 2007. Scratch for budding computer scientists. *SIGCSE Bull.* 39(1):223–227.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL − The Planning Domain Definition Language. *Technical Report – Yale Center for Computational Vision and Control*.

Plch, T.; Chomut, M.; Brom, C.; and Barták, R. 2012. Inspect, edit and debug PDDL documents: Simply and efficiently with PDDL Studio. In *Proceedings of ICAPS'09*, 15–18.

Raimondi, F.; Pecheur, C.; and Brat, G. 2009. PDVer, a Tool to Verify PDDL Planning Domains. In *Proceedings of ICAPS'09 Workshop on Verification and Validation of Planning and Scheduling Systems, Thessaloniki, Greece*.

Simpson, R. M.; Kitchin, D. E.; and McCluskey, T. L. 2007. Planning domain definition using GIPO. *Knowledge Eng. Review* 22(2):117–134.

Vaquero, T.; Tonaco, R.; Costa, G.; Tonidandel, F.; Silva, J. R.; and Beck, J. C. 2012. itSIMPLE 4.0: Enhancing the modeling experience of planning problems. In *Proceedings of ICAPS'12*, 11–14.

Wilson, J. M. 2003. Gantt charts: A centenary appreciation. *European Journal of Operational Research* 149(2):430 – 437.