Automated Planning and Robotics Simulation with PDSim

Emanuele De Pellegrin, Ronald P. A. Petrick
Edinburgh Centre for Robotics
Heriot-Watt University
Edinburgh, Scotland, United Kingdom
{ed50,R.Petrick}@hw.ac.uk

Abstract

This paper presents the current state of development and plans for the PDSim system. PDSim is an external tool that can be installed on the Unity game engine adding support for the simulation of classical plans using 3D animations and visualization methods defined by the user. New additions planned for PDSim aim to enable users to intuitively define animations without the need to learn a new scripting language and connect with the Robotic Operating System (ROS) to unlock the potential use of the simulation framework for automated planning problems in robotics.

Introduction

The task of modelling planning domains and verifying plan solutions can be quite challenging, especially for real-world planning problems. While representation languages like PDDL (McDermott et al. 1998) provide a standard way of representing planning models supported by a range of planners, it can be difficult to catch modelling errors due to the complexity of the knowledge that needs to be specified (e.g., definitions of state properties, actions, and objects) and the level of abstraction that is often required for ensuring the generation of tractable solutions.

Although several tools do exist to aid in the validation of planning domain models (e.g., VAL (Howey and Long 2003)), and formal plan verification methods are a growing area of research (Bensalem, Havelund, and Orlandini 2014; Cimatti, Micheli, and Roveri 2017; Hill, Komendantskaya, and Petrick 2020), approaches based on visualisation methods and visual feeedback can also play an important role in addressing the problem of correctly modelling planning domains. Visual tools can also serve as an environment for displaying, inspecting, and simulating aspects of the planning process, which can aid in plan explainability for human users (Fox, Long, and Magazzeni 2017).

PDSim (De Pellegrin 2020) introduced a system to visualise and simulate plans for classical planning problems defined in PDDL. While visualisation of planning domains and plan solutions is not a new idea (Vrakas and Vlahavas 2005; Vaquero et al. 2007; Chen et al. 2020; Tapia, San Segundo, and Artieda 2015; Le Bras et al. 2020), PDSim approaches the problem by building a graphical environment for plan visualisation and simulation within the Unity game engine (Unity Technologies 2020). PDDL is used to define the structure of the domain knowledge and the problem formulation (e.g., planner requirements, language models used in the domain, plus standard definitions of the domain and problem). These components are used by a planner to check that a solution exists and to generate a list of actions (a plan) that satisfies the goals. Using the plan, PDSim interprets the action effects as 3D animations and graphics effects to deliver a visual explanation of the world and its actions during plan execution. In particular, the main "actors" involved in PDSim simulations are the properties responsible for defining the initial state of the world and the action effects.

In this paper, we describe new additions to PDSim that improve the quality of the simulation and provide support for integrating PDSim more widely (e.g., in robotics applications). While the original version of PDSim focused on visualising actions during run time in a 3D environment, the PDDL parsing interface was very weak as it only supported a small set of PDDL features. This meant that if the user didn't modify the domain and problem appropriately, errors could have occurred in the simulation. This interface has now been improved to support a wider range of PDDL language features. Several extensions have also been implemented to improve the mechanism for creating custom animations. For instance, the old version of PDSim only supported the C# scripting language and required familiarity with the standard flow of the Unity game engine. A new animation system has now been implemented based on the xNode framework, enabling more flexibility for plan animation. Finally, support for the Robot Operating Sytem (ROS) (Quigley et al. 2009) has also been added, offering a way to connect and publish custom messages to ROS, enabling the use of PDSim as a visualization tool for robotics and similar applications.

The rest of the paper is organised as follows. First, we review related work and provide a brief overview of the PDSim system. We then describe recent additions to PDSim with a technical overview of the components and examples of their use. We conclude with future work and a discussion of the PDSim development roadmap.

Related Work

PDSim (De Pellegrin 2020) is part of the small ecosystem of simulators for automated planning which use visual cues and animations to translate the output of a plan into a visual environment. The closest approach to ours is Planimation (Chen et al. 2020) which uses the Unity game engine as the front end to display objects and animate their position while following a given plan. Animations are defined using an ad-hoc language (namely an animation profile) similar to PDDL. PDSim aims to remove this additional step with its new animation system that provides a more intuitive system for users (see Figure 6, discussed in greater detail below).

The Logic Planning Simulator (LPS) (Tapia, San Segundo, and Artieda 2015) also provides a planning simulation system that represents PDDL objects with 3D models in a user-customizable environment. The approach is integrated with a SAT-based planner and a user interface that enables the execution of a plan to be simulated while visualising updated to the state of the world and individual PDDL properties in the 3D environment. Unlike PDSim and Planimation, LPS is not based on Unity but provides its own user interface for plan visualisation. Several user-specified files are also required to define 3D object meshes, the relationship between PDDL elements and 3D objects, and the specific animation effect to be produced.

Several systems also exist to help users formalize planning domains and problems through user-friendly interfaces. For instance, systems like GIPO (Simpson, Kitchin, and McCluskey 2007), ItSimple (Vaquero et al. 2007) and VIZ (Vodrázka and Chrpa 2010) use graphical illustrations of the domain and problem elements, removing the requirement of PDDL language knowledge, to help new users approach planning domain modelling for the first time. Other software such as Web Planner (Magnaguagno et al. 2017) and Planning.Domains (Muise 2016) use Gantt charts or tree-like visualisation methods to illustrate the generated plan and the state space searched by a particular planning algorithm. PlanCurves (Le Bras et al. 2020) uses a novel interface based on time curves (Bach et al. 2015) to display timeline-based multiagent temporal plans distorted to illustrate similarity between states. All of these tools attempt to assist users in understanding how a plan is generated and to help detect potential errors in the modelling process.

Simulators using a game engine such as MORSE (Echeverria et al. 2011) or Drone Sim Lab (Ganoni and Mukundan 2017) are also prevalent in robotics applications. A game engine offers benefits like multiple cameras to follow the simulation, a physics engine and realistic post-processing effects with no need to implement them from scratch (Ganoni and Mukundan 2017). PDSim is built by extending the Unity game engine editor (Unity Technologies 2020) and using the components offered by the engine such as a path planner, lighting system, and scene management, among others.

Planning Domain Simulation with PDSim

PDSim has been developed to work as an external module for the Unity game engine, a 2D and 3D game engine widely known and used in the video game industry. Unity offers a customisable GUI editor and many available components such as a built-in physics engine, realistic shaders and materials for 3D models, and a path planning library, to rapidly prototype a project. Thanks to its modularity, it is possible to extend the interface to fit user needs, for instance by defining custom animations for PDDL objects or by extending existing animations to reuse models from different simulations without creating them from scratch every time.

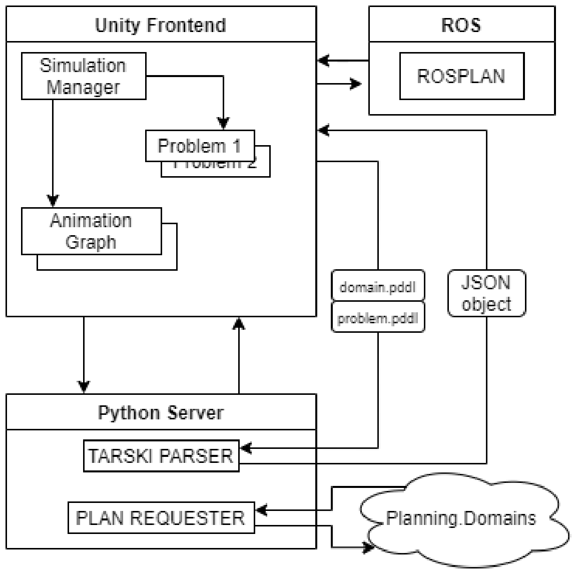


Figure 1: PDSim system architecture.

Every new simulation in PDSim creates a Unity scene[1], an essential component of the editor in which all objects and components are stored and managed. When a new simulation is created, the PDDL parser component creates a copy of the elements defined in the PDDL domain and problem files that can be used with the C# scripting language (the object-oriented programming language in Unity). In PDSim, all of the PDDL elements (e.g., types, predicates, actions, etc.) have a corresponding class representation in C#. The user can customise the simulation in different ways, for example, to let it use the path planning component for the movement instead of a basic translation animation, randomise the colours of a model, play sounds, spawn a particle system, and so on. During plan generation, the simulation manager looks in the simulation environment object to see if a plan is already available. If the plan is not saved locally, the simulation manager will send a request to a Python server to generate a plan using the cloud planning service (currently Planning.Domains (Muise 2016)) for the domain and problem that are stored locally on the server (De Pellegrin 2020).

A general overview of the system architecture is given in Figure 1, showing the individual core components and the communication between components through a network. The Unity front-end can send requests or connect either to the Python server (the back-end of PDSim) or to the Robot Operating System (ROS) (Quigley et al. 2009), the popular framework for robotics applications.

Examples of plan execution using PDSim are illustrated in Figures 2 and 3. Figure 2 shows an example of a stacking animation from the Blocks World domain, where blocks are

[1]See https://docs.unity3d.com/Manual/CreatingScenes.html

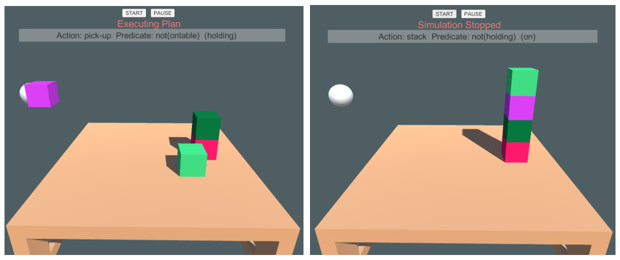

START
PAUSE
Executing Plan
Action: pick-up Predicate: not(ontable) (holding)
START
PAUSE
Simulation Stopped
Action: stack Predicate: not(holding) (on)

Figure 2: Blocks World plan execution.

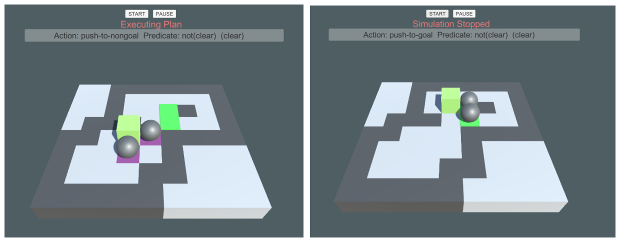

START
PAUSE
Executing Plan
Action: push-to-nongoal Predicate: not(clear) (clear)
START
PAUSE
Simulation Stopped
Action: push-to-goal Predicate: not(clear) (clear)

Figure 3: Sokoban plan execution.

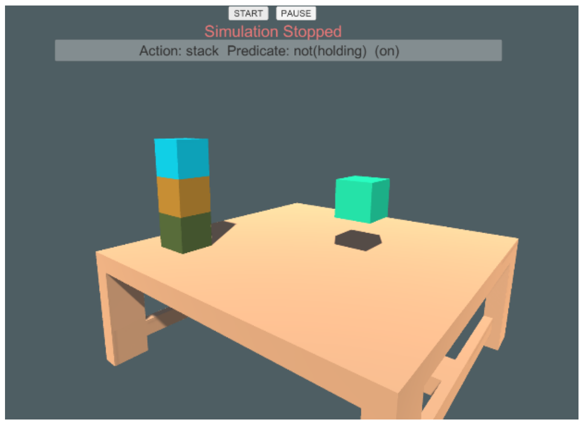


Figure 4: Plan validation check in PDSim.

stacked during plan execution, while Figure 3 shows stones being moved by the player in the Sokoban domain.[2] Provided a plan can be generated, PDSim will attempt to animate the plan during execution. This includes plans where modelling errors may be present in the original PDDL domain or problem files. For instance, Figure 4 shows an example from the Blocks World domain where the *stack* action has been modified to not set the target block as *not clear* when stacking another block on top of it. Regardless of the error, a plan is generated and the simulator can animate it. However, in this case the PDSim simulation has rendered the block as being stuck in mid-air, thus indicating a possible error in the domain formulation.

[2]The plan execution examples in Figures 2-4 are taken from (De Pellegrin 2020). We refer the reader to that paper for more information on the simulation of these domains in PDSim.

```
{
’objects’:
    [{’name’: ’apn1’, ’type’: ’airplane’},
     {’name’: ’apt1’, ’type’: ’airport’},
     {’name’: ’apt2’, ’type’: ’airport’},
     ...],
’predicates’:
    [{’name’: ’in-city’,
      ’attributes’: [’place’, ’city’]},
     {’name’: ’at’,
      ’attributes’: [’physobj’, ’place’]},
     {’name’: ’in’,
      ’attributes’: [’package’, ’vehicle’]}],
     ...
}
```

Figure 5: JSON output from PDDL parser.

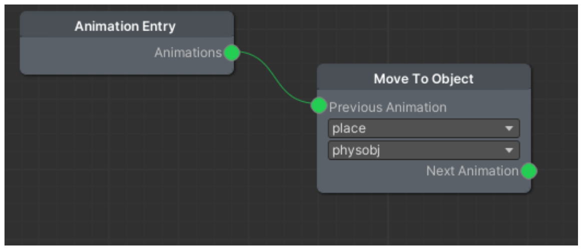


Figure 6: New animation definition system.

Extending the PDSim System

We now describe the major improvements to PDSim, which include a new and robust PDDL parser, a new animation definition system, and interface support with ROS.

Parser

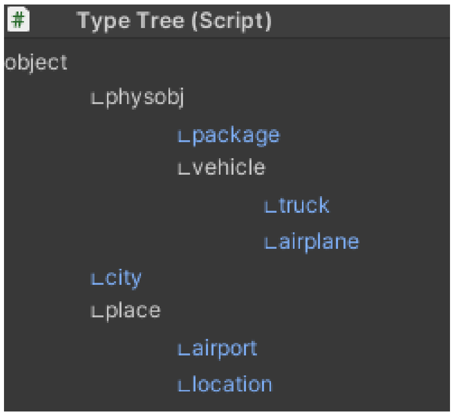
PDSim now offers a client-server connection with an external python server using ZeroMQ[3]. This unlocked the possibility of using several packages and libraries available with python and integrating them in the Unity engine communication on an internal network. In particular, the Tarski library[4] is used to parse the PDDL domain and problems files and produce a Javascript Object Notation (JSON) version. An example of the parsed output used in PDSim is shown in Figure 5. The new parser now enables PDSim to support a more comprehensive set features of the PDDL language (primarily version 1.2, plus some additional features of higher PDDL versions) and doesn't require the user to modify the domain to follow the previous PDDL constraints. Tarski is well supported by numerous researchers and constantly updated, providing a useful tool for future additions to PDSim.

Animation System

PDSim's new animation system has been developed to be much more intuitive, using nodes and graphs to connect

[3]See https://github.com/zeromq.

[4]See https://github.com/aig-upf/tarski.


Type Tree (Script)

```
object
    ⌊physobj
        ⌊package
        ⌊vehicle
            ⌊truck
            ⌊airplane
    ⌊city
    ⌊place
        ⌊airport
        ⌊location
```

Figure 7: New type representation system.

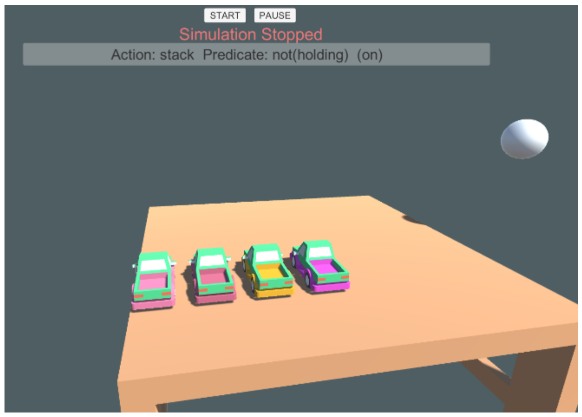


Figure 8: Animation example for Blocks World.

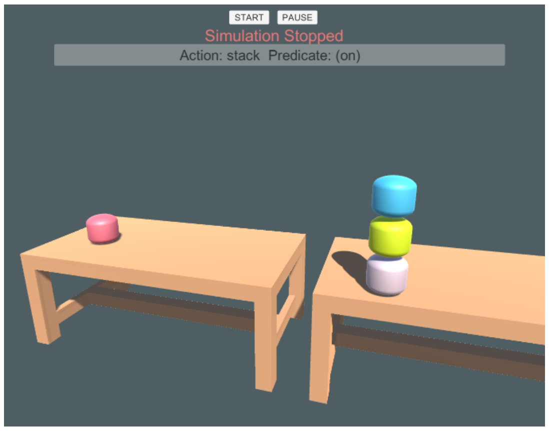


Figure 9: Animation example with modified domain.

components that are related to the animation. Figure 6 shows an example of an animation definition for the predicate *at* taken from the Logistics domain. Nodes can be connected as user preferences. For instance, in the example, the graph is intended to capture the idea that when the animation starts, it should animate the translation (*Move To Object*) of the *place* to the *physobj* objects. The new animation system in PDSim is an implementation of the xNode framework[5] for Unity. This framework allows the use of a node system that was modified for PDSim. For each predicate defined in the domain, the *AnimationNodeGraph* is assigned and the user can customise the animation by inserting new nodes as illustrated in Figure 6. During plan execution, if a predicate is in the effects list of the action being executed, the animation graph is loaded and every animation node is played. For example, in the Blocks World domain, the predicate *on* might have an animation graph that specifies a *Move To Object* animation node to move the first block on top of the second. When the action *stack* is called the block is moved accordingly to reflect the animation graph specified.

ROS interface

Robotics applications are a common application domain for automated planning tools. The Unity engine is also becoming an important tool for roboticists to help simulate robots and robot deployment environments (Green et al. 2020). ROS is widely used as a robotics system and ROSPlan (Cashmore et al. 2015) is the main framework for automated planning and robotics. As a result, PDSim has been extended to support communication with ROS and its packages using the ROS-TCP-Connector library[6] for Unity. This library now makes it possible to set up a simulation and send custom messages to ROS, or to subscribe to ROS topics.

Examples

Figure 7 shows how types are now handled in PDSim. After parsing the PDDL domain, a tree data structure of types is created to simplify the task of checking if a type is a sub or super-type during plan execution. Leaf types (highlighted in blue in the example) are the PDDL objects responsible for animations and 3D visualization in PDSim. The type tree helps running animations identify a type and all its subtypes. For example, in the Logistics domain the predicate *at* requires a *package* type and a *vehicle* type. From the type tree, we can get that both *truck* and *aeroplane* are vehicles so that an animation that uses a vehicle type will animate both of these two sub-types.

Figure 8 shows an example of how PDSim can be highly customisable. For instance, in the Blocks World domain the user can set a preferred model representing the *block* type (e.g., a car). The example also shows how the animation of predicates can be customised: the *on* predicate has been defined to stack the objects on the left side rather than on top of each other.

Finally, Figure 9 illustrates the flexibility of some of the visualisation features in PDSim. In this example, the Blocks World domain has been modified to introduce a *table* type. The user can set the model for this type and PDSim can reuse the animations defined for the non-modified domain to successfully display the plan simulation.

[5]See https://github.com/Siccity/xNode.
[6]See https://github.com/Unity-Technologies/ROS-TCP-Connector.

Conclusion

This paper described a set of recent additions to PDSim, a simulation system for PDDL that can be used to animate classical plans. Additions to the basic PDSim system included a new parser supporting a more comprehensive set of PDDL language features, a new animation system, and the ability to communicate with the Robot Operating System (ROS). Future work on the system aims to provide support for temporal planning and epistemic planning models. A web version of PDSim is also currently in development to interface with the Planning.Domains (Muise 2016) web editor as an internal plugin. As a sub-project from PDSim, a planner for the Unity engine is also in development to be used for game development, which will help in the development of complex AI gaming systems.

References

Bach, B.; Shi, C.; Heulot, N.; Madhyastha, T.; Grabowski, T.; and Dragicevic, P. 2015. Time curves: Folding time to visualize patterns of temporal evolution in data. *IEEE transactions on visualization and computer graphics* 22(1): 559–568.

Bensalem, S.; Havelund, K.; and Orlandini, A. 2014. Verification and validation meet planning and scheduling. *International Journal on Software Tools for Technology Transfer* 16: 1–12.

Cashmore, M.; Fox, M.; Long, D.; Magazzeni, D.; Ridder, B.; Carrera, A.; Palomeras, N.; Hurtos, N.; and Carreras, M. 2015. ROSPlan: Planning in the Robot Operating System. In *Twenty-Fifth International Conference on Automated Planning and Scheduling*.

Chen, G.; Ding, Y.; Edwards, H.; Chau, C. H.; Hou, S.; Johnson, G.; Sharukh Syed, M.; Tang, H.; Wu, Y.; Yan, Y.; Gil, T.; and Nir, L. 2020. Planimation. doi:10.5281/zenodo.3773027. URL https://doi.org/10.5281/zenodo.3773027.

Cimatti, A.; Micheli, A.; and Roveri, M. 2017. Validating domains and plans for temporal planning via encoding into infinite-state linear temporal logic. In *Proceedings of AAAI*, 3547–3554.

De Pellegrin, E. 2020. PDSim: Planning Domain Simulation with the Unity Game Engine. In *Proceedings of the ICAPS Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*.

Echeverria, G.; Lassabe, N.; Degroote, A.; and Lemaignan, S. 2011. Modular open robots simulation engine: Morse. In *2011 IEEE International Conference on Robotics and Automation*, 46–51. IEEE.

Fox, M.; Long, D.; and Magazzeni, D. 2017. Explainable Planning. In *Proceedings of the IJCAI Workshop on Explainable AI*.

Ganoni, O.; and Mukundan, R. 2017. A framework for visually realistic multi-robot simulation in natural environment. *arXiv preprint arXiv:1708.01938* .

Green, C.; Platin, J.; Pinol, M.; Trang, A.; and Vij, V. 2020. Robotics simulation in Unity is as easy as 1, 2, 3! URL https://blog.unity.com/technology/robotics-simulation-in-unity-is-as-easy-as-1-2-3.

Hill, A.; Komendantskaya, E.; and Petrick, R. P. A. 2020. Proof-Carrying Plans: A Resource Logic for AI Planning. In *International Symposium on Principles and Practice of Declarative Programming (PPDP)*, 1–13.

Howey, R.; and Long, D. 2003. VAL's Progress: The Automatic Validation Tool for PDDL2.1 used in the International Planning Competition. In *Proceedings of the ICAPS Workshop on The Competition: Impact, Organization, Evaluation, Benchmarks*.

Le Bras, P.; Carreno, Y.; Lindsay, A.; Petrick, R. P. A.; and Chantler, M. J. 2020. PlanCurves: An Interface for End-Users to Visualise Multi-Agent Temporal Plans. In *Proceedings of the ICAPS Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*.

Magnaguagno, M. C.; Fraga Pereira, R.; Móre, M. D.; and Meneguzzi, F. R. 2017. Web planner: A tool to develop classical planning domains and visualize heuristic state-space search. In *ICAPS Workshop on User Interfaces and Scheduling and Planning (UISP)*.

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 CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

Muise, C. 2016. Planning.domains. ICAPS System Demonstration.

Quigley, M.; Conley, K.; Gerkey, B. P.; and Faust, J. 2009. ROS: An Open-Source Robot Operating System. In *Proceedings of the ICRA Workshop on Open Source Software*.

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

Tapia, C.; San Segundo, P.; and Artieda, J. 2015. A PDDL-based simulation system. In *Proceedings of the IADIS International Conference Intelligent Systems and Agents*.

Unity Technologies. 2020. Unity. URL https://unity.com.

Vaquero, T. S.; Romero, V.; Tonidandel, F.; and Silva, J. R. 2007. itSIMPLE2.0: An Integrated Tool for Designing Planning Domains. In *Proceedings of ICAPS*, 336–343.

Vodrázka, J.; and Chrpa, L. 2010. Visual design of planning domains. In *Proceedings of ICAPS Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*, 68–69.

Vrakas, D.; and Vlahavas, I. 2005. A Visualization Environment for Planning. *International Journal of Artificial Intelligence Tools* 14(6): 975–998.