A PDDL-BASED SIMULATION SYSTEM

Cristóbal Tapia, Pablo San Segundo and Jorge Artieda

ABSTRACT

This paper presents a planning simulation system that allows automated planning practitioners to experiment with planning domains, defined using the PDDL language. The proposed system has a modular architecture including a SAT planner (PDDL parser, SAT encoder and SAT solver), a plan simulator and a 3D view user interface. Most of these modules can be changed with other with similar purpose.
This system allows users to test and validate PDDL domains and solutions, using an attractive user interface that includes the simulation of the execution of the plan on a 3D view as an animation.

1. INTRODUCTION

This paper presents Logic Planning Simulator (LPS), a planning simulation system that allows automated planning practitioners to experiment with planning domains, defined using the PDDL language (McDermott et al. 1998) (Fox & Long 2003) (Helmert 2008). The system can run and test easily some simple planning problems. Moreover, the plan execution can be viewed on visual 3D scenery, and both the 3D objects and scene are fully configurable.

The authors approach is classical planning using SAT planning, in which the planning solver is based on a propositional satisfiability (SAT) solver and its SAT compiler, to convert from a PDDL model to a propositional logic formulation. Currently, LPS is limited to STRIPS + typing requirements, but ongoing research will add more PDDL options to our system. Despite these limitations, this approach is useful on many planning problems. SAT planners can be very efficient (Weld 1999), as shown on Sat-Plan (Kautz & Selman 1992), (Kautz et al. 2006), DSatz (Iwen & Mali 2002), BlackBox (Kautz & Selman 1999), Madagascar (Rintanen 2011) among other high-performance planners (Linares et al. 2015).

SAT planners require a conversion from PDDL to propositional logic, this is usually called SAT encoding or compilation. This is a subject for study because depending on the encoding, a huge number of logic variables and clauses can be generated, preferably by an automatic method (Ernst et al. 1997)(Wehrle & Rintanen, 2007) (Björk 2009) (Helmert 2009) (Huang et al. 2010), and some can be redundant (Sideris & Dimopoulos, 2010). SAT planners include this feature, as part of the PDDL input process. General purpose SAT solvers as miniSAT (Een & Sörensson 2006), can also be used for planning problems if a SAT compiler is provided (Gomes et al. 2008).

As part of the system, we have included an user interface that shows a 3D scene with 3D objects representing objects in the model (defined using PDDL files), and can execute a plan solution step by step, showing the state of the world on the 3D scene and the list of predicates that hold at each step.

This planning simulation system is part of an intelligent agent architecture we are developing for a service-oriented mobile robot. The architecture includes a knowledge base and world models, and is able to simulate plans conforming to user defined tasks and goals before actually executing the plan. We note that the architecture may be extended to include learning, following the ideas found in other architectures (McNeill & Bundy 2007) (Jiménez et al. 2008) (Jiménez et al. 2009) (Guzman et al. 2011) (Guzman et al. 2012).

This paper describes a new Logical Planning Simulator (LPS) as a contribution for the PDDL practitioners community. LPS allows for easy online validation and testing of logical planners and includes a graphical representation tool. It fills and important gap for practicioners and researchers alike, since we are not aware of a similar system in literature.

This paper is organized as follows: Section 1 presents motivation and state-of-the-art. Section 2 further describes the architecture of the system in detail. Finally Section 3 presents some conclusions and future work.

2. PLANNING SIMULATION SYSTEM

This paper presents LPS (Logic Planning System), a planning simulation system for robot applications (and many other purposes) that allows automated planning practitioners to use PDDL formulated planning domains run on visual 3D scenery and test easily some simple planning problems. This system is part of an intelligent agent architecture, and supports its development and testing.

2.1 Automated Planning

Automated Planning is a part of Artificial Intelligence, that states the problem of selecting a course of action that reach a goal. This work is based on classical planning, on which some restrictions apply on the world model, so the planning problem can be clearly defined and solved using a logical approach. (Ghallab et al. 2004) (Russell &Norvig 2009). Formally this is defined as planning task as $P = (V, A, s_0, s_g)$ where:

- $V= \{v_1,.., v_n\}$ is a set of finite domain state variables
- A is a set of actions, each is a pair (pre_a, eff_a) , of partial variable assignments called preconditions and effects
- S_0 is a complete variable assignment called initial state
- S_g is a partial variable assignment, the goal

Classical planning is based on a world model with perfect information, deterministic instant actions, a unique initial state and we suppose there is a single agent on the world.

A plan is a sequence of actions that lead from the initial state to a goal state. An action *a* can be applied on a state S_n if pre_a complies with S_n. Then the effects of eff_a over S_n will change the current state to another state S_{n+1}. A plan execution is the successive application of actions in the plan sequence over states, from the start state until a goal state is reached.

PDDL is the most used automated planning language, to define general planning problems, and it is the ICAPS official language for the planning competition. Successive versions have been created, as more features have been asked for by competing planners.

2.2 Simulator Modules

The planning system developed has some parts or modules, in a way that almost all of them can be replaced by other alternative modules with the same function and interface, and so we can test performance and verify correctness of each module.

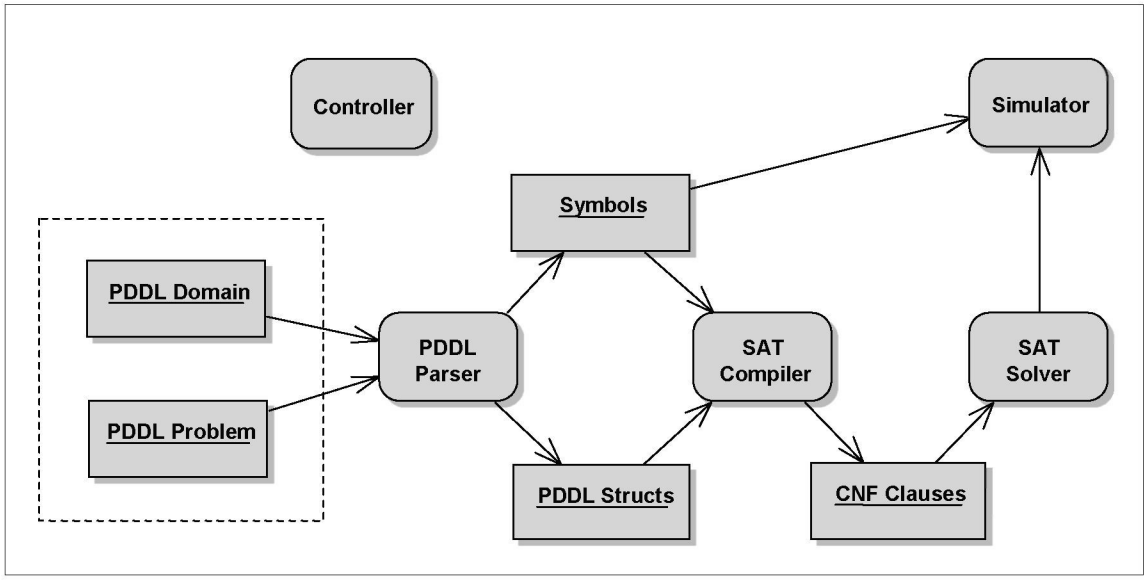

Figure 1. Main modules on the Logic Planning Simulator system

The planning system we have developed has some modules and main data structures as shown on Figure 1. The language used to define the planning problem is PDDL, so there is a PDDL Parser that reads a PDDL domain file and a PDDL problem file and produces a set of structures. These structures are used by the SAT Compiler, a module that generates a logic formulation of the planning problem, as a CNF formatted file. A SAT solver reads the CNF file and will output a solution if it exists or a label indicating unsatisfiability .

The Simulator module can decompile a solution from the SAT Solver module, and show the sequence of actions (as defined on the PDDL Domain) that reach the goal. The result can be seen on 3D scenery step by step on a graphical user interface.

There is a Controller module that controls the execution of the different modules. This Controller is responsible of calling the SAT Compiler and SAT Solver a number of times, as planning using SAT requires explicitly encoding the maximum time or maximum number of steps of the plan.

2.3 PDDL Parser

This planning system contains a parser for the PDDL language, written in C++. This parser supports a subset of PDDL 1.2 (STRIPS + typing), but our purpose is to enhance the parser to include all features of PDDL 3.1 (current version).

PDDL defines two kinds of files: domain and problem. A domain file declares the predicates, functions and types of objects available on the world and the actions available. A problem file describes the state of the world at the start and the goal the planner would meet. This way, domain files are usable for many different situations.

The PDDL parser has been developed so it can handle a couple of files (domain and problem), or parse only one of them. Anyway, the domain or problem are parsed and stored as convenient structures that can then be used to explore, analyze or translate the problem.

Main structures are directly adopted from PDDL definition, so there are appropriate structures for actions (including its parameter list, preconditions and effects), predicates and type hierarchy of objects.

The parser builds, as part of these structures, a symbol table that contains an entry for each identifier used on the domain/problem PDDL declaration. Each entry contains information about the identifier (name) used on the PDDL file, its type (action, predicate, function, object) type of object, and other information depending of the type of the identifier. The next figure shows the classes used to store the PDDL definitions.

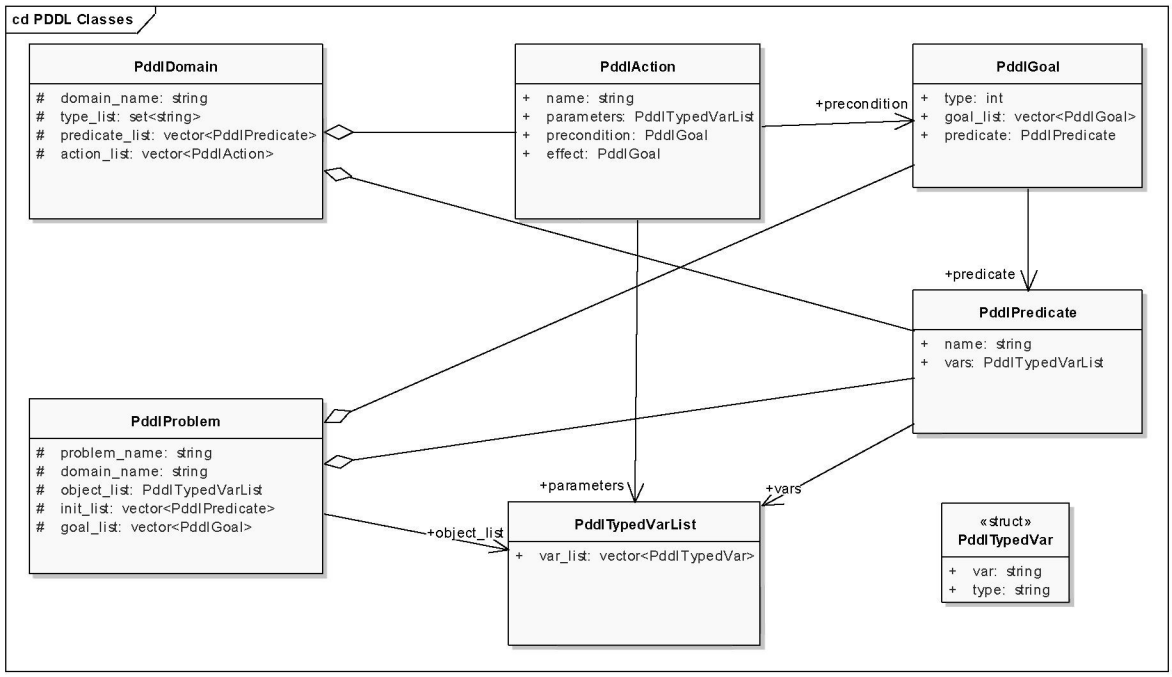

Figure 2. Diagram with PDDL related classes

During the development of the parser, we tested different approaches: hand written and a list-parser. The latter is based on the fact that PDDL is a well structured language, with a Lisp-like syntax. Both parsers generate the same objects and structure, and similar performance.

2.4 PDDL to SAT Compiler

Since SAT-Plan (Kautz & Selman 1992) appeared, using a SAT solver to obtain a solution for a planning problem has been employed by many researchers with good results. As hand-crafting the logic expressions for a planning problem can be difficult and error prone, some compilers for automatic transformation have been documented (Kautz & Selman 1996) (Ernst et al. 1997)(Robinson et al. 2007)(Helmert 2009) (Sideris & Dimopoulos 2010), some encodings are more compact than other (have less variables and/or clauses) and can be solved faster by a SAT solver.

The compiler we have developed accepts PDDL 1.2 with STRIPS actions, and typed objects. Typing has been introduced because it can fit well on PDDL descriptions of domains, and can significantly reduce the generation of grounded actions.

One of the main concerns of a SAT compiler is the number of variables and clauses generated. These numbers can be very high when grounding the actions on the domain. Some techniques are applied to reduce the instantiation of predicates on the actions, for example, the compiler can detect static predicates (predicates that do not change on the course of a plan, because they are not present on the effect part of the actions), and use them as kind of type. The actions are usually encoded as one of this types: regular, simple split, overloaded split, and bitwise encoding. The simple operator splitting has been used for action representation, as this produces fewer variables than standard representation and it is still possible to have concurrent (different) actions. The encoding uses explanatory frame axioms. There is the option for generating complete mutually exclusive action clauses (that prevents concurrent actions on the plan), or conflict exclusive action clauses, more compact and so the plan can contain concurrent actions. Also, the SAT compiler has been implemented using C++ for performance reasons.

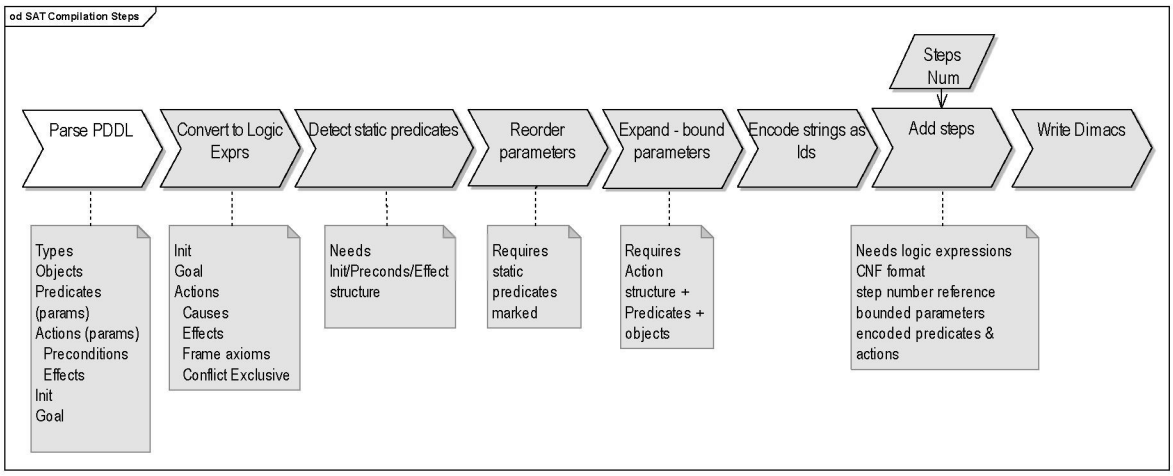

Figure 3. Steps on the SAT Compiler

The SAT Compiler follows some steps as shown in Figure 2. After PDDL has been parsed, effects and preconditions on actions are simplified to logic expressions. Then, there is a step to detect static predicates (Helmert 2009), that will simplify clauses generation later on the process. There is an optional step consisting of reordering the parameters on its declaration, which helps to reduce the number of grounded instances of an action. After that, action parameters are grounded, and then encode all generated actions as numbers, keeping this relation on a symbol table. The last steps on this process are duplicating the clauses and logic variables that represent actions and predicates, given a number of time steps used as planning maximum number of steps.

2.5 SAT Solver

As part of the planning system, a SAT solver has been included. This solver is based on the use of a bit-board library and an exact algorithm, and has high performance when working with bits and logic clauses and has already been used for SAT [SanSegundo 2008]. The performance of this solver is not as good as other award winning SAT solvers, as some SAT techniques have not been used on our implementation, but we expect to reach a comparable performance on future works. This solver is sound and complete.

The solver basically reads a CNF DIMACS file as input, and the output is a set of variables that satisfy the complete formula, if it is possible, or the string "UNSAT" otherwise. This is the standard output specification used on SAT solver competitions.

As the variables are expressed as just numbers, there will be a translation, used the symbol table, so some of these variables represent grounded actions that are the plan solution.

The SAT solver is a standalone executable, so it is possible for an user to replace easily the SAT solver to another one and use it. Besides our SAT Solver implementation, we have tested other SAT Solver as miniSAT (Een & Sörensson 2006).

Although the SAT Solver only needs to solve a problem, is worth to notice that when using SAT planning, the SAT Compiler and the SAT Solver will be called some times, incrementing the number of steps of the plan length (SAT encodings require a fixed number of steps from the start to goal). The Controller on the simulator is the module in charge of calling the compiler and the solver iteratively until a solution is found or when reaches a maximum number of steps, previously defined on the simulator.

2.6 Simulation and Plan Execution

Currently, the Controller module is tightly integrated with the Simulation module, which contains the user's interface and a 3D view. When the user wants to solve a plan, he provides a set of PDDL files with the domain and problem, and the Simulator will call the PDDL parser, SAT Compiler and SAT Solver modules. As we are using SAT planning, the SAT Compiler and SAT solver will be called a number of times, specifying each time the maximum number of steps for the plan. We use a simple incremental step number

heuristic. The SAT Compiler has been developed to have this into account, so it can keep information about the current problem and generate the logic formulation faster as it is called on successive calls.

After a correct solution has been found, it is necessary to convert the SAT solution (a set of variable numbers) to a plan solution (SAT decompilation). The Symbol table provides all the information required so the Simulator can perform the actions that comprise the plan solution, including its grounded parameters and time step.

The Simulator shows a 3D view of objects on the world, and may then execute the plan both as an animation or step by step sequence. Moreover, the simulator also shows concurrent plans, i.e. plan solutions that contain more than one action on the same time step.

The simulator also shows a list of the state variables that hold on the current state.

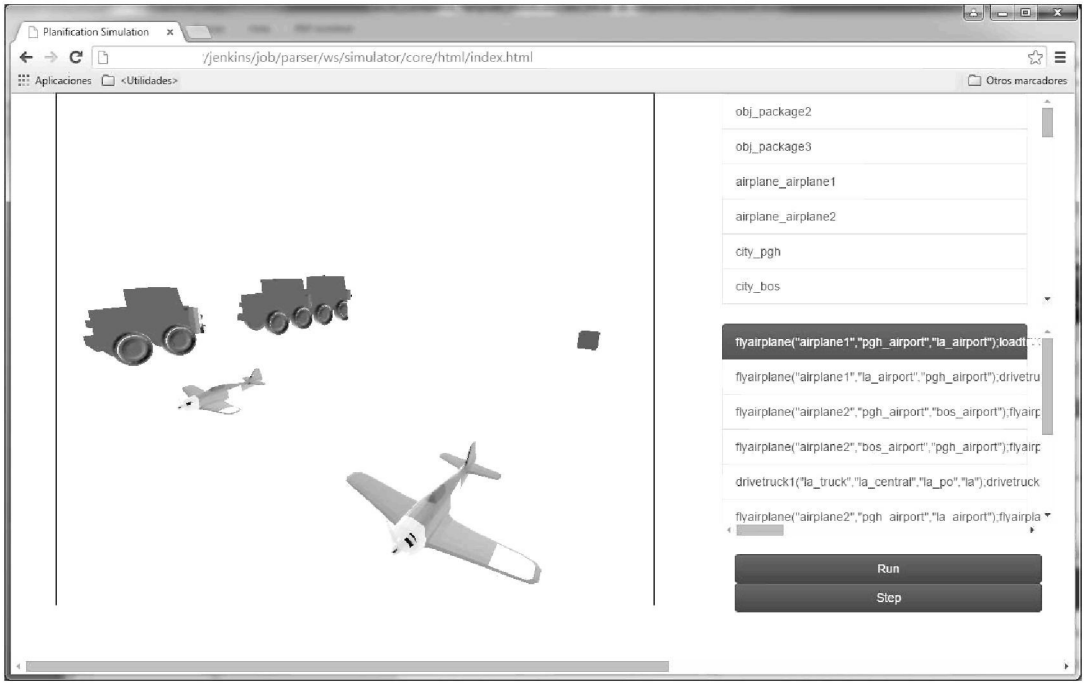

Figure 4. Screen capture of the simulator running on a web browser

For the 3D scene, the simulator requires additional files. It supports .obj format files as 3D object meshes. There is also a file (JSON format) that specifies relations between the PDDL domain model and the 3D objects that the user wants to use on the scenery view. This file also specifies what effect (animation, position change, changes on the world, etc.) will produce an action on the world.

3. CONCLUSION

We have presented LPS, a new planning simulator that accepts STRIPS planning problems on PDDL language. This system can be used to test PDDL domains and problems and show a 3D scene of the world model, and/or run a plan solution step by step.

The system can also be used to test SAT solvers, SAT encodings or planners (replacing the module on the system by another provided by the user) on a complete environment.

The system has a modular architecture that allows the user replace one module with another one (planner, SAT encoder/decoder, SAT solver).

The model definition and world representation in 3D uses standard file formats (PDDL, JSON configuration file, *obj* 3D object files) and they are easy to modify, so users can use their own PDDL model files and convert the 3D representation to a suitable representation.

Future work includes a PDDL code editor with syntax highlight, a user area for online access (user login, PDDL file storage on servers and other features) and an easy editor for 3D scene configuration. We expect users to be able to use this system online for the complete description and validation of PDDL problems.

ACKNOWLEDGEMENT

This work is partially funded by the Spanish Ministry of Economy and Competitiveness (Project *Assistive Navigation using Natural Language*, DPI 2014-53525-C3-1-R), whose kindness we gratefully acknowledge. Finally we express our gratitude to reviewers for their insightful suggestions and comments.