A planning problem validator based on reachability analysis

Moses Symeonidis
Department of Informatics, Aristotle University of Thessaloniki
Thessaloniki, Greece
mousisss@csd.auth.gr

Dimitris Giouroukis
Department of Informatics, Aristotle University of Thessaloniki
Thessaloniki, Greece
dgiourou@csd.auth.gr

Dimitris Vrakas
Department of Informatics, Aristotle University of Thessaloniki
Thessaloniki, Greece
dvrakas@csd.auth.gr

ABSTRACT

AI planning is the field that focuses on creating abstract representations of various processes. As such, it is obvious that it is a broad field and demands specialized tools to be used in order to ease the process of designing domains and problems. The literature shows that most of these tools do not offer an option to re-examine and validate the result of the aforementioned designing process.

This work presents a novel approach to validating a description of a problem as well as presenting information related to this validation process in an efficient way. It introduces a proxy web service that implements the process of finding invalid descriptions as well as the extension of the relevant tool VLEPpO, that suited the needs of the visualization process.

CCS Concepts

•General and reference → Validation; •Computing methodologies → Planning and scheduling; •Human-centered computing → *Visualization design and evaluation methods;*

Keywords

validation; planning; visualization; knowledge engineering; artificial intelligence; web service; scheduling

1. INTRODUCTION

The process of writing and verifying planning problems is a process that can be quite complex and demands a lot of resources in order to produce or verify a satisfying solution. For that reason, specialized software has been created with the purpose of easing the whole process, either for experienced and novice knowledge engineers alike. A brief investigation on the existing toolset shows that there is need for further research and investigation concerning the ease of the process of solving planning problems. This work focuses on the creation of a new system that verifies planning domains and problems through the development of a specialized web service as well as through the extension of existing means of visualization in order to better utilize the newly created functions. To be more precise, through the use of the web services we try to determine if a planning problem is solvable and if not present to the knowledge engineer the logical problem(s). Finally, these services are tightly integrated into VLEPpO [7], a platform for designing and validating planning problems in order to offer a desktop experience and showcase a valid use for the aforementioned services.

SETN '16, May 18-20, 2016, Thessaloniki, Greece

DOI: http://dx.doi.org/10.1145/2903220.2903237

2. RELATED WORK

In the related literature, many different tools that focus on the planning domain exist. These tools are either experimental or industry-ready. All of them offer a variety of designing and visualization features but none of them offers a concrete solution to validating the abstract description of a planning domain or a planning problem.

The GIPO system [9] is one of the most well known examples of presenting information regarding planning problems. It is defined as an "experimental environment for planning domain knowledge engineering" by the authors of the fourth release of the system. The main environment of the GIPO system relies on an object oriented approach in order to represent domains, plans or other entities. Information for a random object is propagated into the environment through the use of other core components that observe and interact with the object model component. Domain description is done through a generic abstraction layer that can translate graphical representations to specific language descriptions. GIPO supports PDDL (until v2.2) or OCL (Object Centered Language) as targets. GIPO offers an API in order to interface with external planners but the planners must use OCL 2.1 or typed/conditional PDDL. The output or input of GIPO is processed by Python scripts and every external planner must implement its own set of scripts.

GIPO provides editors and accompanying visual tools for domain creation/manipulation and planning preprocessing or description. Since it is object oriented, there is a designer made specifically for creating objects and specifying the underlying relationships between them. In order to properly design the various states of an object, the object life history editor can be used. A user can draw state machines for an object which describe the transitions between states. Another graphical editor, the coordination editor, is used to present transitions with their respective states. The notion of a constraint or a static fact is described using a non graphical approach. For problem instances, GIPO uses a list-view of objects that a user can assign specific values to each one.

Automatic domain validation is not possible in the GIPO environment but it offers automatic checks on the syntactic level of the description of the domain. As a mean to ease the debugging process of a problematic domain, GIPO uses steppers where users can see the effects of the application of an action step-by-step. The action has to be hand picked by the user.

Another system for interactive planning and execution of plans is SIPE-2 [12]. SIPE-2 focuses on the performance of the creation of a plan, thus to address the problem efficiently, SIPE-2 includes many heuristics in order to reduce computational complexity.

The system has the ability to generate automatically or under interactive control combined operators and creates plans to achieve the desired goals in the given world. An interesting part of the system is the monitoring of the plan execution process. This feature provides the ability to the user to feed more information to the system when there are some changes in the world.

Moreover, the system utilizes graphical interfaces for knowledge representation and has the ability of to visualize a plan. Generally, SIPE-2 is a system with a wide range of features and capabilities. Finally, it should be noted that the SIPE-2 uses ACT formalism, which does not correspond directly to PDDL. However, this shortcoming makes almost impossible to import or export domains or problems written in PDDL to SIPE-2.

itSimple [10] constitutes an environment for knowledge, planning and design engineering. The tool can ease the various processes of designing and solving planning problems. It uses a variety of established and well-known languages, like XML, UML, Petri Nets and PDDL for knowledge representation and it supports them interchangeably. This means that itSimple can export or import from and to any of the supported languages using internally maintained translators that change every language in an intermediate XML format. The language that is used for describing requirements in itSimple is UML but it also uses PDDL in order to validate and verify the application domain model.

The main focus of itSimple is to help the user find information in most stages of a planning application by using various means of visual feedback. At its core, itSimple represents information in an object oriented manner. The base modeling of a given problem is designed in UML, starting from a more abstract representation to specific details, like class diagrams, case diagrams and Petri nets.

Finally, plan verification is conducted in a simple PDDL editor, where a problem and a domain are presented to the user in text form. The domain and the problem are extracted from the UML diagrams. In order to facilitate the steps that are needed in correcting a false plan, itSimple uses Gantt and XY charts. The users have to examine the generated charts in order to extract meaningful information regarding their planning application.

EUROPA [1] is not just a stand-alone application for solving and visualizing planning problems but rather a whole framework for creating custom planners. EUROPA is a work of NASA and it licensed as an open source project.

The project focuses on flexible integration of new applications of planning problem solving. The client API provided by the framework helps applications interface with the built-in extensions and the kernel of EUROPA. The extensions offer constraint management, resources management, solver modules and a modeling language implementation. The language that is used for modeling is NDDL, a version of PDDL from NASA along with a parser for it. ANML can also be used as a language.

The deepest layer of EUROPA consists of its kernel which contains a constraint engine, a plan database, an engine for rules and a model interpreter. The plan database is a separate module that can be used in external applications. It helps represent information in a more abstract fashion.

In order to demonstrate the knowledge engineering capabilities of the platform, EUROPA also offers a modeling and visualization platform. A modified version of Eclipse is used as a basic NDDL editor that supports syntax highlighting and an outline. The same features as well as more will be utilized for ANML since there is an ongoing work for an editor that will support visual representation of the core language. Currently there is support for Gantt timeline charts, resource charts and action visualizations. EUROPA also offers an interactive console, for query execution onto the plan database, solver initialization and step control as well as to spawn helper windows for various object related actions. The visualizations can be run either through Eclipse, as a plugin of the IDE, or in a stand-alone fashion.

Automatic translation between description languages is already implemented but is very limited. Although the authors recognize that PDDL support would be necessary since it is the dominant modeling language in the research community, EUROPA currently has no PDDL translator or editor in its default configuration.

Modplan [2] is classified in the category of planning workbenches. For representation of information ModPlan uses PDDL. This fact makes it a tool that is more effective in the hands of planning experts than designers of domain knowledge. Moreover ModPlan utilizes VAL for plan validation and the Vega animation system which in turn allows a user to zoom in or out on a selected part of the plan.

In addition, the tool visualizes Gantt charts for temporal plans. Every Gantt chart depicts the estimated duration of every activity. Finally, plan animation is provided for some benchmark domains and, through the usage of JABBAH [6], shows output plans for business processes using another Gantt diagram.

VisPlan [5] is a graphical tool that focuses on plan visualization. VisPlan aims to visualize already existing plans and present possible flows to an end-user. It should be mentioned that VisPlan supports STRIPS-like plans and temporal plans. It recognizes these kind of plans automatically and verifies them based on their type.

Moreover, the users can modify or repair manually a selected plan. They can easily add, modify, remove or change the order of actions of a plan. As a tool which also focuses on plan validation and repair VisPlan has not any interface to create or modify domain or problem files. For this reason, the program demands that the domain and the problem are written in PDDL and are already saved on the disk.

Vodrazka and Chrpa present the tool VIZ [11]. VIZ is a simple, lightweight system which uses a straightforward approach to modelling a planning domain. As is mentioned in the release paper VIZ is inspired from the works of GIPO and itSIMPLE. The main goal of the project is to be more user friendly and less expert oriented. The users can create simple diagrams which may represent a domain or a problem and finally the tool can translate these diagrams to PDDL. VIZ does not have any method to generate a solution plan.

InLPG [4] offers a system that approaches the concept of planning in an abstract manner. It is a framework for domain-independent plan generation and visualization that is essentially a way to interact with a planner in a more dynamic way. It offers visual feedback and interaction during the creation process of the plan.

VLEPpO [7] was designed in order to create a more user-friendly interface than other solutions and it focuses on implementing an accurate representation of PDDL and all of its elements. A large number of these language constructs can be visually manipulated by a user using only the integrated visual editor.

Every user-made design can be exported to PDDL as well as an efficient textual representation on files that use the VVF extension. The same mechanism is used in order to load files written in PDDL or saved in VVF. In order to solve the occasional problem it uses an external planner, specifically LPG-td [3] but other planners can be integrated as well. Plans that are created through the external planner can be visualized as well.

The system includes lots of features that provide useful visual feedback. For example, a visual "map" is constructed in order to present relations that only have two arguments and these arguments are of the same type. This way, any state of the problem is more readable by the user.

In conclusion, most of the aforementioned systems are useful only to domain experts. They provide visual feedback for a solution to a given problem but not for the whole process of designing a domain and its corresponding problem. A solution that would provide an easy to use visualization that would abstract some of the information would be preferable for newcomers to the planning field as well as students or even domain experts.

Moreover, most of the systems tend to use an internally developed way to represent information. Even though most of them support input/output to PDDL, or even provide an editor or basic visualization for PDDL objects, they do not provide a straight-forward way to visually inspect every aspect of the PDDL that is to be used which would help to introduce someone to the field.

This work focuses on creating a validator for designing domains or problems and easing the process of finding errors in them. In order to achieve this, a web service was created that provides the logic for such a task as well as an appropriate graphical user interface. The validator was developed as an add-on to VLEPpO, facilitating the process of discovering logical errors in planning domains and problems.

3. VALIDPLAN

As it was shown in the previous section, a significant number of tools for editing planning problems, was produced especially, after the adoption of PDDL, in 1998, as the dominant language for defining planning problems. Most of these tools offer a graphical user interface that assist the domain engineer in designing, testing, checking the syntax and even vizualizing planning domains and problems. However, very little has been done in the area of validating domains and problems in terms of logical correctness. It is a common situation for users of PDDL based tools to design new planning problems that although they seem to be syntactically correct, prove to be unsolvable when fed to planning systems. ValidPlan is an approach to create a system that extracts information from unsolvable problems in order to assist the domain engineer in debugging PDDL files from logical errors. The proposed approach performs a reachability analysis on planning problems, by creating a series of planning graphs, a construct proposed by Blum and Furst in [2].

3.1 Planning Graphs

A planning graph is a leveled acyclic graph that interleaves levels of facts and actions. The graph starts with fact level f_0 containing the initial state, and then proceeds with: a) action level a0 with all the problem actions that have their preconditions in f_0 and b) fact level f_1 that contains f_0 plus all the positive effects of actions in a_0 . Planning graphs also stores information concerning mutual exclusions (mutexes).

A mutual exclusion between two facts in level f_k indicates that there is no state containing both of them that is reachable from the initial state after executing $\leq k$ parallel steps. Similarly a mutual exclusion between two actions in level am indicates that these two actions cannot be executed in parallel at step m of a parallel plan.

The process of building a planning graph contains the iterative expansion of the graph by adding new fact and action levels and by calculating mutex relations in any new level. The expansion stops when it reaches a fact level f_n which contains all the problem goals and there are no mutual exclusions among them (possible solution level).

Since the planning graphs expand monotonically, level i has equal or less nodes and equal or more mutex relations than level $i+1$, it is easily shown that even for unsolvable problems the graph will eventually reach a fact level z (level-off), which will contain the maximum number of facts and the minimum number of mutexes and therefore further expansion of the graph will be pointless.

3.2 Overview of the proposed methodology

ValidPlan takes the PDDL files of the planning domain and problem and builds a planning graph until it reaches a possible solution level or the graph levels-off. In the first case the problem is considered to be solvable and ValidPlan terminates. In the second case the problem is certainly unsolvable and ValidPlan uses the information stored in the graph in order to help the user understand why.

Initially, ValidPlan uses the final level of the graph in order to divide the goals of the problem in two distinct sets: reachable and unreachable. If a goal gk is present at the final level of the graph, it can be proven that there is at least one serial plan achieving it and thus gk is reachable from the initial state. On the other hand, if gk is absent from the final level of the graph it can also be proven that there is no serial plan achieving it and therefore gk is not reachable from the initial state.

The second task of ValidPlan is to check in the last fact level for possible mutexes among the reachable goals. If two goals are marked as mutexed at the final level of a graph that it has leveled-off, it means that there is no state in the search space in which both of them hold.

Using the information from these tasks ValidPlan is able to identify three possible reasons for the problem to be unsolvable: a) There is at least one unreachable goal g_k and there is no action in the problem having g_k in its add list. b) There is at least one unreachable goal g_k and although there are actions in the problem having g_k in their add list, these actions were not included in the planning graph. c) There is at least one set of reachable goals g_k and g_m, that are marked as mutual exclusive in the final level of the graph.

Case a is simple to identify and it is easy to communicate the logical error to the user, since a simple message "There is no action adding g_k" is enough for the domain engineer to point the error out (e.g. he forgot to define an operator, or to add the relevant predicate to an existing operator).

Case b is also relatively simple because the logical error of the problem is related to the reasons why the actions achieving g_k were not included in the planning graph. In order for an action to be included in level a_i of the planning graph, two conditions must hold: a) all the preconditions of the action must be present in f_i and there is no mutex in f_i between any pair of its preconditions. Since the last level of the graph f_z contains the maximum number of facts and the minimum number of mutexes, these two conditions are checked over f_z. The user is presented with a list of actions achieving g_k and is asked to select one of them. Then ValidPlan checks the preconditions of the action over f_z and the user is informed about the result (e.g. some precondition(s) of the selected action are not reachable, or there is one or more mutex(es) among them). If this information is not sufficient for the user, he has the ability to re-execute ValidPlan recursively using the preconditions of the selected action as goals in order to look deeper for the reasons.

Case c refers to the situation where there are two goals g_k and g_m , which are marked as mutexed in f_z . Two facts g_k and g_m are marked as mutexed in level f_z if: a) one is the negation of the other, or b) any action at level a_{z-1} achieving g_k is marked as mutexed with all the actions at a_{z-1} achieving g_m and vice versa. Therefore, the rationale behind the unsolvability of the problem can be regressed to the reason why g_k and g_m are marked as mutexed. ValidPlan will present the user with the list of reachable actions achieving g_k and the list of reachable actions achieving g_m and give him the option to select one action from each list. Based on the user's selection and the information from the planning graph it will inform the user for the reason why these specific actions are mutually exclusive (e.g. the first one is deleting a precondition of the second one). In case the reason for the mutex is Competing Needs (e.g. the preconditions of the two actions are mutexed), then ValidPlan will be recursively investigate the reason for that.

4. CASE STUDY

ValidPlan facilitates the debugging of logical errors in planning domains and problems through three distinct operations on the user's PDDL files. These three operations, namely Predicate Categorization, Atomic Goal Reachability and Group Reachability, will be exemplified in this section using a case study on a simple transportation domain with a robot carrying objects between connected rooms, which is a slightly modified version of the Gripper Domain.

4.1 The Domain

This domain consists of a number of rooms (that are treated as single points for simplicity), a number of objects (balls) that need to be transported and a robot moving across rooms (when it is permitted) and carrying balls (one at a time) . By modeling this domain in PDDL, one gets the following predicates: **room(R), connected(R1,R2), ball(B), at(B,R), atRobby(R), handsempty, holds(B)**.

There are three operators in the domain, move, pick and drop and their STRIPS-like encoding is presented in Table 1.

Operator	Preconditions	Add	Delete
move	*atRobby(X)* *connected(X, Y)*	*atRobby(Y)*	*atRobby(X)*
move2	*atRobby(X)* *connected(Y, X)*	*atRobby(Y)*	*atRobby(X)*
pick	*atRobby(X)* *at(B,X)* *handsempty*	*holds(B)*	*at(B, X)* *handsempty*
drop	*atRobby(X)* *holds(B)*	*at(B, X)* *handsemty*	*holds(B)*

Table 1: Domain Operators

Operator	Preconditions	Add	Delete
move	*atRobby(X)* *connected(X, Y)*	*atRobby(Y)*	*atRobby(X)* *connected(X, Y)*
move2	*atRobby(X)* *connected(Y, X)*	*atRobby(Y)*	*atRobby(X)* *connected(Y, X)*
pick	*atRobby(X)* *at(B,X)* *handsempty*	*holds(B)*	*at(B, X)* *handsempty*
drop	*atRobby(X)* *holds(B)*	*at(B, X)* *handsemty*	*holds(B)*

Table 2: Domain Operators with faulty move and move2

4.2 Predicate Categorization

Consider the case where the robot has to go from room A to room B, get a ball and return to A and by accident the domain expert has included the predicate **connected/2** in the delete list of the move operator. Note that this is a common mistake in visualization tools like VLEPpO, where the operators are designed using a state transition mode, where one defines explicitly what holds both states, rather than an add/delete list mode. The "faulty" operators are presented in Table 2, while the initial state and the goals are as follows:

- **I={room(r1), room(r2), atRobby(r1), ball(b), at(b, r2), handsempty, connected(r1,r2)}**
- **G={atRobby(r1), at(b,r1)}**

This problem is obviously unsolvable, because once the robot moves from **r1** to **r2** the fact **connected(r1,r2)** will be deleted from the state representation and therefore after the robot picks the ball there will be no way for returning back to **r1**.

The basic idea behind the validator is to find unreachable goals and to allow the user to traceback actions achieving each goal separately until he reaches the initial state or face a dead-end, which will hopefully provide him with enough

Operator	Preconditions	Add	Delete
move	*atRobby(X)* *connected(X, Y)*	*atRobby(Y)*	*atRobby(X)*
pick	*atRobby(X)* *at(B,X)* *handsempty*	*holds(B)*	*at(B, X)* *handsempty*
drop	*atRobby(X)* *holds(B)*	*at(B, X)* *handsemty*	*holds(B)*

Table 3: Domain Operators

information for sorting the logical error out. This process will be elaborated in the following sections.

For common errors like the one mentioned above, however, the validator offers a simpler solution that consists of presenting the user with a classification of the domain predicates in two categories: static vs dynamic. The predicates that lead to facts that hold their value (true or false) throughout the whole problem's search space are considered to be static, while the rest of them are dynamic. For example the predicate *ball/1* is a static one, while *atRobby/1* is obviously dynamic.

In order to determine whether a predicate is dynamic or not, the validator uses the following rule that can be easily checked with a simple iteration among the domain's operators:

Predicates appearing in the add list or in the delete list of any of the domain'soperators are marked as dynamic, while the rest of them are marked as static.

Applying the above rule for the domain predicates using the operators presented in Table 2 one would get the following lists:

- **Static:** *room/1, ball/1*
- **Dynamic:** *connected/2, at/2, atRobby/1, handsempty/0, holds/1*

For the knowledge engineer the fact that *connected/2* is marked as dynamic is a certain alarm and upon request the validator will inform him for the reason why it is marked as dynamic (delete list of operator *move/move2*).

4.3 Atomic Goal Reachability

As it was outlined in the previous section, the main idea behind the proposed validator is to analyze the reachability of goals and sub-goals and use the information from this analysis in order to guide the domain expert to identify possible logical errors.

For instance let us consider the problem instance presented in a previous subsection, where:

- **I={room(r1), room(r2), atRobby(r1), ball(b), at(b, r2), handsempty, connected(r1,r2)}**
- **G={atRobby(r1), at(b,r1)}**

Suppose now that the operators in hand are those presented in Table 3, where the domain expert failed to deal with the symmetric relation of the connected predicate.

Once again the problem is unsolvable because the fact *at(b,r1)* is unreachable, since the robot can go from *r1* to *r2* in order to pick the ball but not from *r2* to *r1* in order to drop it. In such cases (Atomic Goal Reachability problems) ValidPlan can assist the domain expert by giving him a tool for tracing from the unreachable goal back to the initial state. By clicking on the problem's goals the validator provides the user with a list of the problem's actions that have the goal in their add list, using color coding in order to show if each action is reachable from the initial state or not. In this case, the only suitable action is *drop(b,r1)*. The validator presents the user with all the preconditions of the selected action, using color coding in order to discriminate the reachable from the unreachable ones. By investigating the preconditions of *drop(b,r1)* the knowledge engineer will be informed that *move(r2,r1)* is not achievable from the initial state due to its precondition*connected(r2,r1)* that is absent from the initial state, enabling him to understand the domain's error and fix it.

4.4 Group Reachability

This function of the validator deals with goals (or subgoals) that cannot be achieved simultaneously due to the mutual exclusions. Mutexes are a key component in planning graphs, although the kind of mutexes stored by GraphPlan have a temporal annotation, i.e. two facts of actions are marked as mutexed for a specific level of the graph but this mutex may be lifted as the graph continues to expand. These temporal mutexes may be misleading for the purpose of a domain validator but it is safe to consider that when the graph has leveled-off, all the mutexes at the final level are static and can be used to assist the knowledge engineer in the debugging of the domain.

Group reachability issues are the most common reason for unsolvable problems and the knowledge engineer might have to trace back a few (or more than a few) levels before he is presented with the actual root of the error. However, we will present a rather simple scenario in order to show how the validator deals with cases like that.

Consider the full set of operators presented in Table 1 and the following initial stateand goals:

- **I={room(r1), atRobby(r1), ball(b1), ball(b2), at(b1, r1), at(b2, r1), handsempty}**
- **G={holds(b1), holds(b2)}**

This problem is clearly unsolvable because although both goals are atomically reachable they are mutually exclusive, unless of course the domain is enriched with multiple gripper support.

When the problem is submitted to the validator, the latter will inform the user that the problem is unsolvable because the two goals are mutually exclusive. By investigating the reason for the mutual exclusion one can get a clear view of the reasons behind the unsolvability of the problem.

For example, when the validator is executed on the above problem instance it will report that the set of goals is unsolvable because *holds(b1)* is mutexed to *holds(b2)*. It will then present to the user a visual tool from which he can select for each goal one of the actions achieving it. Every time the user selects a pair of actions achieving **G** the tool will provide information regarding why the selected pair of actions is mutually exclusive. In the specific problem in hand, the reason is that both actions have handsempty in their precondition along with their delete list (interference) and therefore there is no possible serialization of *pick(b1,r1)* and *pick(b2,r1)* because handsempty will not hold for the second action.

5. IMPLEMENTATION

ValidPlan was implemented as a SOAP web service in order to offer its functionality over the web and to allow any visual or textual tool for editing PDDL files to use it for validation. For the implementation of the web service, Java was chosen as the programming language and the JSON representation for encoding the output of the program. Our work also incorporates the PDDL4j library [1] as well as the implementation of Graphplan from the JavaGP library [8]. It should be mentioned that a proxy web service was created, based on WSDL prototype for having a description of the whole service readily available to clients. For the implementation of the service, the programming language that was chosen is PHP. PHP is responsible for calling the Java executable to generate the results of the processes of validation.

In order to test the validator and come up with an integrated tool for editing planning problems, ValidPlan was implemented in VLEPpO. VLEPpO is a visual tool for editing and designing planning problems and therefore the operations of the ValidPlan web service were accompanied by relevant visual aids.

The current VLEPpO implementation utilizes the Swing Framework for UI creation as well as the Java2D library for the implementation of trivial graphics. No external libraries were needed. From VLEPpO the user has the ability to use ValidPlan either as a standalone plugin for users working offline or as a web service (in order for the user to benefit from future updates of the validator).

Since ValidPlan is implemented as a web service the process of incorporating it to other tools is relatively simple (loosely coupling). The service already offers a WSDL definition of itself. Messages are propagated in SOAP and the payload is encoded in JSON format.

The core functionality of ValidPlan through VLEPpO is composed of the following functions: a) dynamic/static literals discovery and b) domain/problem validation.

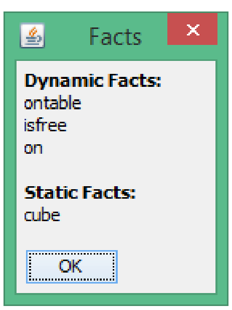

Figure 1: A window showing the dynamic and static facts of a problem description.

In order to represent the discovered dynamic or static facts in the description of a problem, our solution employs two textual lists on a pop-up box. The first one lists the set of the dynamic facts and the second one is used for the static ones. This type of information, although important for debugging the design process, does not require a complex visualization and thus, the straightforward way of presenting two different lists was chosen. The basic window that presents the information to the users is shown in Figure 1.

The validation process requires a lot of different pieces of information to be visualized. First, a user may require the service to return reachable and non-reachable goals. They are also presented in an interactive fashion where users can select any non-reachable goal from the respective list and then investigate these goals further. Each of the non-reachable goals is re-inserted into the process and then is presented along with the respective action and its preconditions. Preconditions are colored according to whether they can be achieved or not. Every non-reachable precondition can be selected with a right-click from a point and the process can be re-run recursively, with the selected precondition as input. This way, it is easier to pinpoint a problematic description and correct it, from the standpoint of a user.

The process of validation also involves the discovery of mutexes. The implemented system can identify and return a list of mutexed goals as well as the reasons for why a pair of goals is mutexed. There is a lot of information that has to be presented and to keep the visual clutter low our implemented visualization follows the following logic: for every mutexed pair of goals, with a fixed/selected first goal, the user can select a pair of actions from their respective list and from there, a specific type of conflict. Since every type of conflict is different in nature, these types have to be presented in a separate manner. The problematic parts of the description are highlighted in red color. The mutexed goals that have been selected by the user are represented by red nodes. Our implementation not only manages to present why a pair of goals is mutexed but can represent every possible reason in the same space, thus minimizing the effort put by the user to understand why a mutex exists in the first place.

Since ValidPlan was created as an educational tool, performance or scalability were never top priorities since the needs that the tool addresses are small in size and require a small amount of resources.

6. CONCLUSION AND FUTURE WORK

This paper presented ongoing work on creating a validator for planning domains and problems using reachability analysis. The proposed system, namely ValidPlan determines successfully if a planning problem is unsolvable and it sufficiently presents to the user the cause (or causes) of the problem, i.e. the logical errors in the design of the problem. ValidPlan was also integrated in VLEPpO, a visual tool for designing and visualizing planning problems and the first results of an experimental analysis using the enhanced visual tool are more than promising.

In the future we plan to conduct a thorough experimental analysis of how experienced and novice knowledge engineers can benefit from a validator like ValidPlan in the processes of: a) designing new domains and problems and b) debugging faulty domains and problems, initially created by third parties. Apart from the experimental analysis, one of the direct goals is to extend the notion of mutual exclusions used in the reachability analysis in order to cover mutexes of higher order, since the current version of ValidPlan cannot deal with problems sourcing from triangular mutexes, or mutexes involving more parties. Finally, although the mechanism for Group Reachability seems to be working, there are

[1]https://sourceforge.net/projects/pdd4j

specific cases where we have identified tailored solutions that could improve the efficiency of the system and the comprehensiveness of the messages to the user.

7. REFERENCES

[1] J. Barreiro, M. Boyce, M. Do, J. Frank, M. Iatauro, T. Kichkaylo, P. Morris, J. Ong, E. Remolina, T. Smith, et al. Europa: A platform for ai planning, scheduling, constraint programming, and optimization. In *Proceedings of the 22nd International Conference on Automated Planning & Scheduling (ICAPS-12)–The 4th International Competition on Knowledge Engineering for Planning and Scheduling*, 2012.

[2] S. Edelkamp and T. Mehler. Knowledge acquisition and knowledge engineering in the modplan workbench. *International Competition on Knowledge Engineering for Planning and Scheduling*, pages 26–33, 2005.

[3] A. Gerevini, A. Saetti, I. Serina, and P. Toninelli. Lpg-td: a fully automated planner for pddl2. 2 domains. In *In Proc. of the 14th Int. Conference on Automated Planning and Scheduling (ICAPS-04) International Planning Competition abstracts.* Citeseer, 2004.

[4] A. E. Gerevini and A. Saetti. An interactive environment for plan visualization and generation: Inlpg. In *Working notes of Eighteenth International Conference on Automated Planning & Scheduling (ICAPS-08)-System Demonstration, Sydney (Australia)*, 2008.

[5] R. Glinskỳ and R. Barták. Visplan–interactive visualisation and verification of plans. *KEPS 2011*, page 134, 2011.

[6] A. González-Ferrer, J. Fernández-Olivares, L. Castillo, et al. Jabbah: a java application framework for the translation between business process models and htn. *Proceedings of the 3rd International Competition on Knowledge Engineering for Planning and Scheduling (ICKEPSŠ09)*, 2009.

[7] O. Hatzi, D. Vrakas, N. Bassiliades, D. Anagnostopoulos, and I. Vlahavas. Vleppo: A visual language for problem representation. *PlanSIG*, 7:60–66, 2007.

[8] F. Meneguzzi and M. Luck. Leveraging new plans in agentspeak (pl). In *Declarative Agent Languages and Technologies VI*, pages 111–127. Springer, 2009.

[9] R. Simpson. Structural domain definition using gipo iv. *Proceedings of the Second International Competition on Knowledge Engineering for Planning and Scheduling*, 2007.

[10] T. Vaquero, R. Tonaco, G. Costa, F. Tonidandel, J. R. Silva, and J. C. Beck. itsimple4. 0: Enhancing the modeling experience of planning problems. In *System Demonstration–Proceedings of the 22nd International Conference on Automated Planning & Scheduling (ICAPS-12)*, 2012.

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

[12] D. E. Wilkins and A. I. Center. Using the sipe-2 planning system. *Artificial Intelligence Center, SRI International, Menlo Park, CA*, 1999.