VLEPpO: A Visual Language for Problem Representation

Ourania Hatzi[1], Dimitris Vrakas[2], Nick Bassiliades[2], Dimosthenis Anagnostopoulos[1] and Ioannis Vlahavas[2]
[1]*Department of Geography, Harokopio University of Athens, Athens, Greece*
{raniah, dimosthe}@hua.gr
[2]*Dept. Of Informatics, Aristotle University Of Thessaloniki, Thessaloniki, 54124, Greece*
{dvrakas, nbassili, vlahavas}@csd.auth.gr

Abstract

AI planning constitutes a field of interest as its techniques can be applied to many areas. Contemporary systems that are being developed deal with certain aspects of planning and mainly focused on dealing with advanced features such as resources, time and numerical expressions. This paper presents VLEPpO, a Visual Language for Enhanced Planning problem Orchestration. VLEPpO is a visual programming environment that allow the user to easily define domains and problems and acquire solutions, utilizing web services infrastructure.

1. Introduction

AI planning has been an active research field for a long time, and its applications are manifold. A great number of techniques and systems have been proposed during this period in order to accommodate designing and solving of planning domains and problems. In addition, various formalisms and languages have been developed for the definition of these domains, with Planning Domain Definition Language (PDDL) [4][5][6] being dominant among them.

Research among contemporary planning systems has revealed a lack of appropriate integrated visual environments for representing accurately PDDL elements and structures, and consequently using these structures to produce quality plans. This provided the motivation for the work presented in this paper.

The proposed visual tool is intended to cover the need for such an environment by providing an easy to use, efficient graphical user interface, as well as interoperability with planning systems implemented as web services. The elements offered in the interface correspond to PDDL elements and structures, making the representation of most contemporary planning domains possible. Furthermore, importing and exporting to PDDL features are provided as well. Drag and drop operations along with validity checks make the use of the environment easy even for users not particularly familiar with the language.

The rest of the paper is organized as follows: Section 2 reviews related work in the field by presenting several planning systems, while Section 3 discusses the eminent formalisms for representing planning domains and problems. Section 4 presents our visual tool and demonstrates its use through examples, and finally, Section 5 concludes and discusses future goals.

2. Related Work

There have been a few experimental efforts to construct general-purpose tools which offer user interfaces for defining planning domains and problems, as well as executing planners which provide solutions to the problems.

The GIPO system [1] is based on an object-centric view of the world. The main idea behind it is the notion of change in the state of objects throughout plan execution. Therefore, the domains are modeled by describing the possible changes to the objects existing in the domain. The GIPO system is designed to work with both classical and HTN domains. In both cases, it offers graphical editors for domain creation, planners, animators for the derived plans and validation tools. The domain models are represented mainly in an internal representation language called OCL, which is object oriented, in accordance with the GIPO system. Translators from and to PDDL have been developed, which cover only a few parts of the language (typed / conditional PDDL).

SIPE-2 [2] is another system for interactive planning and execution of the derived plans. As it is designed to be performance-oriented, it embodies many heuristics for increased efficiency. Another useful feature is the plan execution monitoring, which enables the user to feed new information to the system in case there is some change in the world. In addition, the system offers graphical

interfaces for knowledge acquisition and representation, as well as plan visualization. SIPE-2 is an elaborate system with a wide range of capabilities. However, it uses the ACT formalism, which is quite complicated and does not correspond directly to PDDL, although PDDL descended partially from this formalism, but also from other formalisms such as ADL. Therefore, there is no way to easily use a PDDL file to construct a domain in SIPE-2, or export the domain or problem to PDDL.

ASPEN is an environment for automated planning and scheduling. It is an object-oriented system targeted to space mission operations. Its features include an expressive constraint modeling language which is used for defining the application domain, systems for defining activity requirements and resource constraints, as well as temporal constraints. In addition, a graphical user interface is included, but its use in confined to visualizing plans and schedules, in systems where the problem solving process is interactive.

ASPEN was developed for the specific purposes of space mission operations and therefore, it has only a few vague correspondences to PDDL. Furthermore, it does not offer a graphical interface for creating the planning domains.

In conclusion, although the above systems are useful, none of them offers direct visual representation of PDDL elements, a feature which would make the design very efficient for the users already familiar with the language. Moreover, even the systems which offer translation to PDDL do not cover important features of the language. It should be mentioned that a couple of other systems exist which provide user interfaces but are not mentioned here because they are developed for specific purposes.

The VLEPpO tool is based on ViTAPlan [3] a visualization environment for planning based on the HAP_{RC} planning system. VLEPpO extends ViTAPlan in numerous ways providing the user with visualization capabilities for most of the advanced features of PDDL [6] and a more accurate and expressive visual language.

3. Problem Representation

A crucial step in the process of solving a search problem is its representation in a formal language. The choice of the language can significantly affect not only the comprehensiveness of the representation but also the efficiency of the solver. The PDDL language is nowadays the standard for representing planning problems. PDDL is based on the STRIPS [7] formalism.

3.1. The PDDL Definition Language

PDDL [4] stands for Planning Domain Definition Language. Although it was initially designed for planning competitions such as AIPS and IPC, it has become a standard in the planning community for modeling planning domains. PDDL focuses on expressing the physical properties of the domain that we consider in each planning problem, such as the available predicates and actions. At the same time, there are no structures to provide the planner with advice, that is, guidelines about how to search the solution space, although extended notation may be used, depending on the planner.

Each domain definition in PDDL consists of several declarations, which include types of entities, variables, constants, literals that are true at all times called timeless, and predicates. In addition, there are declarations of actions, axioms and safety constraints. These elements are explained in the following paragraphs.

Variables have the same semantics as in any other definition language, and are used in conjunction with built-in functions for expression evaluation. In more recent versions of PDDL, fluents seem to gain momentum instead of variables when there is a need for values that can change over time, as a result of an action.

Constants represent objects that do not change values and can be used in the domain operators or the problems associated with a domain.

Relations between objects in the domain are represented by predicates. A predicate may have an arbitrary number of arguments. Ordering of these arguments is important in PDDL. Predicates are used to describe the state of the world at a specific moment. Moreover, they are used as preconditions and results of an action.

Timeless predicates are predicates that are true at all times. Therefore, they cannot appear as a result of an action unless they also appear among its preconditions. In other words, timeless predicates are not affected by any actions available to the planner.

Actions enable transitions between successive situations. An action declaration mentions the parameters and variables involved, as well as the preconditions that must hold for the action to be applied. PDDL offers two choices when it comes to defining the results of the action: The results can either be a list of predicates called effects, or an expansion, but not both at the same time. The effects, which can be both conditional and universally quantified, express how the world situation changes after the action is applied. More specifically, inspired by the STRIPS formalism, the effects include the predicates that will be added to the world state and the predicates that will be removed from the world state.

Axioms, in contrast to actions, state relationships among propositions that hold within the same situation. The necessity of axioms arises from the fact that the action definitions do not mention all the changes in all predicates that might be affected by an action. Therefore, additional predicates are concluded by axioms after the

application of each action. These are called derived predicates, as opposed to primitive ones. In more recent versions of the language the notion of derived predicates has replaced axioms, but the general idea described remains the same.

Safety constraints in PDDL are background goals which may be broken during the planning process, but ultimately they must be restored. Constraint violations present in the initial situation do not require to be fulfilled by the planner.

Finally, in PDDL, we can add axioms and action expansions modularly using the construct addendum.

After having defined a planning domain, problems can be defined with respect to it. A problem definition in PDDL must specify an initial situation and a final situation, referred to as goal. The initial situation can be specified either by name, or as a list of literals assumed to be true, or a combination of both. In the last case, literals are treated as effects; therefore they are added to the initial situation stated by name. Again, the closed-world assumption holds, unless stated otherwise. Therefore, all predicates which are not explicitly defined to be true in the initial state are assumed to be false. The goal can be either a goal description, using function-free first order predicate logic, including nested quantifiers, or an expansion of actions, or both. The solution given to a problem is a sequence of actions which can be applied to the initial situation, eventually producing the situation stated by the goal description, and satisfying the expansion, if there is one.

PDDL 2.1 [5] was designed to be backward compatible with PDDL 1.2, and to preserve its basic principles. It was developed by the necessity for a language capable of expressing temporal and numeric properties of planning domains.

The first of the extensions introduced were numeric expressions. Primitive numeric expressions are values of functions which associate tuples of domain objects. Further numeric expressions can be constructed using primitive ones and arithmetic operators. In order to support numeric expressions, new elements were added to the language. Functions are now part of domain definition. As mentioned above, they associate a number of objects with an arithmetic value. Moreover, conditions were introduced, which are actually comparisons between pairs of numeric expressions. Finally, assignment operations are possible, with the use of built-in assignment operators such as assign, increase and decrease. The actual value for each combination of objects given by the functions is not stated in the domain definition but must be provided to the planner in the problem definition.

A further extension to PDDL facilitated by numeric expressions is plan metrics. Plan metrics specify the way a plan should be evaluated, when a planner is searching not for any plan, but for the optimal plan according to some metric.

Other extensions in this version include durative actions, both discretised and continuous. Up to now, actions were considered instantaneous. Durative actions, as the term implies, have a duration which is declared along with their definition. Furthermore, as far as discretised durative actions are concerned, temporal annotations are introduced to their conditions and effects. A condition can be annotated to hold at the start of the interval, at the end of the interval, or all over the interval during which the action lasts. An effect can be annotated as immediate, that is, takes place at the start of the interval, or delayed, that is, takes place at the end of the interval.

In PDDL 3.0 [6] the language was enhanced with constructs that increase its expressive power regarding the plan quality specification. The constraints and goals are divided into strong, which must be satisfied by the solution, and soft, which may not be satisfied, but are desired.

4. The Visual Language

VLEPpO (Visual Language for Enhanced Planning Problem Orchestration) is an integrated system for visually designing and solving planning problems, implemented in Java. It offers an efficient and easy-to-use graphical interface, as well as compatibility and interoperability with PDDL. The main goal during the implementation of the graphical component of the tool was to keep the interface as simple and efficient as possible, but, at the same time, represent accurately the features of PDDL. The range of PDDL elements that can be represented in the tool is quite wide, and covers the elements that are used more frequently in contemporary planning domains and problems. In the following, the features of the tool will be discussed in more detail.

4.1. The Entity – Relation Model

The entity – relation model is used to design the structure of data in a system. Our visual tool employs this well-known formalism, adapting it to PDDL. Therefore, the entities in a planning domain described in PDDL are the objects of the domain, while the relations are the predicates. These elements are represented visually in the tool by various shapes and connections between them.

A class of objects in the tool is represented visually by a colored circle. A class in PDDL represents a type of domain objects or action parameters. From a class the user can create parameters of this type in operators, and objects of this type in problems, by dragging and dropping a class on an operator or problem, respectively.

The type of a parameter or object is denoted by their color, which is the same as the corresponding class.

Consider the gripper domain for example, where there is a robot with N grippers that moves in a space, composed of K rooms that are all connected with each other. All the rooms are modeled as points and there are connections between each pair of points and therefore the robot is able to reach all rooms starting from any one of them with a simple movement. In the gripper domain there are L numbered balls which the robot must carry from their initial position to their destination.

Following a simple analysis the domain described above can be encoded using four classes: robot, gripper, room and ball. However, since the domain does not support the existence of multiple robots, the one robot can be implicitly defined and therefore there is no need for a robot class. The three remaining classes are represented in VLEPpO using three colored circles as outlined in Figure 1.

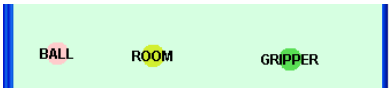

Figure 1. The classes in Gripper domain.

A relation is represented by a colored rectangle with black outline. A relation corresponds to a domain predicate in PDDL and it is used for defining connections among classes. The relations in PDDL and therefore in VLEPpO are of various arities. Unary relations are usually used to define properties of classes that can be modeled as binary expressions that are either true or false (e.g. closed(Door1)).

If at least one pair of class and relation is present in the domain, the user can add connections between them. Each connection represents an argument of a relation, and the class shows the type of this argument. A relation may have as many arguments as the user wishes, of any type the user wishes. The arguments are ordered according to the numbers on each connection, because this ordering is important to PDDL.

The Gripper domain has four relations, as depicted in Figure 2: a) *at-robbot*, which specifies the position of the robot and it is connected only with one instance of room, b) *at* which specifies the room in which each ball resides +and therefore is connected with an instance of ball and an instance of room, c) *holding* which defines the alternative position of a ball, i.e it is held by the robot and therefore it is connected with an instance of ball and an instance of gripper and d) *empty* which is connected only with an instance of gripper and states that the current gripper does not hold any ball.

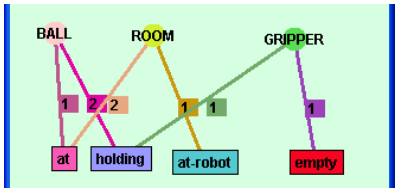

Figure 2. The relations in the Gripper domain.

Note here that although PDDL requires only the arity for each predicate and not the type of objects for the arguments, the interface obliges the user to connect each predicate with specific object classes and this is used for the consistency check of the domain design. According to the design of Figure 2, the arity of predicate holding, for example, is two and the specific predicate can only be connected with one object of class ball and one object of class gripper.

The aforementioned elements, classes, relations and connections combined together form the entity – relation model of the data for the planning domain the user is dealing with.

4.2. Representing Operators

Operators have direct correspondence to PDDL actions, which enable transitions between successive situations. The main parts of the operator definition are its preconditions and results, as well as the parameters. Preconditions include the predicates that must hold for the action to be applied. Results are the predicates that will be added or removed from the world state after the application of the action. Operators in the visual tool are represented by light blue resizable rectangles in the Operator Editor, comprised by three columns. The left column holds the preconditions, the right column holds the effects, and the middle one the parameters.

Dragging and dropping a relation on an operator will add the predicate to the preconditions or effects, depending on which half of the operator the shape was dropped on. Parameters can be created in operators by dropping classes on them. Adding a connection in the ontology enables the user to add corresponding connections in the operators. Other elements that can be imported in operators will be discussed in more detail in the section about advanced features.

For example, in the gripper domain there are three operators: a) *move* which allows the robot to move between rooms, b) *pick* which is used in order to lift a ball using a gripper and c) *drop* which is the direct opposite of pick and is used to leave a ball on the ground

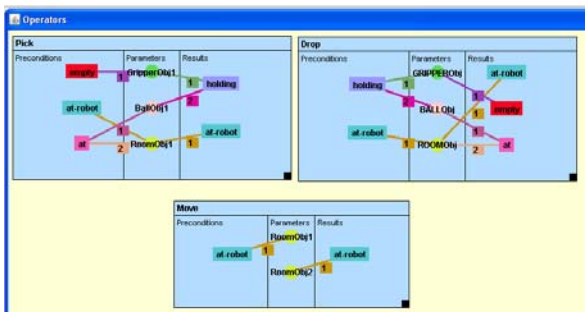

Figure 3. The operators in the Gripper domain.

The default view for an operator is in preconditions / results view which follows a declarative schema that is different from the classical STRIPS/PDDL approach. However, there is a direct way to transform definitions from one approach to the other.

Although the preconditions/results view is more appropriate for visualizing operators, the system gives the user the option to use the classical add/delete view. If selected, the column on the left, as before, shows the preconditions that must hold for the action to be executed, but the column on the right shows the facts that will be added and deleted from the current state of the world upon the execution of the action.

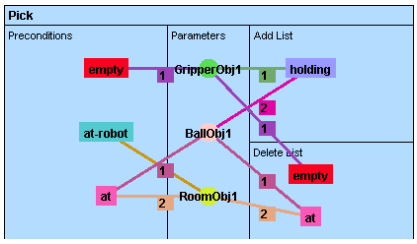

Figure 4. Pick operator in add/delete lists view.

Consider for example the Pick operator of the Gripper domain. According to the STRIPS formation, the operator is defined by the following three lists as depicted in Figure 4:

```
prec={empty(GripperObj1),at-robot(RoomObj1),
at(BallObj1,RoomObj1)}
add={holding(GripperObj1,BallObj1)}
del={empty(GripperObj1),at(BallObj1,RoomObj1)}
```

The equivalent operator in Preconditions/Results view is presented in Figure 5.

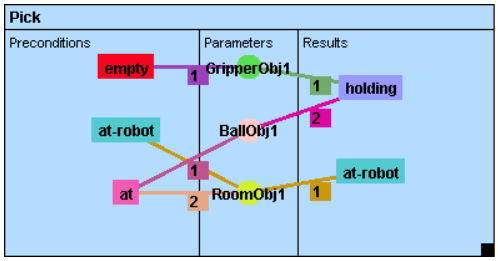

Figure 5. Pick operator in preconditions / results view.

4.3. Representing Problems

For every domain defined in PDDL a large number of problems that correspond to this domain can also be defined. Problem definitions state an initial and a goal situation, and the task of a planner is to find a sequence of operators that, if applied to the initial situation, will provide the goal situation. The problem shape in the visual tool is much like an operator in form, but different semantically. It is represented by a three-column resizable rectangle in the Problem Editor. Left column holds the predicates in the initial state, right column holds the predicates in the goal state, and middle column holds the objects that take part in the problem definition.

Figure 6 presents a problem instance of the gripper domain, which contains two rooms (Bedroom and Kitchen), one ball (Ball1) and the robot has two grippers (rightGripper and leftGripper). The initial state of the problem defines the starting locations of the robot and the ball (Kitchen and Bedroom respectively) and that both grippers are free. The goals specify that the destination of both the ball and the robot is the kitchen.

4.4. Advanced Features

The basic PDDL features described above are adequate for simple planning domains and problems. However, the language has many more features divided into subsets referred to as requirements. An effort has been made in order for the visual tool to embody the most important of them.

An advanced design element offered by the system, which has direct representation in PDDL, is a constant. The constant is visually represented similarly to a class, but it is enhanced with a red circle around it to discriminate it from a class. The constant must be of a type, and the tool enables the user to drag and drop it on a class to denote that. Constants can be used either in an operator or in a problem, where it behaves similar to a parameter or an object, respectively.

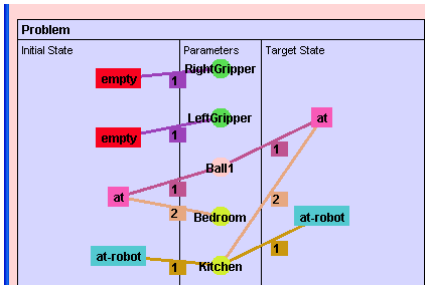

Figure 6. A Problem instance of the Gripper domain.

A derived predicate is another advanced PDDL feature that is represented by a group of design elements in the visual tool. The term refers to predicates that are not

affected by operators, but they are derived using a set of rules by other relations. Derived predicates existed in the first version of the PDDL language under the name "axioms". Visually, they are represented by a rounded rectangle with a specific color, but they are not complete unless they are enhanced with an and/or tree that indicates the way they are derived by other relations. Consequently, AND, OR and NOT nodes for the construction of the tree are also offered as design elements. In the current implementation, AND and OR nodes are binary, that is, they accept only two possible arguments, while NOT node is by default unary. Each of the node arguments can be either another node of any type, or a relation.

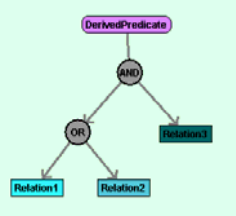

Figure 7. A derived predicate with AND/OR tree.

Among the advanced features is the option to indicate that a predicate is timeless, that is, the predicate is true at all times. This operation involves a lot of validity checks, which will be explained in the corresponding paragraph.

Another PDDL feature incorporated in the tool are numerical expressions. In order for numerical expressions to function properly, the definition of a number of other elements is involved. Consequently, a combination of design elements in each frame is used. First of all, in the ontology frame the user can import functions. They are represented by rectangles with double outline. These functions may or may not have arguments. As with simple relations, functions can be dragged on operators. However, in order to appear in the PDDL description of an operator, they must be involved in a condition or in an assignment. The next step is to actually import conditions and assignments which involve these functions in the operator. In that case, a dialog box appears facilitating the import of a condition or an assignment, by showing all the available options that the user can select among. Furthermore, for each function imported in the tool, a new rectangle appears in the problem frame, which corresponds to this function. This rectangle is used to declare the initial values of the function for the problem at hand.

Moreover, the system supports discretised durative actions. The definition of such a durative action includes setting the duration of an operator, in combination with temporal annotations. In this case, the action is considered to last a specific period of time, which can be specified by right click on the operator. The preconditions can be specified to hold at the beginning of this period, at the end of this period, or all over the period (combination of these choices is also possible). Effects can be immediate, that is, happen at the beginning of the action, or delayed, that is happen at the end of the action.

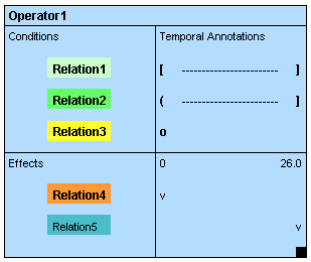

Figure 8. An example of a durative action.

Finally, a very useful element for problem designing is maps. Maps represent a special kind of relations that have exactly two arguments of the same type, and are expected to have many instances in the initial state of a problem. For each relation that fulfills these conditions a map can be created. Objects which take part in the instances of the relation can then be dragged on the map, and connections can be created between them. Each of these connections represents an instance of the relation that the map corresponds to. In conclusion, maps express a part of the initial state of the world, thus making the problem shape more readable. The use of maps is not mandatory, as the same relations can be simply represented in the problem shape.

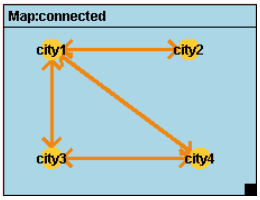

Figure 9. An map for the relation connected(C1, C2).

4.5. Syntax and Validity Checking

A very important aspect in every tool for designing and editing planning domains is syntax and validity checking. Planning domains have to be checked for consistency within their own structures, and planning problems have to be checked for consistency and correspondence to the related domains. This visual tool attempts to detect inconsistencies at the moment they are created and notify the user about them, before they propagate in the domain. In the remainder of this paragraph several examples will be given, in order to illustrate the validity checking processes of the system.

Whenever the user attempts to insert a new connection in an operator or in a problem, necessary checks are performed and if a corresponding connection cannot be found in the ontology an appropriate error message is shown. Special care must be taken to verify that the types

of parameters and objects match to the types of arguments of the predicates.

As already mentioned, the system supports timeless predicates, which are, by definition, true at all times. Therefore, they are allowed to appear in the preconditions of an operator, but not in the add or delete lists. As a consequence, if the user tries to add a timeless predicate in the preconditions part of an operator, it will automatically appear in the effects part, so the add and delete lists will not be affected. Furthermore, if the user tries to set a predicate timeless, checks will be performed to determine if this operation is allowed. Finally, timeless predicates are not allowed to appear in a problem. In all above cases, error messages occur in order to warn the user and help them correct the domain inconsistencies.

Another example is that of constants. Checks are performed to confirm that the class of a constant has already been defined before the user attempts to use the constant in an operator or a problem. Furthermore, additional checks are performed about the types of arguments, similar to those performed for simple objects.

4.6. Translation to and from PDDL

The capability to export the domains and problems designed in the tool to PDDL constitutes another important feature. All of the design elements that the user has imported in the domain, such as predicates and operators, along with comments, are exported to a PDDL file, which is enhanced with the appropriate requirements tag. The user is offered the option to use typing, therefore, the same domain can produce two different PDDL files, one with the :typing requirement and one without it. Details about exporting are presented in the remainder of the paragraph.

Despite the fact that a class in the visual tool always represents the same notion, that is, the type of domain objects or parameters, it takes different forms when it comes to exporting the domain. In case the requirement typing is declared, the class name is included in the (:types) construct of the domain definition, and for each object, parameter and constant a type must be declared. In case typing is not used, classes are treated as timeless unary predicates, that is, predicates that are always true and appear in the corresponding part of the domain definition. In addition, for each parameter in an operator, a precondition that denotes the type of the parameter must be added in the PDDL definition, although it does not apper visually in the tool. Likewise, for each object, a new initial literal denoting the type of this object must be included in the problem definition.

The elements in the Ontology Editor are combined together in order to formulate the domain constructs in the syntax that the language imposes. For example, relations, connections and, if typing is used, classes are combined to formulate the predicates construct. Likewise, functions and derived predicates constructs are formed. As far as constants are concerned, they may appear in the place of parameters in operators and objects in problems, and they also appear in the special construct (:constants) in the domain definition.

Exporting the operators is quite more complicated, because a combination of several elements of the Operator Editor and the Ontology Editor is needed. Slight changes occur to an operator definition depending on whether the :typing requirement is declared.

Finally, exporting the problems is quite similar to exporting the operators, but the problems are stored in a different PDDL file. Therefore, numerous problems can be defined for the same domain. If maps are used, care must be taken to include the information they embody in the list of predicates included in the initial state. Furthermore, if functions are used, their initial values provided by the user in the Problem Editor will be part of the declaration of the initial state of the problem, in the corresponding construct.

The visual tool also offers the feature of importing planning domains and problems expressed in PDDL, visualizing them, and thus enabling the user to manipulate them. However, importing PDDL is subject to some restrictions. The most important is that the domains and problems must declare the :typing requirement. If no typing is used, syntax is not enough, and semantic information is necessary in order to discriminate types of objects from unary predicates.

4.7. Interface with Planning Systems

As the tool is intended to be an integrated system not only for designing but for solving planning problems as well, an interface with planning systems is necessary. This is achieved by providing the ability to discover and communicate with web services which offer implementations of various planning algorithms. Therefore, a dynamic web service client has been developed as a subsystem. The requirement for flexibility in selecting and invoking a web service justifies the decision to implement a dynamic client instead of a static one. Therefore, the system can exploit alternative planning web services according to the problem at hand, as well as cope with changes in the definitions of these web services.

The communication with the web services is performed by means of exchanging SOAP messages, as the web service paradigm dictates. However, in a higher level, the communication is facilitated by the use of the PDDL language, which constitutes the common ground between the visual tool and the planners. An additional

advantage of using PDDL is that the visual tool is released by the obligation to determine the PDDL features that a planner can handle, thus leaving each planning system to decide for itself.

The employment of web services technology for implementing the interface results in the independency of the visual tool from the planning or problem solving module. Such a decoupling is essential since new planning systems which outperform the current ones are being developed. Each of them can be exposed as a web service and then invoked for solving a planning problem without any further changes to the visual tool or the domains and problems already designed and exported as PDDL files.

5. Conclusions and Future Work

In this paper a visual tool for defining planning domains and problems was proposed. The tool offers an efficient user interface, as well as interoperability with PDDL, the standard language for planning domain definition. The elements represented in the tool cover a wide range of the language, while the user is significantly facilitated by the validity checks performed during the design process. The use of the tool is not confined to designing planning problems, but the ability to solve them by invoking planners implemented as web services is offered as well. Therefore, the tool is considered an integrated system for designing and solving planning problems.

Out future goals include the extension of the tool in order to represent even more complex PDDL language elements, as well as other planning approaches, such as HTN (Hierarchical Task Network) planning. Such an extension is believed to broaden the range of real world problems that can be represented and solved by the tool. Visual representation of produced plans, along with plan metrics are also among our imminent goals.

Acknowledgements

This work was partially supported by a PENED program (EPAN M.8.3.1, No. 03EΔ73), jointly funded by the European Union and the Greek government (General Secretariat of Research and Technology).

References

[1] T. L. McCluskey, D. Liu, Ron M. Simpson, “GIPO II: HTN Planning in a Tool-supported Knowledge Engineering Environment”, International Conference on Automated Planning and Scheduling (ICAPS), 2003

[2] Wilkins, D. E., Lee, T. J. and Berry, P., Interactive Execution Monitoring of Agent Teams, Journal of Artificial Intelligence Research, 18 (2003), pp. 217-261.

[3] D. Vrakas, I. Vlahavas, “A Visualization Environment for Planning”, International Journal on Artificial Intelligence Tools”, Vol. 14 (6), 2005, pp. 975-998, World Scientific.

[4] Ghallab, M., Howe, A., Knoblock, C., McDermott, D., Ram, A., Veloso, M., Weld, D. and Wilkins, D., "PDDL -- the planning domain definition language". Technical report, Yale University, New Haven, CT (1998).

[5] Fox, M. and Long, D., "PDDL2.1: An extension to PDDL for expressing temporal planning domains". Journal of Artificial Intelligence Research, 20 (2003), 61-124.

[6] Gerevini, A. and Long, D., "Plan Constraints and Preferences in PDDL3", Technical Report R.T. 2005-08-47, Department of Electronics for Automation, University of Brescia, Italy.

[7] Fikes, R. and Nilsson, N. J., STRIPS: A new approach to the application of theorem proving to problem solving, *Artificial Intelligence*, Vol 2 (1971), 189-208.