Charles University in Prague

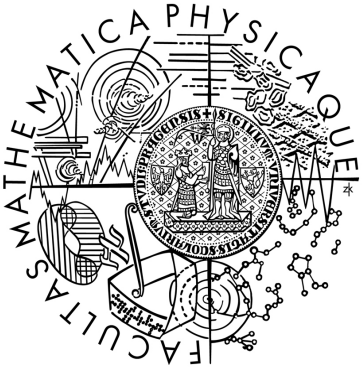
Faculty of Mathematics and Physics

BACHELOR THESIS

2011 **Radoslav Glinský**

Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Radoslav Glinský

Visualization and Verification of Plans

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the bachelor thesis: Doc. RNDr. Roman Barták, Ph.D.

Study programme: Computer Science
Specialization: General Computer Science

Prague 2011

I would like to thank Doc. RNDr. Roman Barták, Ph.D., supervisor of the bachelor thesis, for his professional help and friendly approach.
I would like to thank my parents for their love and trust.

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In …..... Date .....

Názov: Vizualizácia a verifikácia plánov

Autor: Radoslav Glinský

Katedra: Katedra teoretickej informatiky a matematickej logiky

Vedúci bakalárskej práce: Doc. RNDr. Roman Barták, Ph.D.

Abstrakt: Analýza plánov je dôležitou súčasťou kompletných plánovacích systémov. Aby sme umožnili užívateľom orientovať sa aj vo väčších plánoch, vytvorili sme program, ktorý pomáha s analýzou a vizualizáciou plánov. Program sa volá VisPlan - Interactive Visualization and Verification of Plans. VisPlan je neoddeliteľnou súčasťou bakalárskej práce, keďže prakticky implementuje jej vizualizačné a verifikačné riešenia. VisPlan nachádza a zobrazuje kauzálne väzby medzi akciami, identifikuje možné chyby v pláne (a tak overuje jeho korektnosť), zvýrazňuje nájdené chyby v pláne a umožňuje užívateľom interaktívne modifikovať plán, a teda opraviť chyby v pláne alebo ho jednoducho vylepšiť.

Kľúčové slová: Plánovanie, Umelá inteligencia, PDDL, Verifikácia

Title: Visualization and Verification of Plans

Author: Radoslav Glinský

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the bachelor thesis: Doc. RNDr. Roman Barták, Ph.D.

Abstract: Plan analysis is an important part of complete planning systems. In order to make even larger plans transparent and human readable, we have developed a program which helps users with the analysis and visualization of plans. This program is called VisPlan - Interactive Visualization and Verification of Plans. VisPlan is an inevitable part of this thesis as it practically implements its plan verification and visualization solutions. VisPlan finds and displays causal relations between actions, it identifies possible flaws in plans (and thus verifies plans' correctness), it highlights the flaws found in the plan and finally, it allows users to interactively modify the plan and hence manually repair the flaws or just fine-tune the plan.

Keywords: Planning, Artificial Intelligence, PDDL, Verification

Table of Contents

Introduction

Planning is a specific branch of artificial intelligence aim of which is, shortly described, finding a sequence of actions leading from the initial state to a desired goal state. In the recent years, various kinds of autonomous, unmanned robots and machines have been developed and this trend is highly expected to accelerate in the future. Systems with a certain amount of autonomy inevitably need a mechanism determining their activity in order to fulfill a goal. In other words, they need a planner. Therefore, construction of effective and scalable planning systems is gaining importance in a fast pace.

Together with planners, plan analysis is a necessary part of complete planning systems. With a growing complexity of plans, in terms of number of actions and causal relations in plans, the analysis becomes more and more time consuming process. In fact, plans with hundreds of actions are practically unreadable for humans. Having the good tools for plan analysis, a user can easily check the correctness of an examined plan, effectively retrieve a general overview of the plan, find its flaws, identify possible enhancements, etc.

Though a number of available planners grows, the number of plan analysis tools is still limited. The aim of this thesis is to fill this gap, namely to provide a complex plan analysis solution. The thesis should propose the way how to handle different types of plans and how to visualize them in a user-friendly way. Along with that, the thesis should elaborate a proper plan verification with respect to different plan types, their semantics and concrete planning domains. The thesis should also identify already existing plan analysis tools, analyze them and provide a novel functionality upon the comparison and, primarily, contribute to a planning community in a positive way. Eventually, all the solutions of the thesis should be implemented by a sofware program accompanying the thesis.

The result of the thesis is VisPlan - Interactive Visualization and Verification of Plans. It is a program written in Java programming language and it practically implements its plan verification and visualization solutions. The main goal of

VisPlan is to make even larger plans transparent and human readable. VisPlan parses and consecutively visualizes a plan in the way based on its type. VisPlan is capable of verifying the plan, during which it identifies causal relations between actions and possible flaws in the plan. Upon verification it updates the visualized plan. The plan visualization is implemented as a mathematical graph. Actions are illustrated as vertices, causal relations as edges in the graph. As part of the analysis, VisPlan displays information about the actions during plan execution, flaws possibly found in the plan are highlighted. VisPlan goes even further - it allows users to interactively modify the plan and hence manually repair the flaws or fine-tune the plan if they wish. Finally, as VisPlan is a graphical desktop application, it provides a comfortable way of using the previously mentioned functionality.

At the beginning, the thesis introduces the general aspects of artificial intelligence. Planning as a certain subfield of artificial intelligence is discussed in the second chapter. We focus particularly on answering which types of planning problems the thesis deals with and how they are represented by computers. Then, in the next chapter, we describe the history and syntax of PDDL (Planning Domain Definition Language), a de facto standard language for planning problem notation. All the planning problems we want to analyze by VisPlan need to be formalised in PDDL. The fourth chapter compares functionality of VisPlan to other available plan analysis tools, namely itSimple and GIPO. Especially the differences between VisPlan and each of them are accented. The fifth chapter discusses visualization and verifications solutions of the thesis, each respectively to the type of the plan. It describes semantics, used algorithms and, if there had been a multiple choice in the way of implementation, gives reasons why the chosen one has been selected. The last two chapters get closer to the program usage from user perspective and, finally, describe some implementation details.

The program is under continuous development and all the relevant information plus the up-to-date version of the software can be downloaded from:

`http://glinsky.org/visplan`

1. Artificial Intelligence

Nowadays, we live in the world full of various technological advantages compared to our ancestors, such as planes, televison, phones, tractors and we may continue. They all help to make our everyday lives better. The impossible has come true, the unbelievable has come obvious and many difficulties have just dissappeared; in the fields of transportation, communication, entertainment, careers, living standard and so on.

The most of our today's technical appliances is, however, very simple in the means of determination. Generally, on the exact inputs they behave in deterministic way, they don't think and do accept only a predefined set of inputs. In the real world, however, where a single event is influenced by many, initially unknown factors, where possible inputs cannot be accurately described, the needs go even further. These are the cases where the artificial intelligence (AI) succeeds.

Nowadays, AI is still a new science, as the first mentionings come from the late fifties of the twentieth century. The goal of AI is to create and understand intelligent systems. This involves intelligent behavior, learning, adaptation to unknown environments, solving unpredictable situations.

Achievements of the AI are used in many disciplines. Recognition of human speech, computer opponents replacing human ones when playing games, robotic vacuum cleaner's planning its movement, email spam automatic filtering based on the user previous labeling, are only the very few examples. AI is really a universal field.

1.1. Definition of AI

There is not a unique definition of artificial intelligence. However, the most of the definitions use two approaches. The one approach is a thought process, so that artificially intelligent system think, whereas the other one is more interested in behavior of such system. The definitions, in addition, are not consistent when defining what really means intelligence. Is the intelligent system simulating the human behavior? Or does it always try to be rational (meaning to always choose

the best possible option, which is, certainly, not always the human case). Having considered the previous approaches, Table 1 summarizes definitions of (artificially) intelligent systems (Russel and Norvig, 2003):

	Human performance	Rationality
Thought processes	Systems that think like humans	Systems that think rationally
Behavior	Systems that act like humans	Systems that act rationally

Table 1: Various definitions of AI based on different approaches

1.2. Acting rationally approach

From all the approaches to AI which has developed so far, an approach we will adhere to in this thesis is developing systems that act rationally. In such systems there's always someone (or something) who performs the actions. We will call him the agent[1]. Furthermore, such agent performs only the rational actions. That means that he/it maximalizes the possible gain/outcome by always choosing the proper actions.

The approach we have chosen (systems with a rational agent) has some very important advantage which is the fact that the rational behavior can be clearly specified in majority of cases. Human behavior, in contrast, sometimes cannot be be specified at all (people vary a lot from one another plus the same man may act differently in the same situation, according to his/her mood, etc.). The mentioned advantage is crucial when the problem to solve needs to be represented mathematically. Once mathematically represented, computer engineering then provides mechanism (lots of memory and speed) in order to solve the problem effectively.

1 agent comes from the Latin agere, to do

2. Planning

Planning is a certain part of artificial intelligence science. Assumed we are given an initial state of the “world”, it could be shortly described as a process of decision making to find out a sequence of actions needed to accomplish given goal. The result of planning process is a plan - an ordered sequence of actions. Actions of the plan consecutively modify the state of the “world”. After application of the entire action sequence - the plan - the final world state desirably meets the goal - a set of certain facts that should apply in the final state.

2.1. The language of planning problems

In order to represent the real world planning in an artificial world, it is common to simplify the model. The thesis will therefore consider only classical planning model. This model ensures that the world is deterministic, finite, fully observable (we have full information about the world) and static (there are no actions in the world other than the agent’s). If the model is discrete at the same time (discrete in time and actions’ applications) we talk about STRIPS[2]-like planning. Otherwise we consider temporal planning (actions have duration, preconditions and effects specified arbitrarily within action’s application, ...). The thesis handles both STRIPS-like and temporal plan types. This is demonstrated by VisPlan - the software implementation of the thesis.

The language of representation of planning domains and problems should be expressive enough in order to be able to describe many different kinds of planning problems. At the same time it should be somehow delimited so that planners can be designed for solving the general planning problems and be effective, too.

2.1.1. States

Any state of the world will be represented as a conjunction of literals. We will assume the “closed” world what means that if a literal is not explicitly stated in the world state it means that it does not hold in the state. As the consequence of this assumption, any state includes only positive literals. In addition, literals must be

2 STRIPS is an abbreviation of STanford Research Institute Problem Solver

ground (no variables allowed) and function-free. The following example may represent a state in the “monkey” world:

$$\text{monkey(monkey1)} \wedge \text{monkey(monkey2)} \wedge \text{place(place1)} \wedge \text{place(place2)} \wedge \text{place(place3)} \wedge \text{eatable(banana1)} \wedge \text{at(monkey1,place3)} \wedge \text{at(monkey2,place1)} \wedge \text{at(banana1,place3)} \wedge \text{happy(monkey2)}$$

More formally[3], let L (language) be a finite set of possible literals:

$$L = \{p_1, \ldots, p_n\}$$

Then state s is a subset of L containing literals which hold:

$$p \in s \Rightarrow \text{literal } p \text{ holds in } s$$

$$p \notin s \Rightarrow \text{literal } p \text{ does not hold in } s$$

2.1.2. Operators

An operator is an action schema. It means that an action is created (we say instantiated) from an operator when all variables of the operator are substituted with concrete arguments (we say grounded). Each operator specifies its preconditions and effects. For example, an operator for eating a banana in the “monkey” world could look like this:

$$\begin{aligned}&\text{Action(eat(m,b,p),}\\&\quad\text{Precond: monkey(m)} \wedge \text{eatable(b)} \wedge \text{place(p)} \wedge\\&\qquad\qquad\text{at(m,p)} \wedge \text{at(b,p)}\\&\quad\text{Effect: } \neg\text{at(b,p)} \wedge \text{happy(m)}\end{aligned}$$

More formally, let A be a set of possible actions, then:

$$a \in A$$

$$a=(\text{precond}^-(a),\text{precond}^+(a),\text{effects}^-(a),\text{effects}^+(a))$$

$$\text{precond}^-(a),\text{precond}^+(a),\text{effects}^-(a),\text{effects}^+(a) \subseteq L$$

$$\text{precond}^-(a) \cap \text{precond}^+(a) = \emptyset$$

$$\text{effects}^-(a) \cap \text{effects}^+(a) = \emptyset$$

Action a is applicable at state s if:

$$\forall\ p \in \text{precond}^+(a) \Rightarrow p \in s$$

$$\forall\ p \in \text{precond}^-(a) \Rightarrow p \notin s$$

3 Planning problem language formalism in this section has been inspired by: (http://kti.mff.cuni.cz/~bartak/planovani/index.html, 2011)

2.1.3. Planning domain

Planning domain Σ over language L is a trio (S, A, γ), such that:

$S \subseteq P(L)$, S is a set of possible world states

Transitional function γ describes how the resulting world state looks like after application of the given action to a specific world state:

$\gamma(s,a) = (s - effects^{-}(a)) \cup effects^{+}(a)$, if a is applicable at s

2.1.4. Goals

A goal g is a partially specified desired world state in the planning problem:

$g \subseteq L$

$S_g = \{s \in S \mid g \subseteq s\}$ is set of fully specified goal states

For example, the state given as example in the previous "States" section satisfies the partially specified goal $at(monkey2, place1) \wedge happy(monkey2)$.

2.1.5. Planning problem

Planning problem P is a trio (Σ, s_0, g), such that:

$\Sigma = (S, A, \gamma)$ is a planning domain over language L

s_0 is an initial world state, $s_0 \in S$

$g \in S_g$

2.2. Plan

Plan π is a sequence of actions $\langle a_1, a_2, \ldots, a_k \rangle$

$k = |\pi|$ is the length of the plan

We define world state induced by plan using transitional function γ:

- k = 0:

  $\gamma(s, \pi) = s$

- k > 0 and a_1 is applicable at s

  $\gamma(s, \pi) = \gamma(\gamma(s, a_1), \langle a_2, \ldots, a_k \rangle)$

- not defined otherwise

Plan π is the solution for planning problem P if and only if:

$g \subseteq \gamma(s_0, \pi)$

3. Planning Domain and Problem Representation

For a long time there had been no standard language for representation of planning problems. Actually, this had a negative influence on the whole field of planning, as engineers, developers and enthusiasts from all around the world had no common means of communication among themselves.

Finally, in 1998, Drew McDermott and others created a language called PDDL - the Planning Domain Definition Language. At the time of its creation, the main motivation was to unify requirements and input format for planners taking part in the planning competition IPC (International planning competition). The competition was a success (it has been organized regularly since then) and it was right the PDDL what laid the fundamentals for it.

Since the PDDL has been introduced, it has been gaining popularity and, in fact, it has become a standard language for representing the planning problems. Many new features has been added to the PDDL during the last years in order to enlarge set of possible planning domains it can describe. And, in opposite direction, what PDDL can describe is now being treated as the standard and therefore has an impact on the whole planning community.

The language has changed a lot since it was initially introduced in 1998 and is still under development. Throughout the history, several major enhancements in the PDDL syntax has been featured, each of which usually induced a new version. As the environment presented in this thesis has a close connection to the PDDL (VisPlan parses PDDL files as its input), a brief description of the most important syntactic elements will be provided in the following sections with respect to the version in which they were introduced.

Planning tasks specified in PDDL are separated into two files:

1. A domain file for predicates and actions

   For concrete example of domain file see Appendix A (STRIPS-like domain) and Appendix C (temporal domain).

2. A problem file for objects, initial states and goal specifications.

   For concrete example of problem file see Appendix B (STRIPS-like domain)

and Appendix D (temporal domain).

3.1. PDDL 1.2

PDDL 1.2 (Ghallab et al., 1998) is an original version and was used for the first IPC competition. It has introduced the basic concepts of the language. The language has a LISP-like syntax. A set of features used in the PDDL file is listed at the beginning of the file, after the `:requirements` keyword, for example:

```
(:requirements :typing :durative-actions)
```

A domain is structured into components by keywords, such as `:predicates` or `:actions`. We will provide a brief explanation of these components in the next paragraphs. We focus mainly on the components which can be hanled by VisPlan program. Appendix A and B provide samples for most of the discussed elements.

3.1.1 Variables

Variables in the PDDL have the same meaning as in any other language. They are present in the parameters of actions[4], as well as in other functions. They start with a question mark `(?variable)`.

3.1.2. Constants

Constants in the PDDL can be used at the same places as variables, however, with a big difference that they cannot be substituted.

3.1.3. Predicates

Conjunction of predicates represent a state of the world. They carry information about the objects in the world and relations between them, too. A simple example of the predicate might look like: `(smaller ?x ?y)`. In addition, predicates are used in PDDL's actions as preconditions and effects.

3.1.4. Actions

Actions are the means how the world state is changed. Concrete PDDL action definition will be demonstrated on the following example. The action comes from the blocks world planning domain:

4 Actions in PDDL domain file can be treated as operators explained earlier. They are not the actual actions, but rather represent an action schema.

```
(:action putdown
   :parameters (?block)
   :precondition (and (holding ?block))
   :effect (and (clear ?block) (arm-empty) (on-table ?
   block) (not (holding ?block))))
```
[5]

Actions must specify the following:

- name: `putdown`
- parameters: `(?block)`
- preconditions: `(and (holding ?block))`
- effects: `(and (clear ?block) (arm-empty)`
  `(not (holding ?block)) (on-table ?block))`

3.1.7. Planning problem

Planning domain is usually defined in a separate file. This enables us to have many different planning problems sharing a single planning domain.

In the PDDL problem file we include a list of objects present in the world (typed or without types):

```
(:objects rod1 rod2 rod3 d1 d2 d3)
```

The initial situation is declared as the list of predicates which hold at the initial world state. Predicates not listed explicitly do not hold at the initial state[6].

```
(:init (smaller rod1 d1) (smaller rod1 d2) ... (clear
rod2) (clear rod3) (clear d1) (on d3 rod1) (on d1 d2))
```

An example of a goal might look like the following. Listed predicates must be grounded:

```
(:goal (and (on d3 rod3) (on d2 d3) (on d1 d2)))
```

3.2. PDDL 2.1

PDDL 2.1 (Fox and Long., 2003) is based on PDDL 1.2. It only adds some new fetures supporting temporal planning. Numeric state variables has been introduced, as well as durative actions which enable concurrency. See Appendix C and D for examples and overall reference.

---

5 The blocks world is one of the most famous planning domains. This domain consists of a set of cube-shaped blocks sitting on a table or on other blocks. A robot arm then picks up blocks and moves them to different positions in order to build desired stacks of blocks.

6 Closed world assumption.

3.2.1. Functions

Functions in the PDDL present a way how to assign a numerical value to a set of arguments, for example the following defines a numeric function `drive-time` with 2 parameters `from` and `to` of `location` type:

```
(:functions (drive-time ?from ?to - location))
```

Compound expressions created from simple functions (like the example above) and arithmetic operators were introduced as well.

3.2.2. Comparisons and assignments

Comparisons are used among preconditions of the actions. After they numerically evaluate both of their sides they decide whether the condition is satisfied.

Moreover, assignments using operators such as assign, increase and decrease are also possible. The value for the assignment should be stated in the problem file using the following construct: `(= (drive-time loc3 loc1) 7.1)`.

3.2.3. Durative actions

Durative actions bring concurrency to the plans. In the planning domain a new definition for the actions' duration must be stated:

```
:duration (= ?duration (drive-time ?from ?to))
```

Conditions and effects of durative actions can be examined at start, at end or over all of the action's interval. See Appendix C for example.

3.2.4. Plan metrics

Plan metrics determine how should a planner choose actions in the plan in order to maximize/minimize a plan cost based on the metrics criteria. In the PDDL this can be specified using the following language:

```
(:metric minimize (total-time))
```

3.3. PDDL 3.0 and PDDL+

Features of PDDL 3.0 and PDDL+ are not covered by VisPlan so we will only mention a few. In PDDL 3.0 (Gerevini and Long, 2005) declarations about plan quality and plan trajectories have appeared (the states a plan have to go through). Real-time systems and probabilistic planning featured in PDDL+.

4. Existing software

Though the number of planners rapidly grows, the number of available tools for user interaction with planners is still limited. However, there are several publicly available programs dealing with such issues and provide graphical user interface supporting the planning process. The most well-known are shortly described in the next paragprahs. For each described tool the comparison to VisPlan is stated as well in order to emphasise the contribution of VisPlan to the planning community.

4.1. itSIMPLE[7]

Integrated Tools Software Interface for Modeling PLanning Environments (Vaquero et al., 2010)

itSIMPLE is an open source project implemented in Java, available under the GNU General Public License version 3. The tool has been designed to give support to users during the construction of a planning domain application mainly in the initial stages of the design life cycle. These initial stages encompass processes such as domain specification, modeling, analysis, model testing and maintenance. It provides a user-friendly GUI for modeling and analyzing many planning domains at the same time. Specified domain and problem are nicely visualized to users. For these purposes, a special use of UML (Unified Modeling Language) has been developed

XML (eXtended Markup Language) is used as an intermediate language that can support automatic translation from UML to other representations as well, such as PDDL or Petri Nets.

A model can be generated into PDDL language. PDDL representation gives users an opportunity to test their models with several general planners (such as Metric-FF, FF, SGPlan, MIPS-xxl, LPG-td, LPG, hspsp, SATPlan, Plan-A, blackbox, LPRPG, Marvin). These planners are bundled within the software and using them is rather straightforward. Moreover, plans can be specified manually by users, too.

7 The software has been downloaded and more information retrieved from: (http://dlab.poli.usp.br/twiki/bin/view/ItSIMPLE/OverView, 2011)

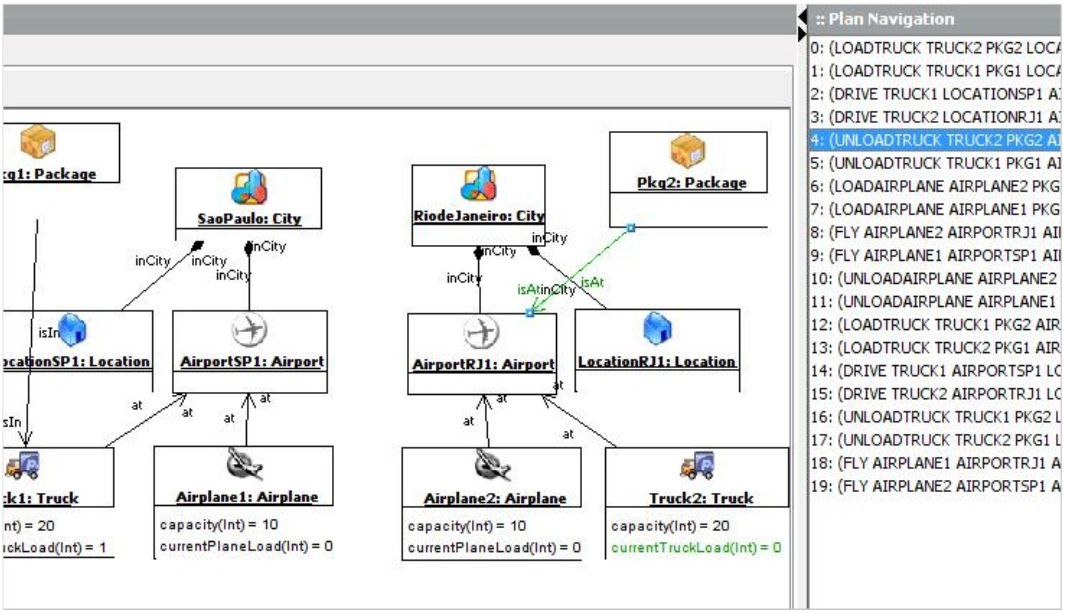


Figure 1: Plan Analysis in itSimple program using Movie Maker.

Once a domain and problem are modeled, itSimple also gives an interface for plan analysis and management (Figure 1). It is possible to observe the behavior of the model during the simulation of plans (given by users or by planners). This is done by using sequence of snapshots of the plan. The interface visualizes relations (predicates) between objects which are true before and after each action is performed. As illustrated in Figure 1, objects are assigned a graphical appearance, relations between objects are shown as arrows and those which are being changed by current plan action are highlighted.

itSimple provides variable observation in charts as well. Each attribute of an object can be continuously tracked in a well-arranged chart.

4.1.1. Comparison to the VisPlan

Compared to the VisPlan, following functionality is not covered by itSIMPLE:

- PDDL domain and problem files are not accepted as an input
- there is no verification of a plan (if the given plan is not valid, missing preconditions are not reported, nor any flaws are recognized)
- causal relations of actions are not shown
- preconditions of actions are not shown

itSimple is an effective tool for modeling planning domains. However, it does

assume that given plans are 100% valid. Regarding to plan visualization, the tool shows a sequence of world states (facts that apply in each state). It does not recognize causal relations[8] of actions, nor gives a compact overview of actions' preconditions and effects.

4.2. GIPO[9]

Graphical Interface for Planning with Objects (Simpson et al., 2007)

GIPO allows a user to create new domain models or import and change old ones. Recognized domains are either classical or hierarchical or requiring durative actions.

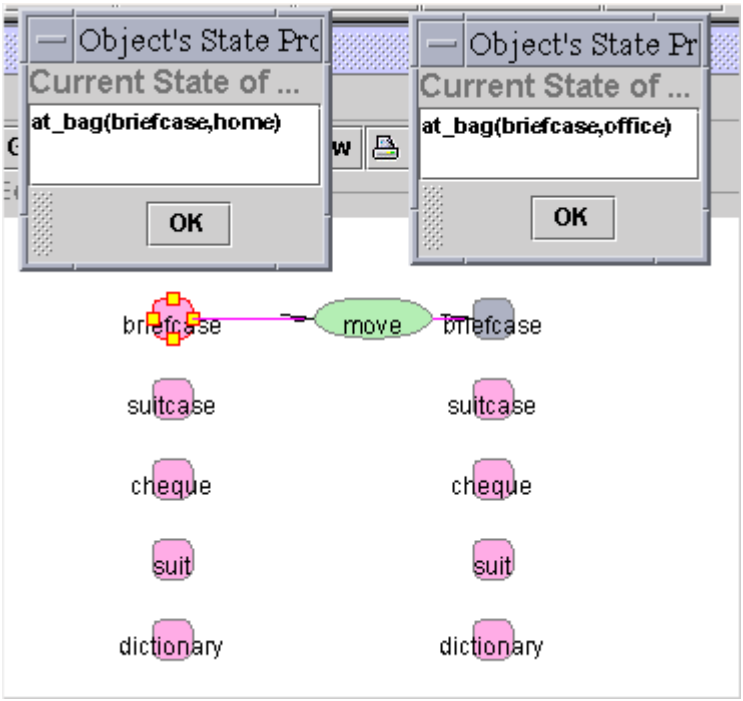


Figure 2: Plan Analysis in GIPO program using Stepper.

GIPO program provides:

- Modeling object types, predicates, operators, tasks (init and goal states) using graphical tools. Models are represented in the OCL (Object Centered Language) language. As the language name suggests, GIPO models planning domains from an object's perspective, which is describing how

8 Causal relation is a relation between two actions where one or more effects of one action are consequently used as preconditions for the other action.

9 The software has been downloaded and more information retrieved from: (http://scom.hud.ac.uk/planform/gipo, 2011)

actions can change a state of the object.

- Validation of domain specification.
- Stepper (see Figure 2) - an interface enabling manual planning. Meaning the user has to choose and instantiate an operator. The Stepper shows objects present in the planning problem. If a particular object is clicked a window appears describing which predicates apply for this object at the given state. When the user chooses an action and if all necessary preconditions are met, the resulting state (after execution of the chosen action) will be generated. The changed objects then appear in a different color and have different sets of predicates applying at that state.
- A hierarchical planner HyHtn bundled within the program.
- Plan animator (not for durative actions): similar to the Stepper. The difference is that the actions are taken as the output of an integrated planner.
- Tools for exporting/importing PDDL models.

4.2.1. Comparison to the VisPlan

Compared to the VisPlan, following functionality is not covered by GIPO:

- overall world state at a specific plan execution time cannot be retrieved
- causal relations of actions are not shown
- the visualized objects cannot be moved
- actions in the plan cannot be modified/created/deleted
- durative actions not supported

GIPO software is mainly used for creating/updating planning domains. In contrast to the thesis, the visualization is very low level and doesn't tell much about the plan. As illustrated in Figure 2, the Stepper shows layers of (always the same) objects. This approach is not acceptable if the number of objects is higher (e.g. more than twenty). In order to show the state of the concrete object (meaning all the predicates that apply to the object at given layer), the object needs to be double-clicked and a new window carrying this information pops-up. Therefore, the progress of object's state cannot be retrieved easily, as multiple pop-up

windows (each carrying the state for the same object but different layer) may look disorganized. In addition, the necessity to open and close all windows is rather unhandy.

4.4. VisPlan contribution

Previously mentioned tools, itSimple and GIPO, are both effective tools for modelling and updating planning domains. However, their plan analysis lacks some handy features such as:

- recognizing causal relations of actions
- compact overview of actions' preconditions and effects
- support for plans with flaws
- information about world state at a specific plan step
- a user friendly interface to modify, insert, and delete actions in a plan and to re-verify the plan in real-time

VisPlan focuses on all above features.

5. VisPlan Functionality

Shortly described, VisPlan is a graphical application (Figure 3) written in Java with the ultimate goal to visualize any plan, to find and highlight possible flaws, and to allow the user to repair these flaws by manual plan modification.

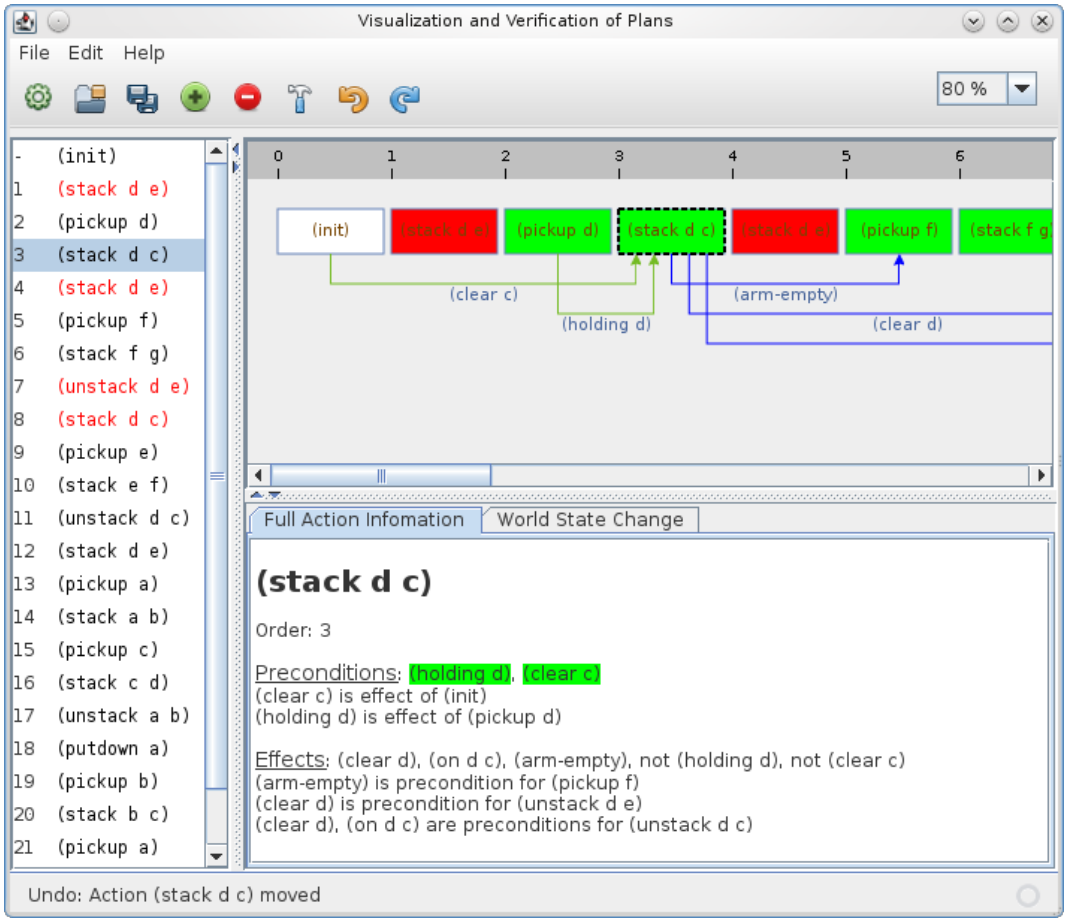


Figure 3: Graphical user interface of VisPlan.

5.1. Program Input

VisPlan works with three types of files that the user should specify as program input:

- planning domain file in PDDL
- planning problem file in PDDL
- plan file specified in text format

VisPlan supports STRIPS-like plans and temporal plans. The program recognizes the plan type (strips/temporal) automatically and verifies and

visualizes it based on its type. The plan type is determined by the planning domain – durative actions indicate a temporal plan, actions with no duration indicate a STRIPS-like plan. The following PDDL requirements are currently supported in the program: strips, typing, negative-preconditions, equality, durative-actions.

Planning domain and problem need to be syntactically correct and mutually consistent (separately parsed planning domain and problem files can be linked with each other). Otherwise, visualization and verification is not performed and errors from the PDDL parser are displayed. Sometimes, PDDL parser encounters errors and issues which are not critical. In these cases, warning and non-critical error messages are displayed and the program continues. Recognized plan actions are given in the following format:

```
start_time: (action_name param1 param2 …) [duration]
```

In the plan file each action is supposed to be on a separate line. The parser recognizes the lines and creates actions given only in the above mentioned format. Other lines are ignored. Eventually, a modified plan can be saved either to the original file or to a new text file.

5.2. Verification

Plan verification is automatically executed after the plan is initially loaded and then after each user interaction modifying the plan. The verification process is based on simulation of plan execution and the main idea is to incrementally construct “layers” of facts. Each fact layer is determined by a corresponding set of facts and an action due to which the layer has been created.

At the beginning of the verification, all possible facts (grounded predicates) are instantiated. This domain-specific data remains fixed and is computed only once at the beginning; re-verifications do not change the data. This attitude permits us not to manipulate with the facts during the whole verification process, but to work only with the indexes to the array of grounded facts. Because of that, operations like checking if an action is applicable, application of action’s effects, finding missing conditions, etc. are just logical bit-sets operations (where one bit-set has its bits set to true at indexes corresponding to the selected grounded facts). Such operations are very fast.

Unlike facts, only actions present in the plan are grounded (meaning related to an operator with grounded conditions and effects). The operator is found based on matching the planning-domain operator and concrete parameters of the action. As mentioned in the previous paragraph, conditions and effects of the grounded operator are represented by bit sets (pointing to the fix array of grounded facts). The verification process makes sure it has a matching operator available for each examined plan action (otherwise, for instance when a user adds a new action, the verification process additionally finds and stores the operator). Actions, which do not comply with any operator definition, are marked as invalid and omitted from the verification. Nevertheless, such actions are still displayed (but distinguished from others by a different colour and marked as invalid).

There are two special “actions” artificially added into the plan. They are called “init” and “goal” and their aim is to represent the initial state and the goal. A classical plan-space approach is used to define these actions. The init action has empty preconditions and the facts that apply at the initial state are considered as its effects. The goal action has empty effects and the set of facts that need to be satisfied at the final world state are considered as its preconditions. By treating the initial state and the goal as regular plan actions we are able to recognise causal relations also at the margins of the plan without any further work. This way we easily find dependencies on the initial state and, eventually, marking the “goal” action as non-applicable means that the goal conditions are not satisfied.

5.2.1. Finding action’s matching operator

In order to find a matching operator for an action we have to go through the planning-domain operator expressions and find an operator which:

- matches action’s name
- matches the number of action’s parameters
- each action’s parameter belongs to a (typed) domain of respective operator’s variable, where the domain is a set of concrete objects in the planning problem such that object’s type is equal to the variable’s type (or variable’s deduced type)

Upon correspondence, every couple (variable, parameter) is bound and added into a “substitution” object. This substitution is consequently applied on the operator’s conditions and effects, thus ensuring they are grounded since then. Afterwards, the algorithm separately converts the grounded conditions’ and effects’ compound expressions into a set of trivial expressions (for STRIPS-like actions each such expression is either a literal or, for durative actions, a timed expression including just one literal).

In the final step, an operator is created based on the trivial expression set from the previous step. For STRIPS-like actions the following bit-sets are instantiated: positive preconditions, negative preconditions, positive effects, and negative effects. If the literal from the literal set is an atomic formula, the index of atomic formula (which is, indeed, a grounded fact, one of the facts in the initially created array of facts) is added to positive preconditions/effects bit-set. On the other hand, if the literal is a “not (atomic formula)”, the index of atomic formula is added to negative preconditions/effects bit-set.

For durative actions the literal is obtained from a timed expression (one of the following: “at start (literal)”, “over all (literal)”, “at end (literal)”). And, similarly, index of literal’s atomic formula is added to one of the following sets: at start conditions, over all conditions, at end conditions, at start effects, at end effects (each positive or negative depending on the literal).

Artificial operators for special “init” and “goal” actions are constructed as well. Conditions for the “goal” operator are obtained in the same way as conditions for any regular plan action with an exception that the goal expression is separately taken from the parsed PDDL problem file. In contrast to the “goal” operator, for the “init” operator there is already a predefined and grounded set of facts (atomic formulas) from the separately taken init expression. These facts (represented as a bit-set) are then assigned to the “init” operator’s effects. In addition to grounded facts, the init PDDL expression may contain equality comparison functions, for instance:

```
(= (drive-time l1 l2) 4.3)
```

Function name plus its arguments (the first argument of the above equality

comparison function) is assigned a numerical value representing time duration (the second argument of the above equality comparison function). Couple (duration function, duration value) is stored and used when creating an operator matching the durative action. At this time, the duration of action is obtained from the parsed PDDL domain file, grounded (by the same substitution as action's conditions and effects) and searched within previously stored duration functions. Duration value of the found function is assigned to the matching operator of the currently manipulated action.

5.2.2. Semantics of STRIPS-like plans

In contrast to temporal plans, semantics of STRIPS-like plans is really straightforward. The order of actions is exactly specified by the sequential plan. In fact, this order is clearly determined by the order of actions in a file accepted as an input to VisPlan (from top to the bottom). Internally, the order is maintained in a linked list. Since we sometimes need to iterate over the actions in a descending (opposite) order a double-linked list is used, thus enabling descending iterations and access to the last action naturally.

Providing the sequence of actions in STRIPS-like plans, all the preconditions' checks and possible world state changes occur instantaneously, at the points when actions are consecutively handled. Preconditions of an action (or, eventually, the goal conditions) are checked against the current world state, meaning the state at the point when the action is being examined. This is right the state induced by the effects of the last applied preceding action.

5.2.3. Verification of STRIPS-like plans

Verification is realised via simulation of plan execution. Firstly, we construct an empty layer of facts. After that, we consecutively try to apply a single action (in the order given by the sequential plan) to the current world state represented by the last fact layer. If the action is applicable, the action is applied and a new world state is computed based on the effects of the action. If the action is not applicable, its effects are not encountered and the verification starts processing the next action in the plan. For instance, after the first "init" action is successfully applied, we

have constructed the initial world state as defined in the planning problem. An action is applicable to a given fact layer if and only if the layer contains all action's positive preconditions and simultaneously excludes all negative preconditions. If the action is applicable, a new fact layer is created. The new set of facts is computed based on the previous fact layer extended by the facts from action's positive effects and excluding action's negative effects.

Fact layer against which an action under examination is trying to be applied is remembered. If applicable, the fact layer which the action has created is stored as well. For STRIPS-like plans the first and the second fact layer are next to each other. In temporal plans, a difference between these two layers can vary a lot, as there can be arbitrary number of other actions' starts and ends between them (each start and end of durative action possibly creates a new layer). Such stored information will be used when finding how the world changes by applying the action (the actual set of facts prior and after the action).

Missing preconditions of the action (if any) and causal relations to previous actions in the plan are also computed for each action during its verification. In the visualization, an action is applicable if and only if its set of missing preconditions is empty. If a precondition of the action is not missing, we find the last fact layer from which the precondition fact is included (on the other hand, for negative precondition we find the last fact layer from which the precondition fact is excluded). The precondition then depends on the action assigned to that fact layer.

After each modification the plan is immediately re-verified.

5.2.4. Semantics of temporal plans

Based on the PDDL 2.1 specification introducing durative actions (Fox and Long, 2003), this section summarizes several aspects of temporal plans' semantics respectively to the extent of the thesis. A single durative action, besides "at start" conditions (equivalent to strips preconditions), defines "over all" conditions (invariant over a duration of the action) and "at end" conditions (needed to hold at the point at which the final effects of the action are asserted). Therefore, a durative action needs to be checked multiple times whether it is applicable or not. Invariant conditions are required to hold over an interval that is open at both ends (starting

and ending at the end points of the action). If one wants to specify that a fact `p` holds in the closed interval over the duration of a durative action, then three conditions are required: `(at start p)`, `(over all p)` and `(at end p)`.

Similarly, from an effect's annotation it is clear when the effect should apply, whether at the start of the interval or at the end of the interval. Effects can be applied only at these two end points of the durative action. This gives us aguideline how the temporal plans can be treated as point-based.

In order to handle concurrent actions we need to define the situations in which the effects of those actions are consistent with one another. The mutex[10] rule applies here. The rule makes sure there is no way of effects' conflict. An effect cannot be both asserted and negated by different actions at the same time. Considering the following example (Fox and Long, 2003):

```
(:action a
   :precondition (or p q)
   :effect (r))
(:action b
   :precondition (p)
   :effect (and (not p) (s)))
```

We might suppose that both actions can be executed simultaneously in a state in which both `p` and `q` hold. However, in such a case it would be necessary to check application of actions in all possible orderings. In order to avoid such complexity and define the semantics clearly, we will adhere to the rule of no moving targets. The rule means that there ano no two actions from which one action is using a value and the other one is changing the same value at exactly the same time . The no moving targets rule makes the cost of determining whether a given set of actions can be applied concurrently polynomial in the size of the set of actions and their conditions and effects.

Temporal plan with durative actions is valid only if both ends of every action are present in the plan. It is also supposed that precise simultaneity, in terms of ensuring that two independent actions are executed simultaneously, cannot be expected. Arbitrarily accurate time control cannot be expected, as well.

10 mutex stands for mutual exclusion, the basic concept from a well-known GraphPlan algorithm

5.2.5. Determining order of actions in temporal plans

In comparison to STRIPS-like plans, temporal plans do not determine order of their actions exactly. Each action in temporal plan can be assigned any start time and any duration. During the execution interval of one action, another actions may possibly start or end without any restrictions.

The way we transform the interval-based plan into a point-based plan involves creating actions to represent the end points of the actions' intervals. The only complication is that invariants must be checked during the corresponding interval. This is achieved by checking the invariants after each of the updating actions.

As already mentioned, verification algorithm for temporal plans transforms durative action into couples representing the end points of the corresponding interval. VisPlan's implementation is the following: each action internally clones / duplicates itself. Then, the original action is marked as "start" action, whereas the duplicated action is marked as "end" action. Having the same step repeated for all the actions, we end up with doubled set of actions (of two types). Such a set is used for the verification purposes solely. Next, the just created set of actions is sorted so that the verification manipulates the actions in the correct order. Sorting algorithm uses a comparator, which is a function deciding which of the two actions, given as parameters, should be earlier and which should be latter in the verification queue.

At the beginning, the comparator finds corresponding times for both given actions. Corresponding time for the "start" action is its start time. Corresponding time for the "end" action is its start time plus duration. The two retrieved times are compared (once the return statement is reached the comparison doesn't continue):

1. The two times differ:
   a) if the first time is less than the second time, the first action is returned as earlier
   b) if the first time is greater than the second time, the first action is returned as latter
2. The two times are equal:
   a) if one of the actions is a duplicate of the other action (if, theoretically, the

action has no duration), the “start” action is returned as earlier

b) if one of the actions is marked as the “start” action and the other one is marked as the “end”, the “end” action is returned as earlier; thus an “in progress” action (meaning the original durative action) is finished before a new one is processed

c) both actions of the same type (both are the “start” actions or both are the “end” actions
   i. an action assigned a shorter duration is returned as earlier one
   ii. an action returned as earlier is the one name of which (with parameters) is sorted earlier in an alphabetical order
   iii. if even the previous case had not arbitrated the result, the actions are completely the same, in fact, and therefore the comparison treats both actions being equal with each other

Due to the fact that the order of durative actions in the input plan file has not been standardized, the plan file order is not taken into consideration at all.

5.2.6. Verification of temporal plans

Verification of temporal plans is similar to STRIPS plans’ verification regarding the plan execution simulation and fact layers’ construction. In a general case, however, one durative action can create more than one fact layer, when both “at start” effects and “at end” effects are encountered.

Prior the verification, at the time the algorithm is creating operators matching to actions, each durative action is checked to have the duration complying with the duration specified for the operator in the planning-domain. In case the two durations vary, a user is prompted (in a new question message dialog) to accept or deny modification of action’s duration to the one specified in the planning domain. Once a user has denied action’s duration modification, he/she is never prompted again for the same action. When many actions from the plan under examination have similar conflict, the user is given a possibility to accept/deny modifications for all actions. Nevertheless, this doesn’t affect new actions eventually added to the plan.

The verification manipulates actions in the sorted order as described in the

previous section. As it has been already explained, every action is examined twice. A decision whether the action is being procesed for either the first time or for the second time is determined by the mark of the action (either “start” or “end” mark). At the first examination of an action, at its start time, “at start” conditions are checked against the current fact layer. If the action is applicable it is applied (taking its “at start” effects into consideration), resulting in creation of a new fact layer. In addition, the action is remembered to be “in progress” internal state. At the second examination of an action, at action’s end time (start plus duration time), the action finds out whether it has been applied at its start. If so, “at end” conditions are checked and, if satisfied, the action is applied (considering its “at end” effects). The action is removed from “in progress” actions at this phase.

As we have discussed in the temporal plans’ semantics section, when applying actions consecutively after each other we should take special care to ensure:

- effects applied by any two actions at the exactly same time must be mutex-free
- effects of an action cannot be used at the exactly same time as conditions for other actions

Considering these very conservative requirements, VisPlan performs the following checks:

- it checks whether any condition of the action under examination has already been asserted by the previously processed actions at the same time
- it checks whether any effect of the action has already been used as a condition for the previously processed action at the same time
- it checks whether effects of the action are mutex-free with respect to possibly asserted effects of the previously processed actions at the same time

If any of the checks answers positively we’ve encountered a mutex. The action is remembered to be the mutex-containing and cannot be applied. Afterwards, when visualizing, the action is treated in different way compared to the rest of the actions. Naturally, a question, why the action has been marked as mutex-containing but not the one which had induced the mutex, arises. VisPlan behaves

uniquely regarding the plans' manipulation since it continues with verification even if any invalid or non-applicable action is found. Such an action is just omitted. This is the key feature of VisPlan and marking action mutex-containing stands within this idea. In fact, VisPlan provides no guarantee for a plan being valid unless it contains exclusively applicable actions (and satisfies a goal conditions). Still, the user is given possibility to manually adjust the plan when situation like this comes up.

When processing an action during the verification either at its start time or its end time, besides checking its own conditions, the algorithm checks also "in progress" actions (those which have already started but haven't finished yet) to verify their "over all" conditions. Such verification is performed only when the inducing action is applied (either at its start time or end time).

In case an action's "at start" effects have been applied at action's start time and it has later been found that any of action's "over all" and "at end" conditions are not satisfied, the verification process is reverted back to the point when the affected action was applied at its start time, the action is omitted then and marked as non-applicable.

Similarly to STRIPS plans' verification, possibly missing conditions for an action are found while processing the action. However, for durative actions we store three different types of missing conditions: "at start", "over all" and "at end" missing conditions sets. Thus, in a future plan analysis, missing conditions are already available without need to be computed.

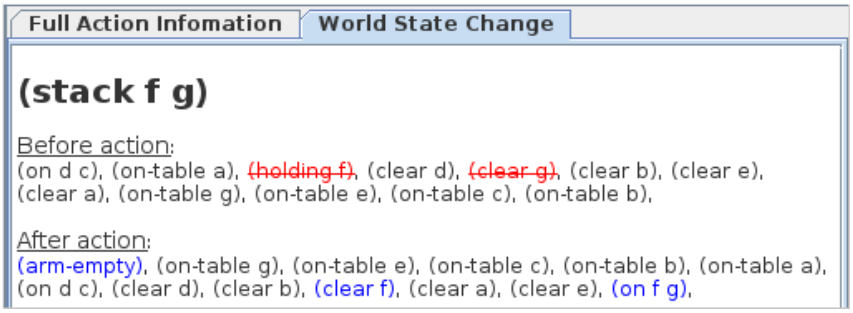


Figure 4: Example of information about world-state change.

5.3. Visualization

As shown in the right-upper frame of Figure 3, plan's actions are visualized as cells (boxes) of fixed size filled by the action name. Each action is coloured green or red (or any other colour chosen in the user preferences of the application) depending on whether the action is applicable or non-applicable. Besides applicable and non-applicable action states there exist several more action states; for each state VisPlan uses different visualization properties (color, available information, ...). Causal relations between the actions are visualized by edges. These edges are annotated by grounded facts that are "passed" between the actions. Only the causal relations for the currently highlighted action are displayed to remove a cluttered view. Display position of the edges is automatically adjusted every time an action is highlighted in order to assure that the edges do not overlap and their labels (describing the causal relations) are fully readable. The edge position adjustment is vertical (with fixed space size between edges), as well as horizontal (source and target points of edges on the same cell have regular space between themselves).

If the process of verification is still going on, actions whose state has not been decided yet are coloured gray (or any other colour chosen by the user). The state of an action can be one of the following:

- invalid (action doesn't match any definition in the planning domain file),
- un-decided (action is still being checked by the verification module),
- applicable (action is valid and can be used),
- non-applicable (action cannot be used due to non-satisfied preconditions),

- mutex-containing (in temporal plans only; any condition or effect is in mutex with the effect of another action in the plan at exactly given time).

Two special actions, “init” and “goal” are coloured differently to distinguish their special meaning. These are the only two actions which cannot be modified in any way.

For the highlighted action, the system displays complete information about the action including the satisfied and violated preconditions and actions giving these preconditions (the right-bottom frame of Figure 3), as well as world change caused by the action (Figure 4). World change illustrates which facts are true prior the action and which after the action. Naturally, world state information is not available for non-applicable, un-decided, invalid or mutex-containing actions. Facts that were subject of change (either added or deleted) are marked (by colour and/or by strike through their names).

On the left side of the window a list of actions is shown to provide a brief plan summary (the left frame of Figure 3). Actions in the list are sorted by their order/start time and are visually differentiated based on their states. The list gets updated every-time a modification is done to the plan. Selecting an action in the list results in adjusting the scrollbar view to comprise the visualized action in the graph and vice versa. If the user needs more space for graphical plan analysis he/she is free to hide the action summary list completely (as well as informative tab pane at the bottom of the application).

During a plan analysis, the ruler (Figure 3) helps to orientate within a time axis. Its default size of units is one inch (without dependence on user’s screen resolution). Size of units can be adjusted by the combo box (upper-right frame of Figure 3) or by dragging any tick of the ruler.

While dragging an action (to change its position), actions providing preconditions and actions using effects of the dragged action are dynamically highlighted, so that the user knows where he/she can drop the action. When actions are swapped it usually changes causal relations between the actions significantly. Due to this fact, highlighting preconditions and effects partially wouldn’t provide enough information. Therefore the plan is re-verified when an

action changes its order while dragging. Having such information the program chooses the correct actions to highlight. Colour for highlighting is the same as colour for preconditions/effects edges. If actual colour of an action is the same as the colour for edges when highlighting, another (but similar) colour is used then.

Each user has an opportunity to set his/her own user preferences regarding the visual appearance and behavior of software according to the personal needs. The user preferences are saved in the home directory of the user and include various (mostly graphical) settings, for instance:

- colors for actions (each state has its own color), edges (both preconditions and effects) and ruler,
- font size (for different GUI components),
- automatic loading of last successfully loaded files (domain, problem, plan) at start-up,
- default action width in STRIPS-like plans.

5.3.1. Visualization of STRIPS plans

As the STRIPS plans are sequential, cells representing the actions are displayed in a row. When changing the order of an action by drag & drop, the new order is computed after each movement by checking the horizontal position of the cell being dragged and ruler's units. In the case the new position is different from the current one, a cell placed at that moment on the "new order" position is immediately repositioned to the "current order" position, and thus these two actions swap their position. When the action is finally dropped, it is just placed in the row.

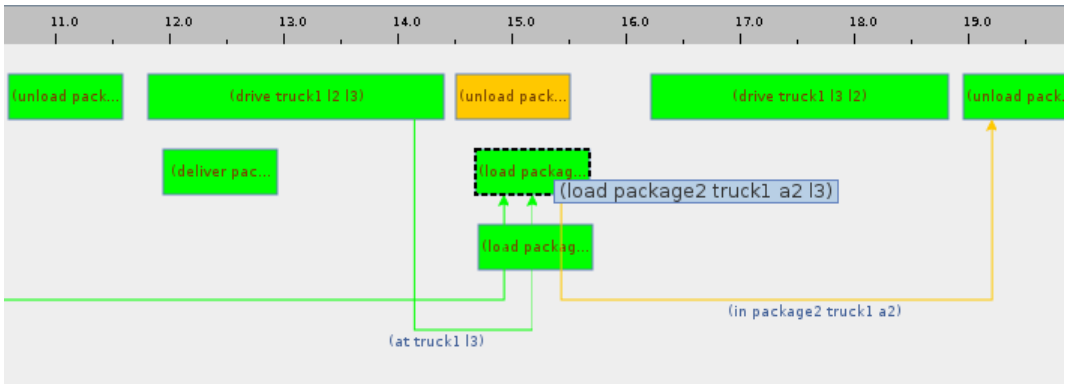


Figure 5: Example of visualisation of temporal plans.

5.3.2. Visualization of temporal plans

Ruler units in temporal plans reflect durations of actions. However, as individual durations of actions within a plan can vary a lot, the median duration has been chosen to be the initial ruler unit. Auxiliary ticks are also present on the ruler. All actions (meaning cells) are also guaranteed to have a minimum horizontal size (in order to be visible even if real duration is too small).

Horizontal position of an action is fully determined by its start time and duration. Although actions in temporal plans can overlap with each other, cells representing the actions are positioned in order to be fully visible. This is performed by placing the cells in rows. All cells in the same row have the same vertical position. Cells position adjustment is iterative and cells are positioned into the first row (from top) where the cell would not overlap with other cells (Figure 5).

When an action is being dragged, in contrast to STRIPS plans, the start time of the action is determined by the horizontal position only (multiplied by the current ruler units). In such a situation re-verification of the plan is done only when the action has changed its position significantly, meaning the relative order of the dragged action margins (start/end) changed with respect to other actions.

5.4. Plan Modifications

In addition to visualization of plans the software supports interactive modification of the plan. The following operations with plans are supported:

- inserting new actions (selection of actions and their parameters is

automatically restricted to the current planning domain and the problem and offered in the corresponding number of pre-filled combo boxes),

- removing actions,
- modifying actions,
- changing the order of actions in STRIPS plans and start time of action in temporal plans by drag & drop technique.

Modifications are revertible and are under control by undo manager. Undo manager waits for performing an undoable (revertible) modification, which is any of the above. When an undoable change is fired, undo manager clones and saves both the current plan and verificator state (this includes the constructed layers of facts, the causal relations among actions, actions' indexes to layers before and after application, missing conditions). On the one hand, this approach is more memory consuming, due to the fact that undo manager saves as many plans and verificator states as is the limit of possible "undo"s. On the other hand, the approach is time-saving. Re-verification is not needed to be performed after each "undo"/"redo". All the necessary steps include just retrieving previous/next plan and verificator state plus redrawing the graph based on the retrieved plan. In comparison with a memory-saving approach, which would save only modifications' description and would perform opposing action during "undo"/"redo", the chosen approach is easier and more "defect-resistant". That is because it coherently maintains entire plans and states.

Besides the already mentioned plan and verificator state, undo manager saves two more items for user-friendliness and informative purposes. These include id of an action causing an undoable change (in order to select this action and to adjust view to comprise it) and a string describing the change (in order to print informative message onto status panel at the bottom of the application).

Modified plans can be saved in the text format to either the same (initially loaded) file or to a new file (save as).

6. Working with VisPlan

In this chapter we will provide a brief introduction to VisPlan from the end-user perspective. While describing the functionality of VisPlan from an algorithmical point of view in the previous chapter, many usability issues have already been mentioned. In the next paragraphs these will be skipped or discussed just very briefly, so that we can focus on a new information.

When we launch VisPlan - a desktop application written in Java programming language for the first time, the GUI (Graphical User Interface) components (main visualization and informative windows) are blank, as we haven't loaded any plan, yet. In order to verify and visualize a plan, the application needs 3 separate files: domain, problem and plan file. We can specify such a trio by either selecting "File -> Load" from the application menu, or clicking on the "load button" from toolbar, or pressing "Ctrl + L" key sequence. In either case, a "Load files" window pops-up (Figure 6). File choosers are available here as well, so that we have an option to specify files by browsing with a file manager. File choosers implement file filters which, in case of domain and problem files, show only files with "pddl" suffix and, in case of plan file, show only files with "txt" suffix.

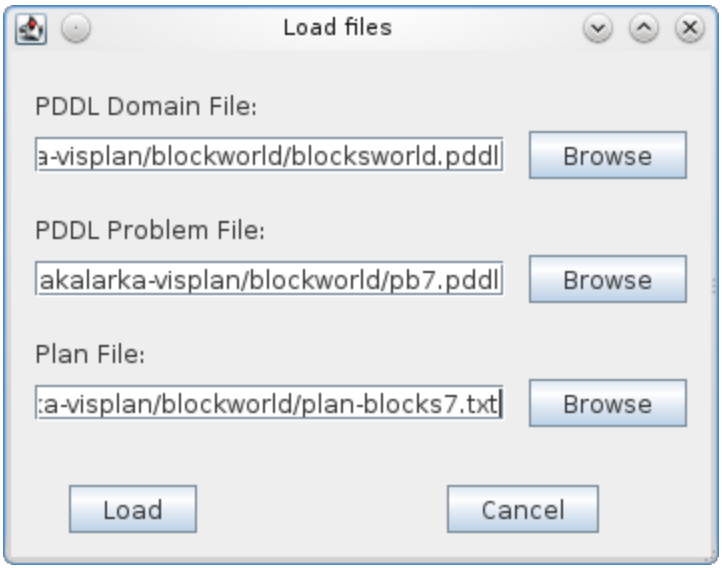


Figure 6: Load Files window in VisPlan application.

After selection, domain and problem PDDL files are parsed and if they contain any syntax error, a list of the error(s) is printed, as shown in Figure 7, and the

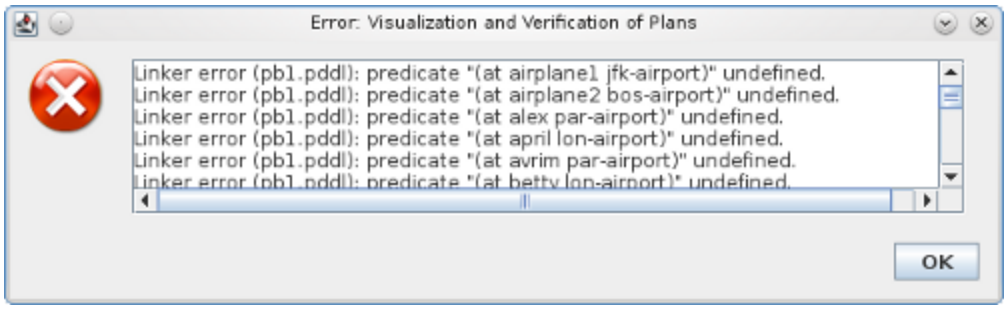


Figure 7: Error/warning messages window in VisPlan

program does not continue with verification.

After domain, problem and plan are correctly parsed, the plan is shown to the user, so that he/she can immediately obtain a rough idea of the plan organization in a time axis. The user is free to adjust sizes of two informative windows (one on the left side and the other at the bottom), or he/she can hide the mentioned windows completely using small split pane arrows (in favor to main visualization window). A concrete example of the GUI with plan already displayed might look the screenshot in Figure 3 (page 17).

Simultaneously, a separate program thread is started which performs plan verification. This verification process is iterative. See section 5.2. Verification (page 18) for more details how the verification is performed.

As an iterative process of verification finishes, the plan visualization changes as well. The plan's actions, visualized as graph vertices, are coloured with respect to their action state (changing default colours will be described in the next paragraphs). In addition to that, if cursor hovers above an action, action's tooltip (showing action's name plus parameters) is displayed. If the cursor hovers above an edge, representing a causal relation, the tooltip displays edge's name and the both actions which it interconnects. Such a name represents predicate (or set of predicates) that is, at the same time, an effect of the source action and a precondition for the destination action. Figure 8 demonstrates this. Green edges represent preconditions, blue edges represent effects of the action in question (again, these colours can be adjusted in the user preferences).

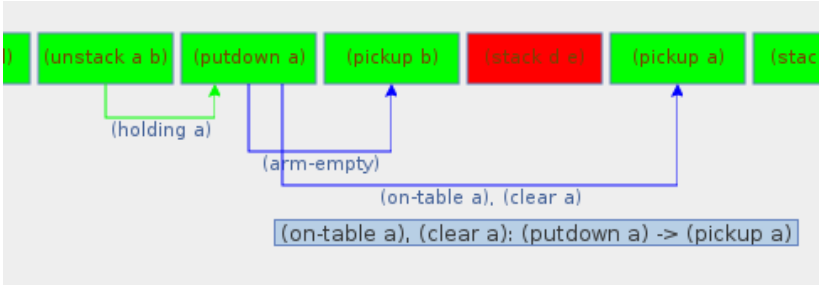


Figure 8: Causal relations' visualization in VisPlan application

In order to keep the graph transparent, the edges are not displayed by default (as this would mean, in most cases, displaying too many edges simultaneously). The edge visibility policy is therefore restricted only to one action at the same time, both incoming (precondition) and outgoing (effect) edges, as shown in Figure 8.

The left frame of Figure 3 (page 17) shows how actions are consecutively ordered in the plan. It provides a brief overview of the actions. This overview might be handy in cases, when a user wants to examine some sequence of the plan, but corresponding actions in the graph do not all fit into the size-restricted window. The overview graphically distinguishes applicable and not applicable and other types of action states (by font colour).

In addition to edges becoming visible when an action is hovered (meaning the cursor appears above the action), full action information report is generated out in a separate window (The bottom frame of Figure 3 (page 17)). This report includes:

- action name (plus parameters)
- either order of the action (for strips plans) or alternatively start time + duration (for temporal plans)
- grounded preconditions (coloured green/red if satisfied/unsatisfied, respectively)
- for each precondition an action on which it depends
- which preconditions are not satisfied (if any)
- grounded effects
- for each effect actions depending on this effect

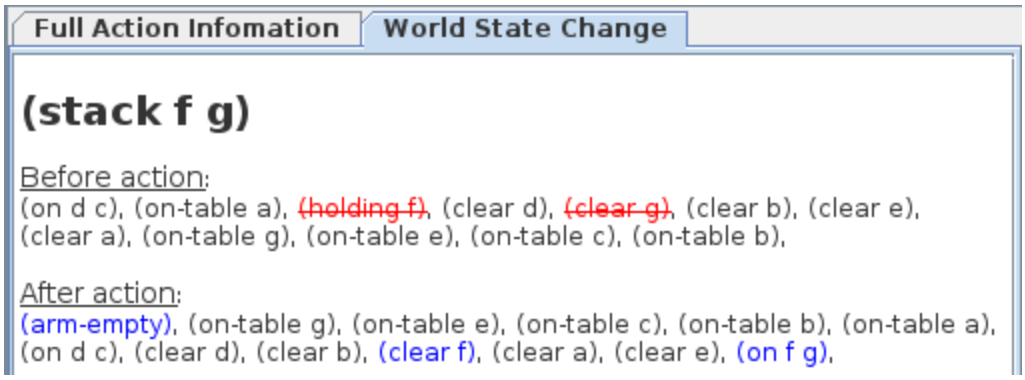


Figure 9: World state change in VisPlan application

Besides full action information, a "world state change" information for the "hovered" action is displayed in another separate information tab (Figure 9). The "world state change" view shows predicates applicable before and after the action. Moreover, predicates removed and added are visually separated from the (unchanged) rest of the predicates. The user can freely switch between the two mentioned information views, as they are located in a common tab pane (at the bottom part of the graphical interface).

It is possible for user to adjust the width of the vertices (actions), so that it better reflects his/her needs. This can be done using a size combo-box (the top right frame of Figure 3 (page 17), or by dragging any ruler's tick.

The following plan changes are available:

- Removal of selected action (by pressing "Delete" button or clicking "Remove action" toolbar button or menu item). All the toolbar buttons provide a tooltip with a brief button's functionality description. Furthermore, toolbar buttons which cannot be used at that moment are disabled automatically.
- New actions can be added (by pressing "Ctrl + I" button or clicking "Insert new action" toolbar button or menu item). In an action addition dialog (Figure 10), these items must be specified: operator name and all its parameters (provided by combo boxes), order (for STRIPS-like plans) or alternatively a start time and duration (for temporal plans). Correctness of the items is automatically validated.

- Existing actions can be modified (by double-clicking on the action or by clicking “Modify action” toolbar button or menu item). An action addition dialog pops-up (Figure 10), however, with the predefined values for in this case.
- Changing order (or start time for durative actions) by drag&drop technique.
- Eventually, an “undo”/“redo” feature is available as well, which is very useful when the user has performed any unwished plan modification. “Undo”/“redo” can be invoked by clicking on respective toolbar buttons, menu items or by pressing the “Ctrl + U” or “Ctrl + R” keyboard buttons.

If any of the the above plan modification actions is performed, the plan is revalidated from the removed/added/modified action onwards.

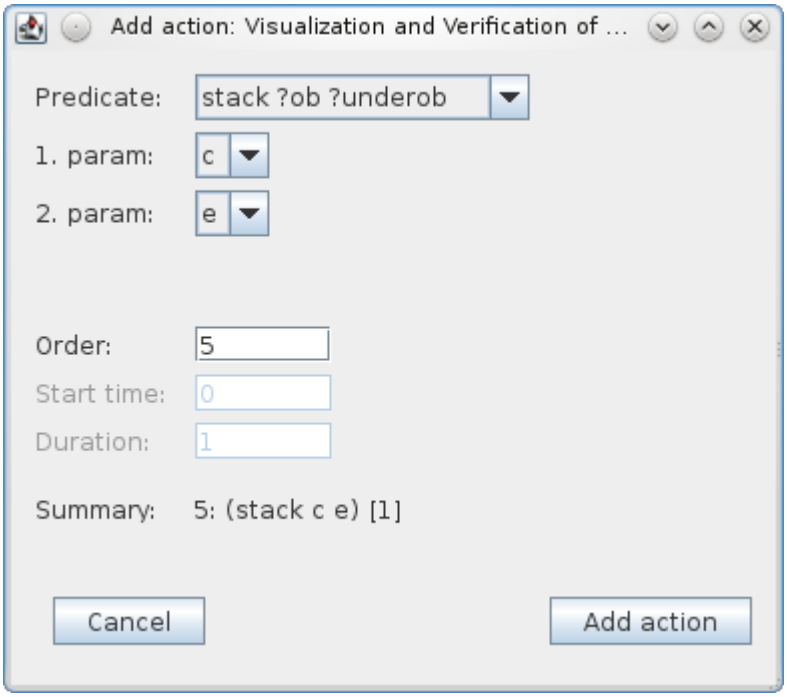


Figure 10: Adding a new action in VisPlan application

If the plan is modified, the user is given an option to export the plan into text file. Moreover, if the plan had been changed but not saved before the user exits the application, an exit confirmation dialog is displayed where the user is prompted to either save the modified plan or exit without saving.

Each user has an opportunity to adjust his/her own user preferences according to

his/her personal needs. User preferences include various settings, for instance:

- actions' and edges' colours (single colour for different kinds of actions/edges),
- resizability of vertices representing actions,
- automatic loading of last successfully loaded trio (domain, problem, plan) at startup,
- and much more.

User preferences are stored in the 'preferences' file under application's '"visplan" directory. ".visplan" directory is automatically created in the user's home directory just after the settings are saved for the first time. Application's preferences dialog can be invoked by either selecting "Edit -> Preferences" from application's menu, or clicking on the "Preferences" toolbar button, or pressing "Ctrl + P" key sequence. Then, preferences dialog pops-up (Figure 11).

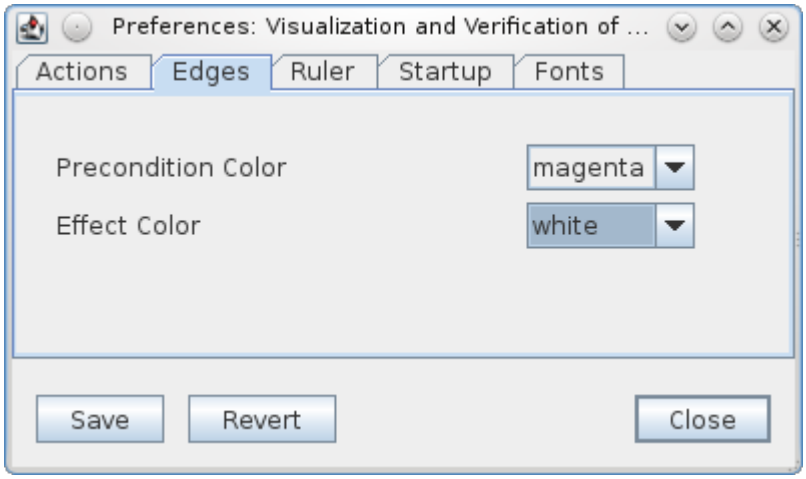


Figure 11: User preferences settings in Visplan Application

Useful information about the application can be easily found by invoking an "About" dialog (select "Help -> About" from menu). Besides short application description, version and author, user is provided a link to the VisPlan's homepage, where news, up-to-date version of the software and other interesting information are regularly stated.

7. VisPlan Implementation

In this chapter an overview of VisPlan implementation will be provided. We will focus on the decomposition of the program to the smaller closed program units, called modules. We will provide a brief description of each module and describe how the modules interact with each other, so that the reader obtains a general overview of how the program works. On the other hand, we will not go into much details about implementation of single classes. The reader may consider to look to the JavaDoc attached on the CD-ROM, where he/she can find more information about the classes and their methods and data fields. Similarly, the alogithms used in the program will not be covered here as they have been mostly discussed previously in the verification and visualization sections.

7.1. External libraries used in the program

7.1.1. JGraph Java library[11]

JGraph is an open source graph visualization Java library. It is based on the mathematical graph theory. JGprah is fully compatible with the Swing. Therefore it can be used within the Swing[12] GUI applications quite easily. For example, the main component for displaying graph in JGraph library is a direct subclass of the Swing class javax.swing.JScrollPane[13], meaning it has all the inherited methods of the JScrollPane component available.

JGraph provides a wide range of graph drawing functionality, such as automatical layouting and performing analysis of graphs. The way how a graph is displayed can be adjusted by JGraph's API (Application Programming Interface) and graphs can also caryy a certain logic, as other objects can be associated with the graph components.

11 The library has been downloaded and more information retrieved from: (http://www.jgraph.com/jgraph5.html, 2010)
12 Swing is the primary Java GUI toolkit
13 It is a component which is both vertically and horizontally scrollable

7.1.2. PDDL4J Java library[14]

PDDL4J is an open source Java library. The goal of the PDDL4J is to provide a low-level functionality for Java applications manipulating files written in the PDDL language.

The library contains a parser for PDDL 3.0 version. The parser can be configured to accept only specified requirements of the PDDL language. After a file is successfully parsed, PDDL4J classes (objects) then carry individual elements of the parsed file. The library also implements an error manager used by the parser to hold possibly encountered errors and warnings.

7.2. Single handling of different plan types

During the program development, at the time when we were adding a support for temporal planning, we have found out that there were too many cases we had to deal with a temporal plan in a different way compared to a STRIPS-like plan. This lead us to create a common parent class for the both types of plans and a subclass for each of the types. The subclasses handle the same situation differently, however, now we have a common way to call their methods - via the abstract parent class. This principle is used everywhere we need to treat a situation in different way based on the plan type. The following classes/modules use the principle[15]: Plan, Op (operator), Verificator, State, Visualizer, Ruler.

7.3. Program modularity

7.3.1. GUIView

GUIView is the main program module. Because it defines the main GUI window with all the graphical components, it defines many listeners and handlers for all the user inputs (such as clicking on an action in the plan, pressing a keyboard button, selecting an item from the file menu, ...). Each listener than reacts on the inputs, in most cases it calls other module's methods.

Besides all the graphical components (including other windows like preferences

14 The library has been downloaded and more information retrieved from: (http://sourceforge.net/projects/pdd4j/, 2010)

15 In the next sections, if we mention a Visualizer class for example, it means we have both StripsVisualizer and TemporalVisualizer classes on mind, but we don't distinguish between them as they have the same API in majority of cases.

or add action dialog) the module contains all other main modules and synchronizes them. These are Plan, Graph, Ruler, State, Verificator, Visualizer and UndoManager.

7.3.2. Plan

The Plan class represents a plan, what is a set of plan actions. Each plan action consists of instantiated operator, start time and duration, all encapsulated into one object of PlanAction class.

7.3.3. State

This module represents states of the world during the plan execution. Moreover, for each plan action it saves preconditions and effects, missing preconditions and links to previous actions on which it depends (causal relations). This can be achieved by various representations. Therefore, State class is abstract and is extended by BitSetState class which implements bitset representation. If the concrete implementation ever needs to be modified or replaced (by more effective representation), a new class would extend State class and thus not influence other modules of the program.

State class is responsible for:

- creating world state layers during the plan execution simulation
- returning specific world state (before or after application of an action)
- returning preconditions and effects of an action
- returning missing preconditions of an action
- returning causal relations of an action

7.3.4. Verificator

It is a module which executes the plan verification. Verification process is executed as an independent thread and runs simultaneously with the main (GUI) thread. This is useful in case the verification of the plan takes a long time. In such case the user can still work with the plan - without waiting for the verification process to finish.

Verificator stands between GUIView and State. GUIView uses Verificator for running the verification, answering questions about applicability and updating

actions' information (like the causal relations of actions, preconditions and effects, missing preconditions, ...). Verificator itself does not contain such data, it asks the State for the data. Then it interprets them to GUIView.

After the verification process is finished, Verificator colours actions (vertices in the graph) with corresponding colour. Moreover, it also creates edges in the graph representing causal relations.

7.3.5. Visualizer

Visualizer's main function is to show the plan. It iterates over plan's actions (meaning vertices) and for each of them it computes where it should be placed (in plane). Visualizer distinguishes between strips/temporal plans. For temporal plans the visualization presents a Gantt chart.

Besides of that it handles plan modifications as well. Especially when the user changes position of actions by drag&drop, it handles the movement of actions.

8. Future Development

The program is under continuous development and all the relevant information plus the up-to-date version of the software can be downloaded from:

`http://glinsky.org/visplan`

In the future VisPlan is intended to support additional features such as:

- wider support of PDDL requirements

  - disjunctive-preconditions: allows or in goal and preconditions

  - existential-preconditions: allows exists in goal and preconditions

  - universal-preconditions - allows forall in goal and preconditions

  - quantified-preconditions: equivalent to existential-preconditions + universal-preconditions

  - conditional-effects: allows when clause in action's effects

  - fluents - allows function definitions and use of effects using assignment operators and numeric preconditions

- support of plans specified in PDDL+

- own planning module

  - this feature would make possible to find a solution (the plan) for a given planning problem directly from the program

  - user would not need to specify already found plan for a problem

  - this would be an important step towards making VisPlan a complex planning system, not only a plan analyser

- support for finding possible plan modifications in order to solve flaws in the plan

  - program would not just recognize flaws but would provide possible plan modifications with an intent to satisfy a goal

- graphs visualizing a timeline of predicates and numerical variables during plan execution

  - user would be able to track chosen predicate or variable (such as amount of gas in a tank of a car), thus gaining even better overview of a plan

Conclusion

The thesis, particularly VisPlan as the practical implementation of the thesis, provide an environment for plan analysis. It encapsulates several different plan-manipulating tasks into one single program. This includes parsing the PDDL domain and problem, parsing the plan file, verification of the plan and, finally, visualizing the plan. The plan visualization is based on showing the causal relations between actions.

VisPlan, however, is not dedicated only to a static plan analysis. It provides tools which interactively modify the plan under examination, so the user has a possibility to follow the plan execution changes in the real time based on his/her modifications. The “undo” feature may be very useful in these cases.

The program pays special attention to automatize everything what is possible and reasonable. From this point of view it doesn’t require any unnecessary actions or input from the users. Some examples of such (artificially intelligent) behavior may include automatic decision of the plan type (and thus automatic decision of the proper ways of verification/visualization of the plan) or automatically prefilled combo-boxes representing arguments for an action in the action addition dialog.

In contrast to other already existing plan analysis tools, such as itSimple or GIPO, VisPlan is natively able to handle also plans which are not valid. This is one of the key features of VisPlan and the main idea is to skip actions from the verification process once found they are non-applicable/invalid. This allows us to examine also the rest of the plan after the first non-applicable action. It can be said that this approach is kind of a novelty. The results of the thesis have also been presented on the KEPS workshop (Knowledge Engineering for Planning and Scheduling) organized within ICAPS 2011 conference (International Conference on Automated Planning and Scheduling) (Glinský and Barták, 2011).

VisPlan is still under continuous development. Many useful features are intended to be added to VisPlan, such as displaying a timeline of predicates and numerical variables. Considering our future goals, summarized in the previous chapter, we are still at the beginning.

Bibliography

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

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

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

Glinský, R.; Barták, R. (2011). VisPlan - Interactive Visualisation and Verification of Plans. In Proceedings of the ICAPS 2011 Workshop on Knowledge Engineering for Planning and Scheduling (KEPS 2011), pp. 134-138. Available on-line at http://icaps11.icaps-conference.org/proceedings/keps/keps2011-proceedings.pdf.

Russel, S.; Norvig, P. (2003). Artificial Intelligence: A Modern Approach (second edition), Prentice Hall, New Yersey.

Simpson, R.M.; Kitchin D.E.; McCluskey, T.L. (2007). Planning Domain Definition using GIPO. The Knowledge Engineering Review 22(2): 117-134.

Vaquero, T. S.; Silva, J. R.; Beck, J.C. (2010). Analyzing Plans and Planners in itSIMPLE3.1. In: Proceeding of the ICAPS 2010 Knowledge Engineering for Planning and Scheduling Workshop. Toronto. Canada, pp. 45-52.

http://dlab.poli.usp.br/twiki/bin/view/ItSIMPLE/OverView

http://en.wikipedia.org/wiki/Tower_of_Hanoi

http://kti.mff.cuni.cz/~bartak/planovani/index.html

http://scom.hud.ac.uk/planform/gipo

http://sourceforge.net/projects/pdd4j/

http://www.jgraph.com/jgraph5.html

http://www.inf.uos.de/schmid/LB-Kog/hanoi.lisp

http://zeus.ing.unibs.it/ipc-5/generators/Domains/trucks-Time.pddl

List of Figures

List of Tables

List of Abbreviations

AI - Artificial Intelligence

API - Application Programming Interface

DARPA - Defense Advanced Research Project Agency

DART - Dynamic Analysis and Replanning Tool

itSimple - Integrated Tools Software Interface for Modeling PLanning Environments

GIPO - Graphical Interface for Planning with Objects

GNU - GNU is Not Unix

GUI - Graphical User Interface

ICAPS - International Conference on Automated Planning and Scheduling

IPC - International planning competition

KEPS - Knowledge Engineering for Planning and Scheduling

PDDL - Planning Domain Definition Language

STRIPS - STanford Research Institute Problem Solver

UML - Unified Modeling Language

VisPlan - Interactive Visualization and Verification of Plans (program)

XML - eXtended Markup Language

Attachments

1. CD-ROM containing:
    - VisPlan program (compiled and bundled in jar file, including libraries)
    - Java source code of VisPlan
    - JavaDoc for VisPlan
    - VisPlan’s sample input files

Appendix

A. Concrete example of (STRIPS-like) domain file

Description:

"The Tower of Hanoi or Towers of Hanoi , also called the Tower of Brahma or Towers of Brahma, is a mathematical game or puzzle. It consists of three rods, and a number of disks of different sizes which can slide onto any rod. The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top, thus making a conical shape.

The objective of the puzzle is to move the entire stack to another rod, obeying the following rules:

- Only one disk may be moved at a time.
- Each move consists of taking the upper disk from one of the rods and sliding it onto another rod, on top of the other disks that may already be present on that rod.
- No disk may be placed on top of a smaller disk."

(http://en.wikipedia.org/wiki/Tower_of_Hanoi, 2011)

```
(define (domain hanoi)[16]
  (:requirements :strips)
  (:predicates
        (clear ?x)
        (on ?x ?y)
        (smaller ?x ?y))
  (:action move
        :parameters (?disc ?from ?to)
        :precondition (and (smaller ?to ?disc)
                           (on ?disc ?from)
```

16 The hanoi domain and problem example has been taken and slightly modified from: (http://www.inf.uos.de/schmid/LB-Kog/hanoi.lisp, 2011)

```
                    (clear ?disc)
                    (clear ?to))
        :effect  (and (clear ?from)
                    (on ?disc ?to)
                    (not (on ?disc ?from))
                    (not (clear ?to)))
))
```

B. Concrete example of (STRIPS-like) problem file

```
(define (problem hanoi-pb1)
   (:domain hanoi)
   (:requirements :strips)
   (:objects rod1 rod2 rod3 d1 d2 d3)
   (:init
         (smaller rod1 d1)
         (smaller rod1 d2)
         (smaller rod1 d3)
         (smaller rod2 d1)
         (smaller rod2 d2)
         (smaller rod2 d3)
         (smaller rod3 d1)
         (smaller rod3 d2)
         (smaller rod3 d3)
         (smaller d2 d1)
         (smaller d3 d1)
         (smaller d3 d2)
         (clear rod2)
         (clear rod3)
         (clear d1)
         (on d3 rod1)
         (on d2 d3)
         (on d1 d2))
   (:goal  (and (on d3 rod3)
         (on d2 d3)
         (on d1 d2)))
)
```

C. Concrete example of (temporal) domain file

Description:

Essentially, this is a logistics domain about moving packages between locations by trucks under certain constraints. The loading space of each truck is organized by areas.

```
(define (domain trucks)[17]
(:requirements :typing :adl :durative-actions :fluents)
(:types truckarea location locatable - object
        truck package - locatable)
(:predicates (at ?x - locatable ?l - location)
        (in ?p - package ?t - truck ?a - truckarea)
        (connected ?x ?y - location)
        (free ?a - truckarea ?t - truck)
        (delivered ?p - package ?l - location))
(:functions (drive-time ?from ?to - location))
(:durative-action load
        :parameters (?p - package ?t - truck ?a1 - truckarea ?l - location)
        :duration (= ?duration 1)
        :condition (and (at start (at ?p ?l))
                (at start (free ?a1 ?t))
                (over all (at ?t ?l)))
        :effect (and (at start (not (at ?p ?l)))
                (at start (not (free ?a1 ?t)))
                (at end (in ?p ?t ?a1))))
(:durative-action unload
        :parameters (?p - package ?t - truck ?a1 - truckarea ?l - location)
        :duration (= ?duration 1)
```

17 The trucks domain and problem example has been taken and slightly modified from: (http://zeus.ing.unibs.it/ipc-5/generators/Domains/trucks-Time.pddl, 2011)

```
        :condition (and (at start (in ?p ?t ?a1))
                (over all (at ?t ?l)))
        :effect (and (at start (not (in ?p ?t ?a1)))
                (at end (free ?a1 ?t))
                (at end (at ?p ?l))))
(:durative-action drive
        :parameters (?t - truck ?from ?to - location)
        :duration (= ?duration (drive-time ?from ?to))
        :condition (and (at start (at ?t ?from))
                (over all (connected ?from ?to)))
        :effect (and (at start (not (at ?t ?from)))
                (at end (at ?t ?to))))
(:durative-action deliver
        :parameters (?p - package ?l - location)
        :duration (= ?duration 1)
        :condition (and (at start (at ?p ?l))
                (over all (at ?p ?l)))
        :effect (and (at end (not (at ?p ?l)))
                (at end (delivered ?p ?l))))
)
```

D. Concrete example of (temporal) problem file

```
(define (problem truck-1)
(:domain trucks)
(:objects
        truck1 - truck
        package1 - package
        package2 - package
        package3 - package
        l1 - location
        l2 - location
        l3 - location
        a1 - truckarea
        a2 - truckarea)
(:init
        (at truck1 l2)
        (free a1 truck1)
        (free a2 truck1)
        (at package1 l3)
        (at package2 l3)
        (at package3 l1)
        (connected l1 l2)
        (connected l1 l3)
        (connected l2 l1)
        (connected l2 l3)
        (connected l3 l1)
        (connected l3 l2)
        (= (drive-time l1 l2) 4.3)
        (= (drive-time l1 l3) 7.1)
        (= (drive-time l2 l1) 4.3)
```

```
        (= (drive-time l2 l3) 3.8)
        (= (drive-time l3 l1) 7.1)
        (= (drive-time l3 l2) 3.8))
(:goal (and
        (delivered package1 l1)
        (delivered package2 l2)
        (delivered package3 l2)))
(:metric minimize (total-time))
)
```