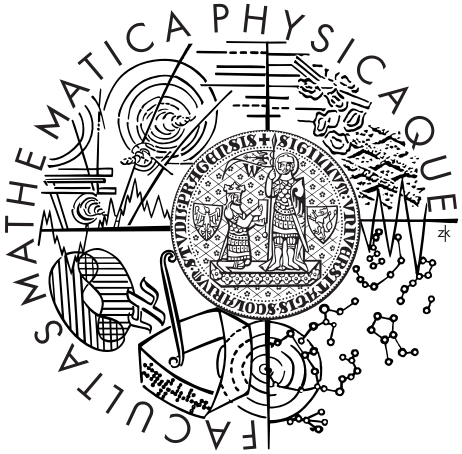
Charles Univerzity in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Bc. Jindřich Vodrážka

Modelling Planning Problems

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis: doc. RNDr. Roman Barták Ph.D.

Study programme: Computer science

Specialization: Theoretical computer science

Prague 2011

Acknowledgements

In this place I want to thank my supervisor doc. RNDr. Roman Barták Ph.D. for his patience, support and critical comments that directed my effort while working on this thesis.

I would also like to thank my parents for their continuous support during my studies.

I declare that I carried out this master 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 Prague on December 9, 2011 signature

Název práce: Modelování Plánovacích Problémů

Autor: Bc. Jindřich Vodrážka

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí diplomové práce: doc. RNDr. Roman Barták Ph.D., KTIML

Abstrakt:
Téma předložené diplomové práce spadá do oblasti znalostního inženýrství v automatickém plánování. V poslední době je pro reprezentaci plánovacích problémů často využíván koncept stavových proměnných. V rámci této práce je tento koncept využit v novém formalismu pro modelování plánovacích domén. Na základě tohoto formalismu je postaven prototyp nástroje určeného pro modelování plánovacích domén a problémů. Možnosti tohoto nástroje jsou pak demonstrovány na ukázkovém příkladu klasické plánovací domény. Nástroj poskytuje možnost exportu do standartního jazyka pro modelování plánovacích domén. Tím umožňuje propojení s existujícími plánovacími systémy.

Klíčová slova: klasické plánování, stavové proměnné, znalostní inženýrství

Title: Modelling Planning Problems

Author: Bc. Jindřich Vodrážka

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: doc. RNDr. Roman Barták Ph.D., KTIML

Abstract:
This thesis deals with the knowledge engineering for Automated Planning. The concept of state variables has been recently used with benefits for representation of planning problems. In this thesis the same concept is used in a novel formalism for planning domain and problem modeling. A proof-of-concept knowledge modeling tool is developed based on the new formalism. This tool is then used for modeling of example classical planning domain to show its capabilities. The export to standard domain modeling language is also implemented in the tool in order to provide connection to existing planning systems.

Keywords: classical planning, state variables, knowledge engineering

Contents

Introduction

Process of knowledge integration into computer systems is gaining significant attention in the recent years. One particular area where this claim certainly holds is the Automated Planning.

While planners are being developed to solve complex problems there is also much effort put into the development of knowledge engineering techniques for modelling of planning problems. The knowledge engineering deals with the task of formal description of models used for planning and schedulling, their validation and maintenance.

There is a wide range of problems which are being modelled. Beginning with comparatively easy benchmark problems and scaling up to applications such as crude oil distribution in petroleum plant or computation of genome edit distances. Throughout the text we will often refer to a specific area as planning domain.

By using the word "easy" when talking about the benchmark problems we mean "easy to explain" or "easy to describe". Modifications of these problems are still challenging enough to be included in the International Planning Competition.

We can observe that the knowledge engineering is an universal interdisciplinary area. Creating a model of a complex problem is a task for an expert who understands the problem thoroughly. It is only through the model of the problem we can benefit from the planning technology. Therefore we need to provide either high level language or a modelling tool that can support the description of the problem models. There is a gap to be filled with the KE tools which are easy to use for non-experts in planning. These tools (or languages) can be specialized in a specific area of interest (e.g. bussines modelling) or provide a general-purpose foundation for modelling various problems from different areas. The general-purpose approach in particular poses a great challenge for KE.

In this thesis we are going to review existing general-purpose approaches for planning problem modelling and we will also look into various ways how the input data are transformed in the early stages of planning process.

We will examine in detail the process of knowledge modelling employed in leading contemporary KE tools for general-purpose planning. Based on the previous background research in the field of KE for Automated Planning we will propose a new formalism that will enable implementation of knowledge modelling tools based on the concept of state variables. The concept will be explained along with other concepts for planing problem representation in the first chapter.

The state variables are already succesfully used in many real planning applications. These applications are often specialized to the particular areas (e.g. space research). On the other hand the models based on propositional representations still play important role in the research of general-purpose planning.

In this thesis we will introduce a tool that uses state variables as building stones of the planning domain. With developement of knowledge modeling tool for general purpose planning based on the state variables we hope to bring some new ideas and possibly provide a platform that can be further extended to model "real world" problems. We will show that the state variable based approach is suitable for creation of efficient models and that it also provides a new perspective for the planning domain design process.

The modelling tool presented in this thesis is developed in order to prove that the formalism mentioned before can be actually implemented in a real software application. The program GUI can be used to design planning domain and problems. The program also supports export of the modelled planning domains and problems to the language *PDDL* in order to provide connection to planners.

This work is structured into six chapters. In the first chapter we will give an overview of the automated planning and we will look into classical planning in more detail. The second chapter will describe various approaches to representation of planning problems. We will describe the languages used for knowledge modelling and also some approaches to direct knowledge representation used by planning systems. In the third chapter we will analyze the tools for knowledge modelling currently in existence and also the main difficulities in the process of formal model design. A new approach for knowledge modelling will be presented in the fourth chapter together with suitable formalism. In the fifth chapter we will present a proof-of-concept tool for knowledge modelling based on the formalism presented in the fourth chapter. In the last chapter we will offer comparsion of our formalism to another efficient representation based on the concept of state variables. In the end we will also describe a new point of view on state variable based knowledge modelling.

1. Automated planning

According to [20] planning could be defined as follows:

Definition. Planning *is an abstract deliberation process that choses and organizes actions by anticipating their expected outcomes. At the same time this deliberation aims at achieving as best as possible some prestated objectives.*

Automated planning is an area in AI which is examining this process computationally. There are many forms of planning by the area of application. We can refer to particular applications in space research [22], education [7] or industry [31] or describe more general areas like *path and motion planning*, *perception planning*, *navigation planning*, *manipulation planning* or *communication planning*. All these applications are working with some kind of a dynamic system which can be generalized into a common model. In the first section of this chapter we will introduce this model in order to use it later for reference. In the second section we will describe classical planning to provide background for this work. The last section is discussing other branches of automated planning.

1.1 State transition system

A general model used to describe dynamic systems for automated planning is based on a *state transition system*:

Definition. *A* state transition system *is a 4-tuple* $\Sigma = (S, A, E, \gamma)$ *where:*

- $S = \{s_1, s_2, \ldots\}$ - *a finite or recursively enumerable set of states*
- $A = \{a_1, a_2, \ldots\}$ - *a finite or recursively enumerable set of actions*
- $E = \{e_1, e_2, \ldots\}$ - *a finite or recursively enumerable set of events*
- $\gamma : S \times A \times E \to 2^S$ - *a state transition function*

We can represent this system as a directed graph with S as a set of nodes and with the set of edges D:

$$D = \{(s_1, s_2) | s_1, s_2 \in S \wedge \exists a \in A, e \in E : \gamma(s_1, a, e) = s_2\}$$

For convenience we introduce a special *empty action* called *no-op* and a *neutral event* ϵ.

In planning we use the *state transition system* to model some particular enviroment. Through actions and events the state of the system can be changed. While we can control actions, the events on the other hand are modeling intrinsic dynamic properties of the enviroment and thus can not be controlled. The process of planning can be understood as search for a path in the graph induced from the *state transition system*.

Now we will informally introduce the terms of *planning domain*, *planning problem*, *plan* and *planner* with the first three of them ilustrated later by the example 1.

Planning domain is a term refering to the state transition system Σ.

Planning problem can be informally defined as a triple $P = (\Sigma, s_0, G)$:

- Σ - a planning domain
- s_0 - an initial state
- G - a specification of *goal conditions* (e.g. set of propositions which we want to be true in a goal state)[1]

Plan can be understood as a sequence of actions when considering the special case of *sequential planning* (i.e. no more than one action can be executed in a time). In general, actions can be organized into other than sequential structures in plan. A *valid plan* for a planning problem $P = (\Sigma, s_0, G)$ is any plan that starts in s_0 and conforms to G.

Planner is an entity which processes the *planning problem* $P = (\Sigma, s_0, G)$ as an input and outputs a *valid plan* for P.

Example 1. *In this example we will describe a simple* planning domain. *In this domain we will have vehicles that can move between places and transport packages. The situation is displayed in figure 1.1. The* package *is at* loc1*, the*

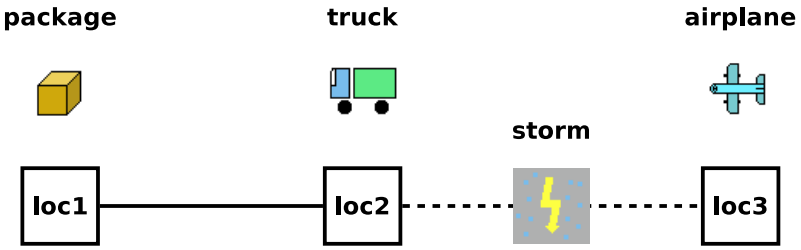


Figure 1.1: Logistic domain example

truck *is at* loc2 *and the* airplane *is at* loc3*. Locations* loc1 *and* loc2 *are connected with a road (line) while locations* loc2 *and* loc3 *are equipped with airport and it is possible for the* airplane *to move from one to another (dashed line) if there is no* storm *between them.*

There are 40 possible states for this system if we count 2 possible states of the storm*, 5 possible positions*[2] *for the* package *and 2 possible positions for each vehicle. Therefore in a state transition system* $\Sigma = (S, A, E, \gamma)$ *for this domain* $|S| = 2 \times 2 \times 2 \times 5 = 40$.

To enumerate the full list of actions from A we would have to fill in all available combinations of suitable[3] *constant symbols from figure 1.1 into these templates:*

$move\_[Vehicle]\_from\_[Location]\_to\_[Location]$
$load\_[Package]\_at\_[Location]\_to\_[Vehicle]$
$unload\_[Package]\_at\_[Location]\_from\_[Vehicle]$

[1] Specification of goal can also include restrictions on the plan or optimalization constraints.
[2] loc1,loc2,loc3,truck and airplane
[3] i.e. truck or airplane in place of Vehicle; loc1,loc2 or loc3 in place of Location; etc.

In the end we would have to include also the empty action *in the list.*

There are only two possible events in E not mentioning the neutral event ϵ*:*

```
begin_storm_between_loc1_and_loc2
end_storm_between_loc1_and_loc2
```

With the planning domain Σ defined, we can define a planning problem $P = (\Sigma, s_0, G)$ *by describing s_0 and G. The initial state s_0 can be any of the 30 states in S (let us suppose that the state s_0 is depicted at figure 1.1). The* goal conditions *can be described for example with the requirement that the* package *has to be at* loc3*.*

If there was no storm *we could have easily come up with the following sequential* plan *for the planning problem P:*

```
move_truck_from_loc2_to_loc1
load_package_at_loc1_to_truck
move_truck_from_loc1_to_loc2
unload_package_at_loc2_from_truck
move_airplane_from_loc3_to_loc2
load_package_at_loc2_to_airplane
move_airplane_from_loc2_to_loc3
unload_package_at_loc3_from_airplane
```

However the structure of the plan can be more complicated. For example if we had to wait for the storm *to end we would need to include an unknown number of* no-op *actions into our plan.*

Complexity of planning is mostly due to the size of the graph that can arise from the state transition system Σ. We can easily come up with the planing problem where the graph grows immensely beyond the possibilities of explicit representation. If we describe a system similar as in the example 1 with the road network of 100 places, 20 trucks and 10 packages we get a system with the approximate number of states $6 * 10^{60}$.

Three implicit representations used in planning are presented in the following section.

1.2 Classical planning

The classical planning is a branch of planning that was historically established in order to reduce the complexity of planning by introducing several restrictions to the general model (*state transition system*). The restrictions with their brief description[4] are as follows:

1. **Finite set of states** - the set of states of Σ is finite
2. **Full observability** - planner has full information about the state of the system Σ

[4]More details can be found in [20].

3. **Deterministic actions** - for every state $s \in S$ and action $a \in A$ the set $\gamma(s, \epsilon, a)$ contains only one state if any.

4. **Static system** - the set E is empty. This means that there are no events modelled in the system.

5. **Restricted goals** - the *goal conditions* for a planning problem are specified by a set of states G. Any plan that brings the system into a state from G is considered valid.

6. **Sequential plans** - the plan is defined as a sequence of actions.

7. **Implicit time** - actions and events have no duration

8. **Offline planning** - the state of the world which is modelled by Σ is not changed during the planning proces

In clasical planning we define a *restricted state transition system* as $\Sigma = (S, A, \gamma)$ without the set of events which is empty as a consequence of the restriction 4. The transition function γ is simplified to:

$$\gamma : S \times A \to S$$

This system can be also represented as a directed graph with the set of nodes S and the set of edges D:

$$D = \{(s_1, s_2) | s_1, s_2 \in S \wedge \exists a \in A : \gamma(s_1, a) = s_2\}$$

There are three representations for the classical planning which are proven to be equivalent in terms of expressive power [20].

- Set theoretic representation
- Classical representation
- State variable representation

These representations will be reviewed in the following three sections. The construction of a *restricted state transition system* $\Sigma = (S, A, \gamma)$ is described in each of them. The paragraph *Set of states* is always refering to S and in a similar manner the paragraphs *Set of actions* and *Transition function* are refering to A and γ. Any additional paragraphs are specific for the given representation.

1.2.1 Set theoretic representation

A representation that uses simple propositions for the planning domain description is called *set theoretic*. To create a set theoretic representation for particular enviroment, one have to define a set of all atomic propositions L.

Set of states in the set theoretic representation is defined as $S \subseteq 2^L$. The propositions that are present in a state $s \in S$ are true in this state.

Set of actions in the set theoretic representation contains elements defined as triples: $a = (precond(a), effect^+(a), effect^-(a))$ each component being a subset of L.

- $precond(a)$ - a set of propositions called *preconditions*
- $effect^+(a)$ and $effect^-(a)$ - sets of propositions called *effects* such that

$$effect^+(a) \cap effect^-(a) = \emptyset$$

Transition function for the set theoretic representation is based on set operations.

Let $s \subset L$ be the current state of the system. If $a \in A$ is an action that is *applicable* in s (i.e. $precond(a) \subseteq s$) then:

$$\gamma(s, a) = (s \cup effect^+(a)) \setminus effect^-(a)$$

If a is not applicable, $\gamma(s, a)$ is undefined. There is also an important property of γ: $\forall a \in A$ *applicable* in state $s \in S$ we can rely on $\gamma(s, a) \in S$.

1.2.2 Classical representation

This representation is based on *first-order logic*. It can be understood as a lifted version of the *set theoretic representation*.

To describe the classical representation we need to define a FOL[5] language $\mathcal{L}$ which contains a finite number of relation symbols, no function symbols and a finite number of constant symbols and variables. The language $\mathcal{L}$ is also further extended in order to include a *set of operators* which is to be described below.

Set of states in the classical representation is defined as $S \subseteq 2^{G_\mathcal{L}}$ where $G_\mathcal{L}$ is set of all *grounded atoms* of language $\mathcal{L}$.

Operators in the classical representation act as the templates for actions. Each operator $o \in \mathcal{O}$ from the *set of operators* of the language $\mathcal{L}$ is defined as:

$$o = (name(o), precond(o), effect(o))$$

- $name(o)$ is an expression representing the operator o. The expression takes the form $n(x_1, \ldots, x_k)$ where n is a domain-unique operator name and $x_1, \ldots, x_k$ are variables that appear in $precond(o)$ or $effect(o)$. The arity of the operator is represented by the number k.
- $precond(o)$ and $effect(o)$ are sets of literals from the language $\mathcal{L}$. (i.e. relations from $\mathcal{L}$ or their negations).

Set of actions in the classical representation is the set of all grounded instances of every operator $o \in \mathcal{O}$.

[5]First order logic

Atom negations. If B is a set of literals we will denote the set of all atoms in B with B^+ and the set of all atoms whose negations are in B with B^-.

Transition function for the classical representation is very similar to the set theoretic representation. Let $s \subset G_{\mathcal{L}}$ be the current state of the system. If $a \in A$ is an *applicable* action (here we require $precond^+(a) \subseteq s$ and $precond^- \cap s = \emptyset$) we can state the rest of the definition exactly as in the case of the set theoretic representation. That is:

$$\gamma(s, a) = (s \cup effect^+(a)) \setminus effect^-(a)$$

If a is not applicable, $\gamma(s, a)$ is undefined.
The following property of γ:
$\forall a \in A$ *applicable* in state $s \in S$ we can rely on $\gamma(s, a) \in S$
is preserved as well.

1.2.3 State variable representation

Motivation behind the state variable representation is explained in the example 2.

Example 2. *Using the classical representation, we can model the position of the package from example 1 with the flexible relation:*

$$packPos(P, X)$$

where $P \in \{package\}$ *and* $X \in \{loc1, loc2, loc3, truck, airplane\}$. *As we know that the* package *can not be simultaneously at two positions we can describe its position with the function:*

$$packPos : P \to X$$

The state variable representation is using functions *instead of* flexible relations.

Classes. Not all constant symbols used in the description of a planning domain can be used in the same way. When we have used the term *suitable constant symbols* in example 1 we had the concept of *classes* in mind. We will use the following definition of *class*:

Definition. Class T *is a set of constant symbols with common properties (e.g. a class of locations, a class of packages etc.)*
If c *is a constant symbol we will refer to it as an* object symbol *and denote its class as* D^c.

Example 3. *In the domain from the example 1 we can define following* classes*:*

$Location = \{\ loc1,\ loc2,\ loc3\ \}$
$Vehicle = \{\ truck,\ airplane\ \}$
$Truck = \{\ truck\ \}$
$Airplane = \{\ airplane\ \}$
$Package = \{\ package\ \}$

It is possible to define class as subset of another existing class (e.g. Truck *and* Airplane*).*

With introduction of classes it is convenient to use variables that can have range of possible values defined:

Definition. *An* object variable *x is a variable with a range of values D^x defined as a union of one or more classes.*

Definition. *A* term *in the state variable representation is either an* object symbol *or an* object variable*.*

Definition. *A k-ary* relation *is defined as an expression $r(a_1, \ldots, a_k)$ where r is a relation name and a_i is its i-th argument.*

For the i-th argument of relation r we define the range of values D^{r_i} as a union of one or more classes.

A rigid relation is defined by r:

$$r \subseteq D^{r_1} \times \ldots \times D^{r_k}$$

Example 4. *As an example of a rigid relation we can consider the way how the road network is defined in the example 1. It can be done with the following ternary relation:*

$reachable($`L1` $-$ *Location*, `L2` $-$ *Location*, `V` $-$ *Vehicle*$)$

The classes for each argument are listed after the arguments for simplicity.

Definition. *A k-ary* state variable *is defined as an expression $v_s(a_1, \ldots, a_k)$ where v is a state variable name and a_i is its i-th argument. Symbol $s \in S$ is refering*[6] *to a state of the restricted transition system Σ.*

For the i-th argument of the state variable v we define the range of values D^{v_i} as a union of one or more classes.

In a similar manner we define the range of the state variable v itself (denoted as D^v).

A function is defined by v:

$$v : S \times D^{v_1} \times \ldots \times D^{v_k} \to D^v$$

Example 5. *The position of the package from the example 1 can be described with the following state variable:*

$packagePos:$ *Package* $\to$ *Location* $\cup$ *Vehicle*

The state variable has only one argument with range of values defined with the class Package *and the range of the state variable itself is defined by the union of two classes (*Location *and* Vehicle*).*

Definition. *An* operator *is defined as a triple*

$$o = (name(o), precond(o), effect(o))$$

where:

- *$name(o)$ is an expression $n(u_1, \ldots, u_k)$ where n is a unique name of o and $u_1, \ldots, u_k$ are object variables that appear anywhere in $precond(o)$ or $effect(o)$.*

[6] If omited, the expression $v(a_1, \ldots, a_k)$ refers to the current state of Σ

- *precond*(o) *is a set of expressions on state variables and relations*
- $eff ect(o)$ *is a list of value assignements to state variables in the form:*

$$v(x_1, \ldots, x_n) \leftarrow x$$

Example 6. *We will refer again to the example 1 for context. As an example of planning operator in the state variable representation we will describe operator* LoadPackage *in the following pseudo-code:*

Operator`[` ***name****: loadPackage (*`V, P`*)*
precond*: vehicleLoc(*`V`*) = packagePos(*`P`*)*
effect*: packagePos(*`P`*)* $\leftarrow$ `V` `]`

We have used two state variables vehicleLoc *and* packagePos *in the set of expressions precond(loadPackage) to ensure that the vehicle and package are at the same location. The assignment of* `V` *to the state variable* `packagePos(P)` *describes actual act of "loading" the package* `P` *to the vehicle* `V`*.*

Language $\mathcal{L}$ used by the state variable representation consists of these components:

- C - a set of object symbols
- V - a set of object variables
- R - a set of relations
- X - a set of state variables
- O - a set of operators

Example 7. *In the following code we will describe the language* $\mathcal{L}$ *for the planning domain from the example 1:*

```
; set of object symbols
; is union of all classes { Location ∪ Vehicle ∪ Package ∪ Truck ∪
    Airplane }
C = { loc1, loc2, loc3, truck, airplane, package }

; set of object variables
V = { L1, L2, L3, L4 − Location,
      V1, V2, V3, V4 − Vehicle,
      P1, P2 − Package }
; set of relations
R = { reachable(L1, L2, V1) }

; set of state variables
X = {   vehicleLoc(truck) ∈ Location
        vehicleLoc(airplane) ∈ Location
        packagePos(package) ∈ Location ∪ Vehicle }

; set of operators
O = {
```

Operator[***name:*** *loadPackage* (V2, P1)
precond: *vehicleLoc*(V2) = *packagePos*(P1)
effect: *packagePos*(P1) ← V2],

Operator[***name:*** *unloadPackage* (V3, P2)
precond: *packagePos*(P2) = V3
effect: *packagePos*(P2) ← *vehicleLoc*(V3)],

Operator[***name:*** *moveVehicle* (V4, L3, L4)
precond: *vehicleLoc*(V4) = L3, *reachable*(L3,L4,V4)
effect: *vehicleLoc*(V4) ← L4] }

Set of states in the state variable representation is defined as:

$$S \subseteq \prod_{x \in X} D^x$$

A state $s \in S$ is defined by the current values of all state variables from X:

$$\{(x = c) | x \in X \wedge c \in D^x\}$$

Set of Actions in the state variable representation contains all grounded operators from O. If $o \in O$ is an operator with $name(o) = n(u_1, \ldots, u_k)$ than we can obtain its grounded version by assigning an object symbol $b_i \in C$ to object variable u_i for $i \in 1, \ldots, k$. We always suppose that $D^{b_i} \subseteq D^{u_i}$.

Transition function in the state variable representation: Let $s \in S$ be the current state of Σ. If $a \in A$ is applicable (i.e. all expressions on state variables from $precond(a)$ holds in s) then the state $\gamma(s, a)$ can be obtained by performing all assignments from $effect(a)$: If there is an assignment $v(x_1, \ldots, x_n) \leftarrow x$ in $effect(a)$ we change the value of v:

$$v_{\gamma(s,a)}(x_1, \ldots, x_n) = x$$

1.3 Extensions of classical planning

In this section other branches of planning are mentioned briefly. It is beyond the scope of this work to go into details. Nevertheless many challenges arise from these areas and there is a good reason to keep their existence in mind while designing a new approach for modelling of planning problems. Real world applications of automated planning often fall into one or more of the following categories.

1.3.1 Temporal planning

When the restriction 7 is relaxed we talk about temporal planning. With the non-zero duration of actions and events comes the need for representation of time and parallel plans are possible.

1.3.2 Planning with numeric fluents

With the introduction of numeric fluents the restriction 1 has to be relaxed. Numeric resources in particular are very important in real world applications.

1.3.3 Planning with uncertainty

Planning systems used in autonomous robots or vehicles are often relaxing restriction 3 in order to model realistic systems. Probabilistic methods are incorporated into planners for domains where the results of actions can not be determined exactly.

1.3.4 HTN Planning

The abbreviation *HTN* stands for *Hierarchical Task Network*. It refers to approach that uses a similar representation as in the case of the classical planning. The main difference is in the planning process itself. A goal for an *HTN-planner* consists of set of *tasks* which can be decomposed into subtasks by using *methods* described along with other input data. This decomposition proces stops at the level of *primitive tasks*. These primitive tasks are then selected according to the *methods* in order to perform the original task.

2. Knowledge representation

The question how do we represent knowledge in the Automated Planning is not a simple one. We need to be more specific in order to answer correctly.

In the begining we need to get some formal description of "the world". There are descriptive languages created for this purpose. These languages are also used to describe *input data* for planners. The term input data is refering to the description of the *planning domain* and the *planning problem* from chapter 1.

In the next step, the input data are usually processed in order to create the *grounded representation* of the given planning problem. This grounded representation is encoded into *data structures* used by the planner itself.

The last step would include *plan validation* and *interpretation* which is beyond the scope of this work. Therefore we will focus on the first two steps.

Transformation of the *input data* to the *data structures* is an important preprocessing phase of the planning process. The overall complexity of this preprocessing phase depends on the choice of the language for domain description on one side and target planner on the other.

This chapter presents some of the main input data languages used today as well as some of the principles for planning problem encoding. We will make the choice of the preferred language and data representation for this work in the end of this chapter together with overall summary.

2.1 Planning domain modelling

In this section we will review some languages used for planning domain modelling. Despite our prevailing interest in classical planning we present also languages that are specialized in temporal planning with the use of resources.

2.1.1 PDDL

Language *PDDL* (Planning Domain Definition Language) was developed in 1998 [19] as a problem-specification language for the AIPS-98 planning competition. It has been used as a standard language for the planning competitions ever since and has became a standard language for representation and exchange of planning domain models. The key design principle to keep the language neutral is best stated in a motto *"physics, not advice"* [18].

The language is using lisp-like syntax in order to be easily parsed by planners for further processing. The definitions of planning domain and planning problem are separated into different files which makes it possible to define multiple planning problem statements based on a single domain definition. The expressive power of the language can be controlled by specifying the set of requirements (`:strips` is the default). This feature makes it possible for the planners to use only a subset of *PDDL*'s language constructs usually resulting in simplified code used for domain and problem encoding. Also this provides a natural way one can use to specify the class of planning problems by naming the used requirements and *PDDL* version.

```
(define (domain Example)
    (:requirements :strips :typing)
    (:types
        Package Location Vehicle - object
        Truck Airplane - Vehicle
    )
    (:predicates
        (vehicleAt ?a - Vehicle ?b - Location)
        (reachable ?a - Location ?b - Location ?c - Vehicle)
        (packageAt ?a - Package ?b - (either Vehicle Location ))
    )
    (:action moveVehicle
        :parameters (?V - Vehicle ?L1 - Location ?L2 - Location )
        :precondition (and
            (vehicleAt ?V ?L1)
            (reachable ?L1 ?L2 ?V)
        )
        :effect (and
            (vehicleAt ?V ?L2)
            (not (vehicleAt ?V ?L1))
        )
    )
    ...
```

Figure 2.1: Example domain in PDDL

Certain modifications were made since the language was introduced. A short list of the main language extensions follows:

- `PDDL 2.1`: numeric expressions, metric, durative actions [8]
- `PDDL 2.2`: derived predicates, timed init literals [6]
- `PDDL 3.0`: constraints on plan trajectories [10]

Classical planning domains in *PDDL* are usually modelled by defining the classes of objects, the relations between them and the actions with preconditions and effects. An example of *PDDL* description of planning domain from example 1 is at figure 2.1.

2.1.2 IxTeT

IxTeT (**I**nde**x**ed **T**im**e** **T**able) is a plan space temporal plannning system that can work with uncertainties and resources. It features a formalism for planning domain modelling.

According to [16] this formalism uses *multi-valued state attributes* and *resource attributes* to describe the state of system. These attributes can be temporally quallified. The former with the predicates *hold* (value asserted over an interval) and *event* (value is instantly changed) whereas the latter with the predicates *use*, *consume* and *produce*.

To describe a change in the enviroment, the *IxTeT* formalism provides syntax for describing planning operators that can be temporally binded to *time-points*. Operators can define sub-tasks (HTN planning). In each operator, assertions are stated to decide applicability, changes are described and sub-tasks specified.

2.1.3 NDDL

NDDL is a domain description language used by *EUROPA* which is a framework for modelling and solving problems in planning, schedulling and constraint programming developed by *NASA* [4].

The language has a syntax that bears close resemblance to *C++* programming language. We can define classes if:

- there is a need to refer to problem specific objects in operators (e.g. locations - each problem can have a different map)
- we need to refer to objects with internal structure (for example a *Path* can have start, end and lenght)
- we need objects that change their state in time (i.e. agents that move between locations)

Once defined, a class can be inherited and/or reused by other classes. The following code example shows the definition of class `Rover` [3]:

```
class Rover {
  Commands commands;
  Navigator navigator;
  Instrument instrument;
  Battery mainBattery;

  Rover(Battery r){
    commands = new Commands();
    navigator = new Navigator();
    instrument = new Instrument();
    mainBattery = r;
  }
}
```

There is no concept similar to planning operators in this language. The model of planning domain using *NDDL* relies on temporally anotated predicates (tokens) and timelines. Changes in the system are constrained by special rules on predicates.

2.1.4 AML

AML is a modelling language used in *ASPEN* which is another system for planning and schedulling developed by *NASA* [23]. The following language constructs play the key role in the planning domain description:

- State timelines - evolution of a discrete state variable over time
- Resources and reservations - modelling of resources available in the domain

- Activities - planning operators with possible bindings in time
- Temporal constraints - modeling of relations between the activities

2.1.5 ANML

Developement of the **A**ction **N**otation **M**odelling **L**anguage started in 2006 [26]. The language aims to be an alternative to all previously mentioned languages. It supports both generative and HTN planning models and it is possible to translate ANML to PDDL in most cases.

Using the language we are able to model domains with usage of resources, temporally qualified actions and (possibly numeric) function symbols that can take objects as arguments (state variables). Both numeric and object variables can be declared. The language allows definition of vectors.

Each action in ANML has its name, the list of parameters (optional), duration and might also define its internal variables and functions. Here is a code sample of an ANML action for an autonomous robot:

```
action move(location from to) {
        duration := 5;
        [all] {
                batterycharge > 2.0;
                position == from :-> to;
                batterycharge :consumes 2.0;
        }
}
```

The following temporally qualified statements can be contained in an ANML action:

- *condition* - expression on variables
  `batterycharge > 2.0;`
- *effect* - variable assignments
  `position :-> to`
- *change statement* - relative changes of resources
  `batterycharge :consumes 2.0;`

In the example above a condition and effect are combined into single line:
`position == from :-> to`
Temporal qualification is specified for whole blocks of statements with an interval (e.g. `[all]`) or a single time point. More than one block can be specified in one action.

2.2 Planning problem encodings

Once described in some modelling language a problem has to be encoded into some structure that can be used by a planner. This work is not dealing with the planning process performed by planners but we are interested in possibilities for planning problem encoding. In this section we list some of these possibilities.

2.2.1 Propositional encoding

Problems from classical planning are easily encoded as sets of propositions. This approach is implementing the set theoretic representation as mentioned in 1.2.1. Planners using this representation perform set operations defined by the transition function γ to explore the state space of given problem.

The state of the logistic domain from the example 1 can be encoded[1] in the set theoretic representation as the set of atoms:

```
s = {   package_loc1 truck_loc2 airplane_loc3
        truck_route_loc1_loc2 truck_route_loc2_loc1
        airplane_route_loc2_loc3 airplane_route_loc3_loc2 }
```

Advantage of the propositional encoding is in its simplicity.

2.2.2 Multivalued state variables

Encodings which use multivalued state variables are growing popular in classical planning comunity. The SAS^+ formalism [2] in particular has inspired many researchers ([12],[5] or [28]). The idea of the *SAS+* formalism is very close to the state variable representation mentioned in 1.2.3 since in both cases the state of the system is determined by a finite set of state variables assignments.

The SAS+ formalism uses state variables instead of atomic propositions as opposed to the set theoretic representation or the classical representation. The advantage is in implicit representation of mutual exclusion of some facts as explained in example 2. Another difference is in the definition of operators. Instead of the *preconditions* and *effects* there are also *prevailing conditions* (i.e. preconditions that are not changed in the operator).

Finite state automata. An approach based on the FSA is described in [28]. The idea is to use a FSA for representation of each state variable from a *SAS+* representation of planning problem. Not only the representation is using finite state automata equal to the *SAS+* representation but it is also possible to combine those automata together.

Although the question which automata should be combined is not fully answered yet this approach has opened a new possibility for translation of propositional domains into *SAS+* formalism. The method presented in [28] uses binary FSA as a start point.

2.3 Summary

Languages. All presented languages has their strenghts and weaknesses. They have been developed for different purposes and there is no way to select the best one. We can however notice that on the one side we have languages developed with certain domains in mind (NDDL,AML) and on the other side those developed for general purpose planning (PDDL,IxTeT,ANML).

[1] In classical planning - i.e. without events (storm)

For the purpose of this work we will prefer the language *PDDL*. The main reason for this decision is the fact that it is the most widespread language used by the planning comunity.

Encodings. The presented approaches to planning problem encoding are selected to represent the two main streams used in the classical planning. We are aware of other possibilities in existence such as planning as SAT [15], planning as PT-net unfolding [14] or planning using BDD [27]. However the encoding using multivalued state variables was the one we found most inspirative for our work.

Combination. In our new approach we would like to design planning domains using directly multivalued state variables instead of propositions. Our motivation is twofold:

1. domains created in this way should be easily translated to the *SAS+* representation
2. translation from the state variable representation into propositional representation is less difficult than the reversed direction. Due to this fact it should be easy to retain compatibility with *PDDL*.

3. Knowledge modelling

If we decide upon the input data format there is still one question left:

How do we obtain the input data?

There is undeniable progress in the field of the automated planning. Planners are capable of solving various problems more and more efficiently and there is a growing number of applications for the planning technology in various areas of human activity. The input data modelling plays the key role here.

Our question can be answered by knowledge engineering which is an essential field of the AI research. The knowledge modelling for the automated planning in particular is discussed in this chapter.

According to [32] the process of design for planning domain models has following phases:

1. Requirements Specification
2. Knowledge Modelling
3. Model Analysis
4. Deploying model to planner
5. Plan synthesis
6. Plan Analysis and Post-Design

There are various tools in existence which are designed to assist through various phases of this design process. Some of them are specialized to certain expert areas where there is a need for exploiting benefits of planning technology such as bussiness process modelling [11]. In the following text we do not mention these specialized tools since we are interested mostly in general-purpose planning.

In the following sections we will present the two most significant contemporary knowledge engineering tools used for the general-purpose planning. In the end of this chapter we will focus on their solution of the phase 2.

3.1 GIPO

Historically the tool *GIPO*[1] is one of the first graphic tools used for building planning domain models. The program *GIPO III*, which features an experimental GUI and tools environment, won the IKEPS competition in the class of general Knowledge Engineering Tools for AI Planning in 2005.

The *GIPO* employs an *object centric view* through the planning domain design proces. The basic assumption of the object centric view is that within any problem scenario that presents a planning problem there will be objects that are changed in some way during the execution of plans [24]. The language OCL_h [25] is employed in the program for the internal data representation as a consequence.

[1] **G**raphical **I**nterface for **P**lanning with **O**bjects

The program is implemented in *Java* programming language and its interface supports the two main operating modes. In the first mode user can design only the classical planning domains whereas in the other mode the *HTN* features are enabled to support HTN planning.

As for the planning domain design there is more than one way to acomplish the task using various integrated tools and editors. The starting point for each of them is the definition of so called *sorts*. The objects within the same sort may undergo same changes and they also posses equal sets of properties.

Let us consider for example sort `vehicle` with cars as objects. After defining the sort in *GIPO* we would be able to define the states such as `parked` and `on_the_way` or the properties like `fuel_level`.

The process would continue with design of *state machines* and operators. Finally, *GIPO* provides tools for task definition, domain validation and manual plan stepper as well as an interfece through which the external planners can be called.

3.2 itSIMPLE

The tool *itSIMPLE* was first introduced as a tool for modelling planning domains in 2005 [33]. Since then the tool evolved into integrated enviroment which can be considered the leading application for planning domain design today. It is under active developement and there are notable achievements in its history.

For example the enviroment was used for the design and investigation of a real application of planning technology in the petroleum industry [31]. Although some simplifications were made to meet the abilities of used planners, the overall approach provided promising results. From the knowledge engineering point of view the *itSIMPLE* proved to be a capable tool for the design of the complex planning domain which featured time, numeric resources, quality metrics and optimalization.

The domain modelling integrated enviroment *itSIMPLE* which is currently available in version `3.5` [29] is written in *Java* programming language and provides complex GUI which supports planning domain design and evaluation in all the five stages: requirements specification, modelling, model analysis, testing with planners and plan evaluation [30]. During this process, the *UML* diagrams are used to present modelled domain to user. Features like export into *PDDL* and domain model analysis with *Petri nets* are available while using *XML* for converting one representation to another.

3.3 Lessons learned

The key task of the knowledge modelling phase is to capture the information about possible changes in target dynamic system.

Therefore we have focused on the methods for change-modelling used by the presented KE tools. Both *GIPO* and *itSIMPLE* use the concept of operators. The main difference is in the way how they describe them. From the user perspective the description of the operators is the most complex task no matter which tool

we use. In the following two sections we will summarize our experience with the operator description in both presented tools.

3.3.1 Operators in *itSIMPLE*

In the *itSIMPLE*, the operators are declared in the class diagram. An operator is later defined in the state machine diagrams of classes changed by the operator. Preconditions and effects of the operator are defined with a formal constraint language of *UML*. The constraints are entered by user in the state machines diagrams. All the features of the *PDDL*[2] language can be used, but there is a limit on the number of arguments for the predicates.

The maximal arity of predicate is limited to 2 as a consequence of the way predicates are defined. We can either define a relation between two classes or declare an attribute for a class. Only two entities figures in both cases.

The change can be represented either with flexible relations or fluents (attributes and most relations are translated as object fluents when exported to *PDDL* version `3.1`).

Summary.

- Although there are many advantages arising from the usage of *UML* diagrams we found the fact that the operator definition is scattered in many diagrams confusing.
- The limited arity of the predicates certainly does not pose a constraint to expressivity but we believe that certain constructs could be expressed in a more compact way if we could use predicates with more arguments.
- It is convenient to use object fluents instead of propositions in most cases.

3.3.2 Operators in *GIPO*

The operators in *GIPO* are described as primitive transitions[3]. To define the primitive transitions we have to define states for each sort. The state of the sort is described by a set of predicates and these sets are used to describe prevailing conditions, necessary changes and conditional changes in each primitive transition.

Another way is to define state machines for each object sort in *object life history editor*. All these state machines are displayed in a special diagram with two kinds of vertexes and three kinds of edges. This diagram can be exported to *OCL*.

Only flexible relations can be used for description of changes in *GIPO*.

Summary.

- Usage of the integrated tools requires complex understanding of the program.

[2] up to the version 3.1

[3] This corresponds with the primitive tasks in HTN planning mentioned in the first chapter.

- Description of change with the flexible relations is often cumbersome and complicated even for simple problems.
- The state machines employed in the *object life history editor* can be represented with the state variables.

4. State variable based modelling

In this chapter we would like to propose a new approach to modelling of planning problems based on the concept of state variables. We will try to explain our motivation in the following paragraphs.

Why state variables ? There is a growing number of planners using the *SAS+* representation. Since the process of extracting the state variables from the propositional representation is complicated it would be of great benefit if we could bypass it.

Both of the major KE tools presented (*GIPO* and *itSIMPLE*) provide a way how to represent the concepts which are very closely bound to the concept of multivalued state-variables when modeling planning domains (*GIPO* operates on *state machines* and *itSIMPLE* uses *state machine diagrams*). This lead us to the conclusion that the state variables are important building blocks for the majority of problems in contemporar planning.

The concept of state variables is also the half-way between the traditional proposition-based approach of the classical planning and the timeline based approach used by planning groups in space research. An approach based on state variables might help to fill the gap between the two branches.

Why do we need a new approach ? All KE tools mentioned so far are designed to guide the user through the process of planning domain design and let her input some knowledge which is later compiled into data formats processed by a particular planner. There are two points to be noted:

- There is no unified way for eliciting knowledge from the user.
- There is a standard language (*PDDL*) used by majority of planners.

This means that every KE tool is forced to conform to the standard, set by *PDDL*, on one end and has no frame to fit in on the other. If we start from the user we end up translating "human readable knowledge" into *PDDL*. This task is not easy because we do not exactly know how does the "human readable knowledge" looks like.

We propose to start from the other end and seek a suitable formalism on the way from *PDDL* to "human readable knowledge". In this way we hope to find a new perspective in planning problem modelling. Rather than providing an user interface we aim to provide a well defined platform for planning problem modelling on which various user interfaces can be based.

4.1 Formalism description

The proposed formalism is based on the *state variable representation* from classical planning. The main goals to achieve were:

- clear distinction between the planning domain and the planning problem
- expressive power comparable to STRIPS

- implicit support for typing
- space for further extension of expressive power

This section provides a complete description of the proposed formal model including several examples to demonstrate the modelling framework.

4.1.1 Planning domain

In our formalism, the planning domain is defined as $\Sigma = (\mathcal{H}, \mathcal{R}, \mathcal{V}, \mathcal{O})$ where:

- $\mathcal{H}$ - a class hierarchy
- $\mathcal{R}$ - a set of declarations of rigid relations
- $\mathcal{V}$ - a set of declarations of state variables
- $\mathcal{O}$ - a set of planning operators

The class hierarchy defines the classes[1] of objects and variables in the planning domain. The relevant definitions are as follows:

Definition 1. Class
Class T is defined as a triple $(name(T), subc(T), const(T))$ where

- $name(T)$ is a class name unique in the domain
- $subc(T)$ is a set of *classes* (subclasses of class T)
- $const(T)$ is a set of *problem independent* constant symbols

Both sets $subc(T)$ and $const(T)$ can be empty.

Classes are structured into a tree hierarchy. A *subtree of class* T is a set of classes denoted $subtree(T)$ and defined recursively as:

$$subtree(T) = \{T\} \cup \bigcup_{R \in subc(T)} subtree(R)$$

For each two classes A and B in the hierarchy we demand that one of the following conditions holds:

1. $B \in subtree(A)$
2. $A \in subtree(B)$
3. $subtree(A) \cap subtree(B) = \emptyset$

When convenient we will use $\mathbf{name(T)}$ instead of $\mathbf{T}$ for simplicity. This is always possible without any confusion because the class names are unique.

Definition 2. Root class
Root class in a planning domain Σ is a special class denoted T_Σ with

$$name(T_\Sigma) = object$$

The class hierarchy $\mathcal{H}$ of Σ is always defined as:

$$\mathcal{H} = subtree(T_\Sigma)$$

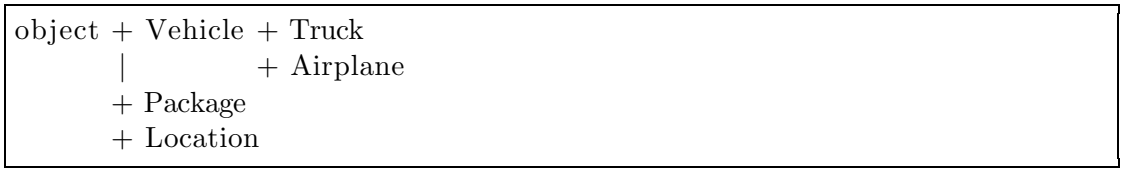


Figure 4.1: Tree structure

Example 8. *A tree structure for the logistic domain from example 1 is shown in figure 4.1.*

Suppose that $\Sigma = (\mathcal{H}, \mathcal{R}, \mathcal{V}, \mathcal{O})$ *is our logistic domain. The class hierarchy* $\mathcal{H}$ *with the tree structure from the figure 4.1 is:*

$$\mathcal{H} = \{Truck, Airplane, Package, Location, Vehicle, object\}$$

with the following class definitions:

$$(Truck, \emptyset, \emptyset)$$
$$(Airplane, \emptyset, \emptyset)$$
$$(Package, \emptyset, \emptyset)$$
$$(Location, \emptyset, \emptyset)$$
$$(Vehicle, \{Truck, Airplane\}, \emptyset)$$
$$(object, \{Vehicle, Package, Location\}, \emptyset)$$

Notice that this example does not contain any problem independent constants *and consequently:*

$$\forall C \in \mathcal{H} : const(C) = \emptyset$$

If we introduce a new class `gate` *into our logistic domain to control ground trafic between places we may need to define class* `gate_state` *like this:*

$$(gate\_state, \emptyset, \{open, closed\})$$

and include it in the tree structure by redefining the root class:

$$(object, \{Vehicle, Package, Location, \mathbf{gate\_state}\}, \emptyset)$$

Constants open *and* closed *in the class gate_state are* problem independent. *They retain their semantics in any planning problem related to* Σ.

Relation declarations in our formalism represent the rigid relations in the domain.

Definition 3. Declaration of relation

Declaration of relation with arity n is a tuple:

$$r = (name(r), args(r))$$

where:

- $name(r)$ is name for relation r; unique in domain Σ

[1] In similar way as types are defined in *:types* section of a PDDL domain definition file

- $args(r)$ is vector of arguments $(a_1, \ldots, a_n)$ where $\forall i : a_i \in \mathcal{H}$

Example 9. *Let us consider a simple planning domain Σ with a robot that can move from one location to another. Let the class hierarchy of Σ be:*

$$\mathcal{H} = \{(robot, \emptyset, \emptyset), (location, \emptyset, \emptyset), (object, \{robot, location\}, \emptyset)\}$$

Now let us declare a rigid relation r:

r	$=$	$(name(r), args(r))$
$name(r)$	$=$	$Adjacent$
$args(r)$	$=$	$(Location, Location)$

Declarations of state variables are included to allow the description of the dynamic properties of the planning domain. Each state variable has its *range* of values which can change throughout the planning process.

Definition 4. Declaration of state variable

Declaration of state variable with arity n
is a triple:

$$v = (name(v), args(v), range(v))$$

where:

- $name(v)$ is a name for the state variable v; unique in domain Σ
- $args(v)$ is a vector of arguments $(a_1, \ldots, a_n)$ where $\forall i : a_i \in \mathcal{H}$
- $range(v)$ is set of classes such that $range(v) \subseteq \mathcal{H}$

Example 10. *State variables can be used to model dynamic properties of one or more objects.*

Let us first reconsider the robot domain from example 9 to show how we can model a property of a single object.

We can declare *the following state variable v in the domain:*

v	$=$	$(name(v), args(v), range(v))$
$name(v)$	$=$	$robotLocation$
$args(v)$	$=$	$(Robot)$
$range(v)$	$=$	$\{Location\}$

This declaration yields a variable to store information about the robot's position for every robot defined later in a planning problem.

For our next example let us describe a building kit domain. *In a building kit domain we have components that can be connected together through joint points. Each component has a fixed number of joint points on its surface. Our example state variable can be declared using two classes $(Joint, \emptyset, \emptyset)$ and $(ConnectionState, \emptyset, \{connected, disconnected\})$:*

v	$=$	$(name(v), args(v), range(v))$
$name(v)$	$=$	$jointConnection$
$args(v)$	$=$	$(Joint, Joint)$
$range(v)$	$=$	$\{ConnectionState\}$

This declaration yields a state variable for every two joints in the domain. Using this description we can build compounds from available components while the state of the system is described with the state variables based on the declaration of jointConnection.

Object variables and terms are used in our formalism instead of variables and constants.

Definition 5. Term

A *term* is defined as an entity with a reference to a class from $\mathcal{H}$. If t is a *term* we denote its class as $L(t)$.

This definition of term leaves our formalism open for further expansion. An example will be mentioned in the end of this chapter (section 4.2.2).

Definition 6. Term - class compatibility

If $T \in \mathcal{H}$ and t is a term, we say that t *is compatible with* T iff:

$$L(t) \in subtree(T)$$

Definition 7. Term - class set compatibility

If $M \subseteq \mathcal{H}$ is a set of classes we say that t *is compatible with* M iff:

$$\exists T \in M : L(t) \in subtree(T)$$

Remark. *Here we would like to stress the following facts:*

- *Every* constant symbol *in our formalism is a* term.
- *If* $T \in \mathcal{H}$ *and the set of problem independent constants* $const(T)$ *is not empty, than:*
$$\forall c \in const(T) : L(c) = T$$

Definition 8. Object variable

An *object variable* b is variable with a range $D(b) \subseteq \mathcal{H}$. Any *term* t that is compatible with $D(b)$ can be assigned to b.

We will denote the term assigned to b as $\bar{b}$.

Definition 9. Object variable - class compatibility

An object variable b *is compatible with class* $T \in \mathcal{H}$ iff:

$$D(b) \subseteq subtree(T)$$

Expressions which refers to declarations of *relations* and *state variables* described earlier, are the building stones of the operators in our formalism. There are two kinds of expressions:

- *Relation expressions* are based on declarations from $\mathcal{R}$
- *Transition expressions* are based on declarations from $\mathcal{V}$

Definition 10. Relation expression

A *Relation expression* E based on n-ary relation declaration $r \in \mathcal{R}$ is defined as a tuple:

$$E = (r, (x_1, \ldots, x_n))$$

where $(x_1, \ldots, x_n)$ is a vector of object variables. If $args(r) = (a_1, \ldots, a_n)$ we demand that:

$$\forall i \in 1, \ldots, n : x_i \textit{ is compatible with } a_i$$

We will use notation $(name(r), (x_1, \ldots, x_n))$ instead of $(r, (x_1, \ldots, x_n))$ when convenient.

Definition 11. Transition expression

A *Transition expression* E based on n-ary state variable declaration $v \in \mathcal{V}$ is defined as a tuple:

$$E = (v, (x_1, \ldots, x_n, \mathbf{f}, \mathbf{t}))$$

where $(x_1, \ldots, x_n, \mathbf{f}, \mathbf{t})$ is a vector of object variables. If $args(v) = (a_1, \ldots, a_n)$ we demand that:

$$\forall i \in 1, \ldots, n : x_i \textit{ is compatible with } a_i$$

For classes of two last object variables we demand only compatibility with at least one class from $range(v)$:

$$\exists T_1, T_2 \in range(v) : \mathbf{f} \textit{ is compatible with } T_1 \ \wedge \ \mathbf{t} \textit{ is compatible with } T_2$$

We will use notation $(name(v), (x_1, \ldots, x_n, \mathbf{f}, \mathbf{t}))$ instead of $(v, (x_1, \ldots, x_n, \mathbf{f}, \mathbf{t}))$ when convenient.

Among the transition expressions we recognize two special cases:

Definition 12. Expression types

Let $v \in \mathcal{V}$ be n-ary state variable declaration.

- A *prevailing* transition expression is $(v, (x_1, \ldots, x_n, \mathbf{e}, \mathbf{e})$
- A *non-prevailing* transition expression is $(v, (x_1, \ldots, x_n, \mathbf{e}, \mathbf{f})$ where $\mathbf{e} \neq \mathbf{f}$

Operators describes changes, which may occur in the planning domain, on abstract level.

Definition 13. Operator

Operator $o \in \mathcal{O}$ is defined as a tuple $o = (name(o), args(o), expr(o))$ where:

- $name(o)$ is name of planning operator which is unique in domain Σ
- $args(o)$ is set of all object variables used in expressions in operator o
- $expr(o)$ is a set of expressions

Example 11. *In this example we provide a complete description*[2] *of the planning domain from example 1. In the following text we describe the content of each set* $\mathcal{H}$,$\mathcal{R}$,$\mathcal{V}$ *and* $\mathcal{O}$.

For the class hierarchy $\mathcal{H}$ *we will refer to the example 8 without the class* T_6 *which was added only to demonstrate the concept of problem independent constants.*

The set $\mathcal{R}$ *of relation definitions in this planning domain contain only one relation:*

r	$=$	$(name(r), args(r))$
$name(r)$	$=$	$reachable$
$args(r)$	$=$	$(Location, Location, Vehicle)$

The set $\mathcal{V}$ *of state variable declarations contains two state variables:*

v_1	$=$	$(name(v_1), args(v_1), range(v_1))$
$name(v_1)$	$=$	$packagePos$
$args(v_1)$	$=$	$(Package)$
$range(v_1)$	$=$	$\{Location, Vehicle\}$
v_2	$=$	$(name(v_2), args(v_2), range(v_2))$
$name(v_2)$	$=$	$vehiclePos$
$args(v_2)$	$=$	$(Vehicle)$
$range(v_2)$	$=$	$\{Location\}$

And finally the set of operators $\mathcal{O}$ *which contains three operators:*

$definition$			*comment*
o_1	$=$	$(name(o_1), args(o_1), expr(o_1))$	
$name(o_1)$	$=$	$moveVehicle$	
$args(o_1)$	$=$	$\{L1, L2, V\}$	$\{Location, Location, Vehicle\}$
$expr(o_1)$	$=$	$\{(r, (L1, L2, V)),$	$reachable(L1, L2, V)$
		$(v_2, (V, L1, L2))\}$	$vehiclePos(V) : L1 \rightarrow L2$
o_2	$=$	$(name(o_2), args(o_2), expr(o_2))$	
$name(o_2)$	$=$	$loadPackage$	
$args(o_2)$	$=$	$\{P, L, V\}$	$\{Package, Location, Vehicle\}$
$expr(o_2)$	$=$	$\{(v_2, (V, L, L)),$	$vehiclePos(V) : L \rightarrow L$
		$(v_1, (P, L, V))\}$	$packagePos(P) : L \rightarrow V$
o_3	$=$	$(name(o_3), args(o_3), expr(o_3))$	
$name(o_3)$	$=$	$unloadPackage$	
$args(o_3)$	$=$	$\{P, L, V\}$	$\{Package, Location, Vehicle\}$
$expr(o_3)$	$=$	$\{(v_2, (V, L, L)),$	$vehiclePos(V) : L \rightarrow L$
		$(v_1, (P, V, L))\}$	$packagePos(V) : V \rightarrow L$

4.1.2 Planning problem

The *planning problem* relevant to a *planning domain* $\Sigma = (\mathcal{H}, \mathcal{R}, \mathcal{V}, \mathcal{O})$ is defined as 5-tuple $\mathcal{P}_\Sigma = (\mathcal{C}, \mathcal{D}, \mathcal{F}, \mathcal{I}, \mathcal{G})$.

- $\mathcal{C}$ - a typed set of constant symbols

[2]We left aside the *storm* since we do not model events.

- $\mathcal{D}$ - a set of relation definitions
- $\mathcal{F}$ - a set of state variable definitions
- $\mathcal{I}$ - an initial state
- $\mathcal{G}$ - a set of goal conditions

All examples in this section are refering to the planning domain definition in example 11 and situation from figure 1.1.

Constant symbols defined in the planning problem $\mathcal{P}_\Sigma$ are representing particular objects or properties of the world described by the planning domain Σ. Let B be the set that contains all constant symbols defined in $\mathcal{P}_\Sigma$[3].

The set $\mathcal{C}$ is defined as:

$$\mathcal{C} = B \cup \{ \bigcup_{T \in \mathcal{H}} const(T) \}$$

Sometimes we need to refer to the set of constants that are compatible with a particular class T:

$$\mathcal{C}\langle T \rangle = \{ \ c \mid c \in \mathcal{C} \wedge L(c) \subseteq subtree(T) \}$$

It is also convenient to define a set of constants that are compatible with a set of classes $Q \subseteq \mathcal{H}$:

$$\mathcal{C}[Q] = \bigcup_{T \in Q} \mathcal{C}\langle T \rangle$$

Example 12. *The set $\mathcal{C}$ contains the following constant symbols:*

constant symbol **c**	$L(\mathbf{c})$
package	*Package*
truck	*Truck*
airplane	*Airplane*
*loc*1	*Location*
*loc*2	*Location*
*loc*3	*Location*

Definition of relations provides semantics for the relation declarations from $\mathcal{R}$. As it was said before, the relations describes the rigid properties in the planning domain. Once they are defined in the planning problem they *do not change* through the rest of the planning process.

Definition 14. Relation based on declaration

Let r be n-ary relation declaration $r \in \mathcal{R}$. With $\mathcal{C}$ defined and

$$args(r) = (a_1, \ldots, a_n)$$

we can denote n-ary *relation based on* r as α_r:

$$\alpha_r \subseteq \mathcal{C}\langle a_1 \rangle \times \ldots \times \mathcal{C}\langle a_n \rangle$$

[3]We suppose that $\forall c \in B : L(c) \in \mathcal{H}$

The set $\mathcal{D}$ is defined as:

$$\mathcal{D} = \{\ \alpha_r \mid r \in \mathcal{R}\}$$

Example 13. *There is only one relation in our domain. Since it is a ternary relation we will define it using a table with three columns. These columns will be populated with the constant symbols from $\mathcal{C}$. A line in the table will represent a triple of constant symbols that are in the relation. Those triples not in the relation are not included in the table.*

a_1	a_2	a_3
$loc1$	$loc2$	$truck$
$loc2$	$loc1$	$truck$
$loc2$	$loc3$	$airplane$
$loc3$	$loc2$	$airplane$

The relation presented in the table defines the route network depicted in figure 1.1.

Definition of state variables There is one big difference between the relations and the state variables. If we *declare a relation*, there is a straightforward way to *define it* once we know all the constant symbols from $\mathcal{C}$.

Unfortunatelly in the case of state variables things are more complicated. By *declaring a state variable* we are merely creating a template based on which *many state variables can be defined*.

There is possibility of combinatorial explosion if we declare a state variable with more than one argument as in the second part of example 10. This is due to the fact that the number of the state variables defined in the *planning problem* depends on the size of $\mathcal{C}$.

For example if there is a state variable $(stateVar, (object, object, object))$ declared in $\mathcal{V}$ then for a *planning problem* with $|\mathcal{C}\langle object\rangle| = n$, there would be n^3 state variables $stateVar$ in the set $\mathcal{F}$.

Definition 15. State variables

Let $v \in \mathcal{V}$ be n-ary state variable declaration. A set of all matching n-tuples that are compatible with $args(v) = (a_1, \ldots, a_n)$ is defined as:

$$M^v = \{\ (c_1, \ldots, c_n) \mid \forall i \in 1, \ldots, n : c_i \in \mathcal{C}\langle a_i\rangle\}$$

For each n-tuple of constants $(c_1, \ldots, c_n) \in M$ a state variable based on v denoted $\beta_v^{(c_1,\ldots,c_n)}$ is defined as:

$$\beta_v^{(c_1,\ldots,c_n)} \in \mathcal{C}[range(v)]$$

In the end we define a *set of all state variables yielded by v*:

$$\chi_v = \{\ \beta_v^{(c_1,\ldots,c_n)} \mid (c_1, \ldots, c_n) \in M^v\}$$

When it is appropriate we will use a shortcut $\beta_v^{\vec{c}}$ or just β instead of $\beta_v^{(c_1,\ldots,c_n)}$.

The set $\mathcal{F}$ is defined as:

$$\mathcal{F} = \bigcup_{v \in \mathcal{V}} \chi_v$$

Example 14. *There are three state variables in the set $\mathcal{F}$. One of them is yielded by the constant symbol* package *and the other two comes from* truck *and* airplane*:*

$\beta_v^{\vec{c}}$	$\in$	$\mathcal{C}[range(v)]$
$packagePos(package)$	$\in$	$\{loc1, loc2, loc3, truck, airplane\}$
$vehiclePos(truck)$	$\in$	$\{loc1, loc2, loc3\}$
$vehiclePos(airplane)$	$\in$	$\{loc1, loc2, loc3\}$

Initial state and goal **Definition 16. Assignment of value to state variable**

Let $\beta_v^{\vec{c}} \in \mathcal{F}$ be a state variable and $d \in \mathcal{C}[range(v)]$ be a constant. An *assignment* of d to $\beta_v^{\vec{c}}$ is defined as a pair:

$$(\beta_v^{\vec{c}}, d)$$

We will use following notation:

$$\beta_v^{\vec{c}} = d$$

Definition 17. System state

Any state of the system determined by the domain Σ and the planning problem $\mathcal{P}_\Sigma$ can be completely descibed by a set of *assignments* Q such that:

$$\forall \beta_v^{\vec{c}} \in \mathcal{F}\ \exists! d \in \mathcal{C}[range(v)] : (\beta_v^{\vec{c}} = d) \in Q$$

We will refer to any set Q that conforms to the previous condition as a *system state*.

The set $\mathcal{I}$ is a *system state*.

It is convenient to define a set of assignments that use a common state variable. Let $\beta_v^{\vec{c}} \in \mathcal{F}$ be the state variable. The set of assignments for $\beta_v^{\vec{c}}$ is defined as:

$$M_{\beta_v^c} = \{(\beta_v^c = d) | d \in \mathcal{C}[range(v)]\}$$

The set $\mathcal{G}$ is a non-empty set of *assignments* that uses only state variables from a set K:

$$K \subset \mathcal{F} : \mathcal{G} = \bigcup_{\beta \in K} M_\beta$$

It is possible for one state variable to have more than one assignment in the set $\mathcal{G}$. Therefore we need to describe which system states conforms to the set $\mathcal{G}$ in order to be labeled as *goal states*.

Definition 18. Goal state

Let Σ be a planning domain and $\mathcal{P}_\Sigma = (\mathcal{C}, \mathcal{D}, \mathcal{F}, \mathcal{I}, \mathcal{G})$ be a planning problem. Let Q be a system state. We say that Q is a *goal state* iff:

$$\forall \beta \in \mathcal{F}, M_\beta \subseteq \mathcal{G} : M_\beta \neq \emptyset \Rightarrow \exists a \in M_\beta : a \in Q$$

Example 15. *In this example we define the sets of assignments $\mathcal{I}$ and $\mathcal{G}$. The set $\mathcal{I}$ contains an assignment for every state variable from $\mathcal{F}$:*

$\mathcal{I}$		
$packagePos(package)$	$=$	$loc1$
$vehiclePos(truck)$	$=$	$loc2$
$vehiclePos(airplane)$	$=$	$loc3$

The set $\mathcal{G}$ contains only one assignment which represents our demand to transport the package *to the location* loc3*:*

$\mathcal{G}$		
$packagePos(package)$	$=$	$loc3$

4.2 Properties of new formalism

In this section we will first prove that our new formalism is powerful enough to describe a restricted transition system from chapter 1. We will also describe the main differences between our formalism and the representations from classical planning.

4.2.1 Expressivity

Theorem 1. *Any pair of planning domain $\Sigma = (\mathcal{H}, \mathcal{R}, \mathcal{V}, \mathcal{O})$ and planning problem $\mathcal{P}_\Sigma = (\mathcal{C}, \mathcal{D}, \mathcal{F}, \mathcal{I}, \mathcal{G})$ as defined in this chapter, can be reduced to a restricted transition system $\Sigma' = (S, A, \gamma)$ defined in section 1.2 of chapter 1.*

Proof. We will describe the construction of the set of states S, the set of actions A and we will define the state transition function γ. To achieve this we will use the planning domain Σ and the planning problem $\mathcal{P}_\Sigma$.

Set of states

$$S \subseteq \prod_{\beta_v^c \in \mathcal{F}} \mathcal{C}[range(v)]$$

A state s is a set of assignments that conforms to the condition in the definition 17.

Construction of action from operator: Let $o = (name(o), args(o), expr(o))$ be an operator from $\mathcal{O}$. Let $R_o \subseteq expr(o)$ be the set of all *relation expressions* used in the operator o:

$$R_o = \{(r, (x_1, \ldots, x_n)) \mid (r, (x_1, \ldots, x_n)) \in expr(o) \wedge r \in \mathcal{R}\}$$

A relation expression $E \in R_o$ constraints possible assignments of constant symbols to *object variables* from $args(o)$. The *constraint set by the relation expression $E = (r, (x_1, \ldots, x_n))$ is met* iff:

$$(\bar{x_1}, \ldots, \bar{x_n}) \in \alpha_r$$

Where α_r is the definition of r from $\mathcal{D}$.

Set of actions A contains every possible action created from every operator $o \in \mathcal{O}$ through any assignment of constant symbols from $\mathcal{C}$ to $args(o)$ that *meets all the constraints* set by R_o.

Definition of applicable action: Let $s \in S$ be a state of the system and $a \in A$ be an action $a = (name(a), args(a), expr(a))$. Let V_o be the set of all *transition expressions* (prevailing and non-prevailing) used in the operator o:

$$\begin{aligned} V_o \;=\; & \{(v,(x_1,\ldots,x_n,\mathbf{f},\mathbf{f})) \mid (v,(x_1,\ldots,x_n,\mathbf{f},\mathbf{f})) \in expr(o) \wedge v \in \mathcal{V}\} \bigcup \\ & \{(v,(x_1,\ldots,x_n,\mathbf{f},\mathbf{t})) \mid (v,(x_1,\ldots,x_n,\mathbf{f},\mathbf{t})) \in expr(o) \wedge v \in \mathcal{V}\} \end{aligned}$$

The action a is applicable in s iff:
For all transition expressions $E \in V_o$ with $\beta_v^{(\bar{x_1},\ldots,\bar{x_n})}$ defined in $\mathcal{F}$:

$$(\beta_v^{(\bar{x_1},\ldots,\bar{x_n})} = \bar{\mathbf{f}}) \in s$$

Transition function γ**:** If an action a is applicable to state s we define a new state $s^{'}$ as a modification of s:
For each non-prevailing transition $E = (v,(x_1,\ldots,x_n,\mathbf{f},\mathbf{t}))$ from $expr(a)$ we replace the assignment $(\beta_v^{(\bar{x_1},\ldots,\bar{x_n})} = \bar{\mathbf{f}})$ in s with $(\beta_v^{(\bar{x_1},\ldots,\bar{x_n})} = \bar{\mathbf{t}})$.
Transition function is then defined as: $\gamma(a,s) = s^{'}$ for every state s and action a applicable to s. □

Example 16. *In this example we will illustrate the key steps of the proof of the theorem 1. We will refer to the planning domain* Σ *from example 11 and the planning problem* $\mathcal{P}_\Sigma$ *described in the examples 12,13 and 14.*

The set of states S *contains 45 states in total. There are three state variables with domain sizes 3, 3 and 5 (example 14). Each state* $s \in S$ *can be represented by a set of three assignments such as:*

$$\begin{aligned} s \;=\; & \{(vehiclePos(truck) = loc2), \\ & (vehiclePos(airplane) = loc3), \\ & (packagePos(package) = loc1)\} \end{aligned}$$

The elements of the set of actions A *are based on opeators from* $\mathcal{O}$*, constant symbols from* $\mathcal{C}$ *and relation definitions from* $\mathcal{D}$*. We will describe the construction of all the actions derived from the operator* `moveVehicle` *(the operator* o_1 *from example 11).*

As a first step we will determine the set of relation expressions used in the operator. The set R_{o_1} *contain only one relation expression* `(reachable,(L1,L2,L3))`.

In the second step we write down the table of constants that represent all the valid assignments to the object variables of the operator. The following table lists all possible assignments for the object variables $L1, L2, V \in args(o_1)$ *which meet the constraint set by the relation expression* `(reachable,(L1,L2,V))`*:*

$L1$	$L2$	V
$loc1$	$loc2$	$truck$
$loc2$	$loc1$	$truck$
$loc2$	$loc3$	$airplane$
$loc3$	$loc2$	$airplane$

In the last step we will create instances of the operator. For each line of the previous table we get an action:

$moveVehicle(loc1, loc2, truck)$
$moveVehicle(loc2, loc1, truck)$
$moveVehicle(loc2, loc3, airplane)$
$moveVehicle(loc3, loc2, airplane)$

Similar procedure would be used to process all the operators from $\mathcal{O}$.

Aplicable actions. *If we consider the system state s described in the first paragraph of this example, we find only two instances of* `moveVehicle` *to be applicable in s:*

$moveVehicle(loc2, loc1, truck)$
$moveVehicle(loc3, loc2, airplane)$

The action `moveVehicle(loc1,loc2,truck)` *can not be applied because of the instance of* non-prevailing transition expression*:*

$$(vehiclePos, (truck, \mathbf{loc1}, \mathbf{loc2}))$$

which requires the assignment $(vehiclePos(truck) = loc1)$ to be in s.

Theorem 2. *Every restricted state transition system $\Sigma = (S, A, \gamma)$ can be expressed as a planning domain $\Sigma' = (\mathcal{H}, \mathcal{R}, \mathcal{V}, \mathcal{O})$ and planning problem $\mathcal{P}_{\Sigma'} = (\mathcal{C}, \mathcal{D}, \mathcal{F}, \mathcal{I}, \mathcal{G})$.*

Proof. We will describe construction of Σ' and $\mathcal{P}_{\Sigma'}$ from the restricted state transition system Σ.

The planning domain Σ'. We will describe construction of $\mathcal{H}$ and the sets $\mathcal{R}$,$\mathcal{V}$ and $\mathcal{O}$.

The set $\mathcal{H}$ will contain only the class $T_{\Sigma'}$:

$$T_{\Sigma'} = (object, \emptyset, \emptyset)$$

The set $\mathcal{R}$ will contain only one relation declaration r:

r	$=$	$(name(r), args(r))$
$name(v)$	$=$	$reachable$
$args(v)$	$=$	$(object, object)$

The set $\mathcal{V}$ will contain only one state variable declaration v:

v	$=$	$(name(v), args(v), range(v))$
$name(v)$	$=$	$stateVar$
$args(v)$	$=$	$()$
$range(v)$	$=$	$\{object\}$

The set $\mathcal{O}$ will contain only one operator o:

$definition$			$comment$
o	$=$	$(name(o), args(o), expr(o))$	
$name(o)$	$=$	$operator$	
$args(o)$	$=$	$\{S1, S2\}$	$\{object, object\}$
$expr(o)$	$=$	$\{(r, (S1, S2)),$	$reachable(S1, S2)$
		$(v, (S1, S2))\}$	$stateVar() : S1 \rightarrow S2$

The planning problem $\mathcal{P}_{\Sigma'}$.

The set of constants $\mathcal{C}$ will contain all states from S. Each of them will be given the root class `object`:

$$\forall s \in S \exists c_s \in \mathcal{C} : L(c_s) = T_{\Sigma'}$$

The set of relation definitions $\mathcal{D}$ will contain only the definition of the binary relation based on r:

$$\alpha_r = \{(c_{s_1}, c_{s_2}) \mid c_{s_1}, c_{s_2} \in \mathcal{C} : \exists a \in A : \gamma(s_1, a) = s_2\}$$

The set of state variables $\mathcal{F}$ will contain only one state variable based on v:

$$\beta_v \in \mathcal{C}$$

In the end we can choose any legal assignment to be the set $\mathcal{I}$ and any set of assignments to construct the set $\mathcal{G}$ in order to get a complete definition for $\mathcal{P}_{\Sigma'}$. □

4.2.2 Discussion of new formalism

Separation of planning domain and problem. The formalism presented in this chapter provides an alternative to the representations for classical planning mentioned in the first chapter. We have proved that it can be used to describe any restricted transition system Σ and that any restricted transition system Σ can be expressed in our formalism.

Though it is possible to describe the planning domain independently from planning problem in the classical representation, there was no suitable way to do this in the state variable representation which can be defined only for particular planning problem.

Our formalism, based on the state variable representation, makes this separate description possible. This possibility enables creation of knowledge modelling tools that can use state variables as a building stones instead of flexible relations. Such modelling tools can be used to create planning domains and problems that can be easily translated to representations such as FSA or directly to *SAS+*. Although there are some successfull approaches [13] and ongoing research [28] to provide conversion from the propositional representation to the state variable representation, there was no formalism which could be used for modelling abstract planning domains directly using only state variables for the description of change.

The approach to planning domain modeling using state variables can also provide a new point of view on some planning domains. We will present an example in the last chapter.

Extensibility. Usage of *terms* in our formalism enables possibilities for further extension over the frame of classical planning. Apart from the constant symbols we did not defined any other kinds of terms yet but we have left the definition open. We can easily extend our formalism with numeric expressions for example which is the main requirement for planning with time and resources.

Inheritance. If we have for example the class `vehicle` it is convenient to define specialized classes like `truck` or `airplane`. They have common abilities like carrying packages but they move differently. The requirement for `:typing` if we are to use terminology of the *PDDL* language is common in planning.

From the three represetations for the classical planning only the state variable representation uses the concept of classes. However the classes in the state variable representation are defined only as sets of objects. It is not clear from this definition what should be the relations between the classes like.

In our formalism we define class hierarchy explicitly as a part of the planning domain.

Problem independent constants. The concept of constants that are defined in the planning domain and are accessible in all related planning problems is included in the language *PDDL*. We have found this concept convenient for representation of some key features in some planning domains.

5. Implementation

In this chapter we would like to introduce a proof-of-concept implementation of a modeling tool based on the formalism proposed in the previous chapter. This tool is not intended to provide user friendly interface.

The developement of presented program was driven mainly by my previous experience with the developement of *VIZ* [34]. Based on this experience I focused on the implementation of the main domain design steps rather than their visual appearance. In this way I have deliberately avoided the task of user interface design. The task is impossible without proper specification of target user group and requirements that this target group might have.

Resulting GUI is strictly minimalistic and can be used by anyone who is familiar with the state variable representation described in 1.2.3. The program was developed with a possibility of further extension with more sophisticated visualization tools in mind. Its main purpose is to act as a core component ready for further developement and experiments.

5.1 Program architecture

The program is developed in *Java* programming language in order to be portable. All XML processing routines are using `JAXB` framework [21].

5.1.1 Input and output.

The state of all internal data structures can be exported/imported to/from XML. Analogous to *itSIMPLE*, XML is suitably used as an intermediate language (figure 5.1). Different structure of XML files is used for the planning domain and for planning problems. The program can convert XML descriptions of domain and problems to *PDDL*. Supported subset of *PDDL* can be determined by the version 3.1 and the requirements:
`:strips :typing :equality :object-fluents`

Conversion of planning domain. Here we will demonstrate conversion of the planning domain Σ from the example 11 to *PDDL* as implemented in the convertor used by Vizzard. The convertor is actually translating XML files defined by the XML scheme definitions from the attachement E.

Class hierarchy $\mathcal{H}$ will result in the following *PDDL* code:

```
(:types
        Vehicle Package Location - object
        Truck Airplane - Vehicle )
(:constants
        foo - Location )
```

We have added dummy location `foo` to demonstrate translation of problem independent constants.

Relation `reachable` from the set $\mathcal{R}$ will be encoded as:

```
(:predicates
        (reachable ?A0 - Location ?A1 - Location ?A2 - Vehicle) )
```

State variable `packagePos` from the set $\mathcal{V}$ will result in:

```
(:functions
(packagePos ?A0 - Package) - (either Vehicle Location))
```

Operators `moveVehicle` and `loadPackage` from the set $\mathcal{O}$ takes the form:

```
(:action moveVehicle
:parameters ( ?L1 - Location ?L2 - Location ?V - Vehicle)
:precondition (and (reachable  ?L1 ?L2 ?V )
                   (= (vehiclePos ?V) ?L1) )
:effect (assign (vehiclePos ?V) ?L2) )

(:action loadPackage
:parameters ( ?X3 - Vehicle ?V - Vehicle
              ?P - Package ?L - Location)
:precondition (and (= (vehiclePos ?V) ?L)
                   (= (packagePos ?P) ?L) )
:effect (assign (packagePos ?P) ?X3) )
```

Planning problem described in the examples 12, 13, 14 and 15 will be translated as:

```
(define (problem example)
(:domain logistic)
(:requirements :strips :typing :equality :object-fluents)
(:objects
 truck - Truck
 loc1 loc3 loc2 - Location
 package - Package
 airplane - Airplane
)
(:init (reachable loc1 loc2 truck )
       (reachable loc2 loc1 truck )
       (reachable loc2 loc3 airplane )
       (reachable loc3 loc2 airplane )
       (= (packagePos package) loc1)
       (= (vehiclePos truck) loc2)
       (= (vehiclePos airplane) loc3) )
(:goal (and (= (packagePos package) loc3))) )
```

5.1.2 Internal data structures.

The scheme for the main data structures used in the program is displayed in the figure 5.2. These structures represent modular components which are used to store the planning domain and problem data. Their functions reflect the

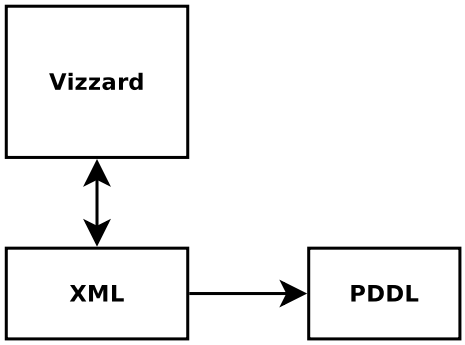


Figure 5.1: Data formats

formalism defined in the chapter 4. The two main structures (`dmDomain` and `dmTask`) can be exported to XML files.

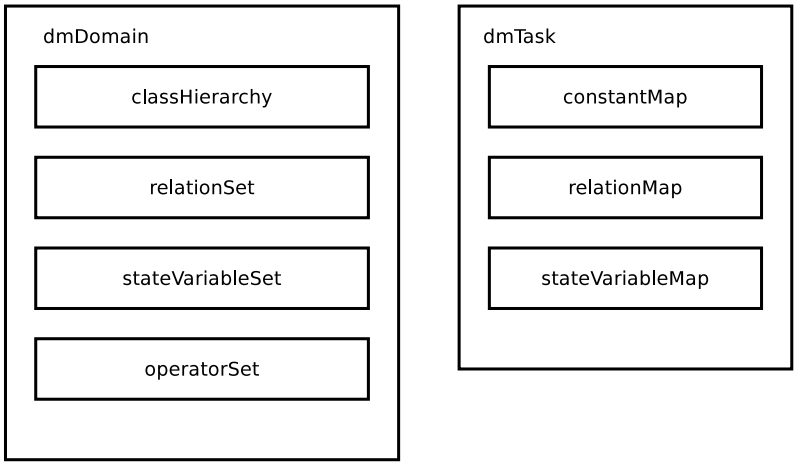


Figure 5.2: Data formats

The structure called `dmDomain` manage the data for a planning domain Σ. The inner structures of `dmDomain` are:

- `classHierarchy` - is a tree structure used to store the class hierarchy $\mathcal{H}$
- `relationSet` - is a table with relation declarations ($\mathcal{R}$)
- `stateVariableSet` - is a table with state variable declarations ($\mathcal{V}$)
- `operatorSet` - is a list of operators ($\mathcal{O}$)

Each of these structures provide an interface with methods for read-write access to the modelled data. These methods can be used by future visualization tools.

Every `dmTask` structure is defined with a reference to `dmDomain` and it manage the data for a planning problem $\mathcal{P}_\Sigma$:

- `constantMap` - is a map that contains a set of problem dependent constant symbols ($\mathcal{C}$)
- `relationMap` - is a map that contains a relation definition table for every relation declaration ($\mathcal{D}$)

- `stateVariableMap` - is a map that contains a set of state variables for every state variable declaration ($\mathcal{F}$). An initial value stored with every state variable is sufficient to describe an initial state $\mathcal{I}$. To describe a set of goal states $\mathcal{G}$ we may store a set of target values as well.

Each of these structures also provide an interface to enable further extension with more sophisticated visualization tools.

5.2 Program demonstration

In this section we will demonstrate the modeling capabilities of Vizzard by recreating a classical benchmark domain from the International Planning Competition [17].

Detailed description of program GUI used for this task can be found in the sections *Declarations* and *Operators* of the attachement D. To create a particular problem we will refer to the section *Tasks* from the same attachement.

5.2.1 Depots domain

For this demonstration we have chosen a well known benchmark domain called *Depots*. It is a classical planning domain used in the third International Planning Competition [17]. The domain is a combination of two other planning domains - *Blockworld* and *Logistic*.

Domain description. The domain contains `places`,`trucks`, `hoists` and `packages` stored on `pallets` in piles. A package can be loaded on a truck in order to be transported from one place to another. Loading is done with hoist which can unstack the package from the top of pile. We can see an example situation on the figure 5.3.

5.2.2 Knowledge modelling with Vizzard

The design of the planning domain can be informally divided in two phases:

1. identification of the most important entities in the domain and establishing of some relations

2. description of changes that may occur in the domain

Declarations. Using Vizzard, we can describe the sets $\mathcal{H}$,$\mathcal{R}$ and $\mathcal{V}$ in the *Declarations* panel to acomplish the first phase. We will refer to section D.2 in the attachement D for the details concerning the GUI functions. The panel with the *Depots* domain declarations in displayed in the figure 5.4.

Operators. Now we can describe the operators. There are five actions possible:

1. `Lift` - a package is lifted by a hoist from the top of the pile

2. `Drop` - a package is droped by a hoist on the top of the pile

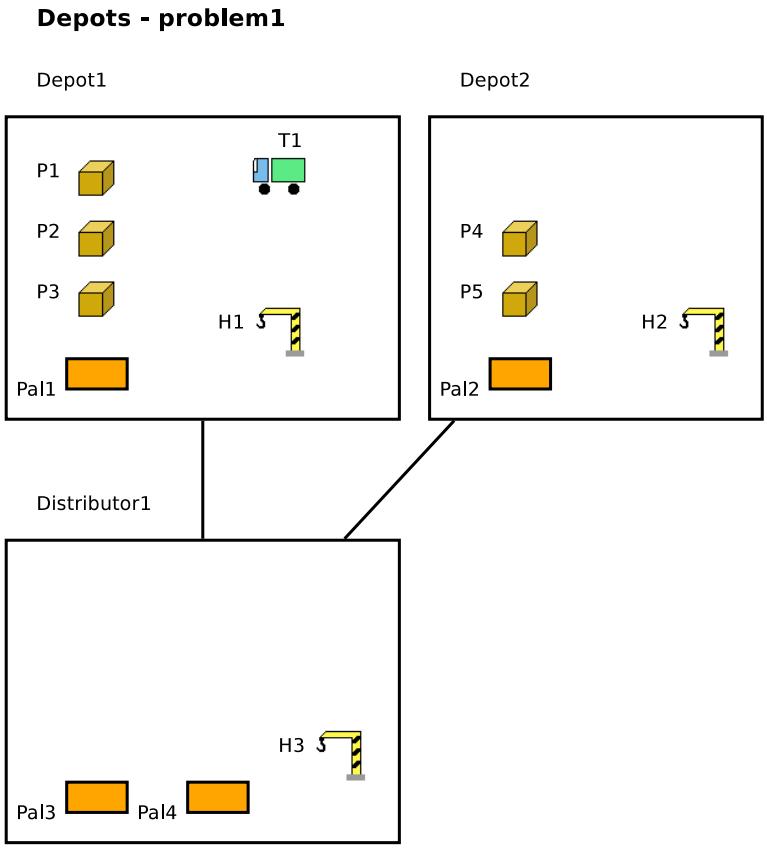


Figure 5.3: Example situation

3. `Unload` - package is lifted by a hoist from a truck
4. `Load` - package is droped by a hoist on a truck
5. `Drive` - a truck moves from one place to another

The figure 5.5 shows the operator `Load` presented in Vizzard as an example. The details concerning the GUI can be found in the section D.3 in the attachement D.

1. The *non-prevailing* expression on the first row describe the change of `Crate` position (the state variable **cratePos**). The `Crate` was previously lifted by the `Hoist` and it will be loaded to `Truck`.
2. The second row present a condition (expressed with a *prevailing transition*) that the `Truck` has to be at `Place` when loaded.
3. The third row contain a *relation expression* that binds `Hoist` on the same `Place` where the `Truck` is standing.
4. The *non-prevailing* expression on the fourth row change property of the `Truck` using task independent constant symbols in the operator.
5. The *non-prevailing* expression on the fifth row is describing change of `Hoist`'s property.

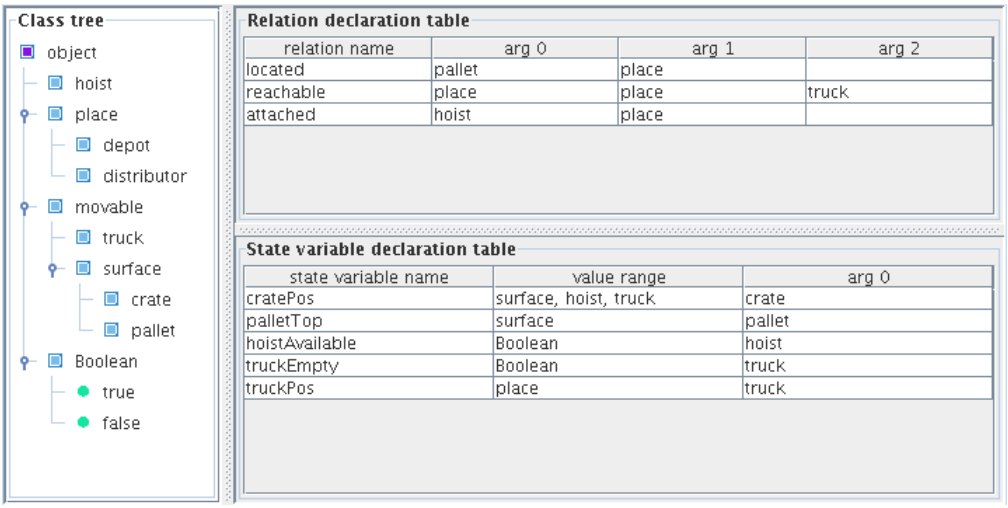


Figure 5.4: Depots domain declarations

cratePos	C [crate]	:	H [hoist]	->	T [truck]
truckPos	T [truck]	==	P [place]		
attached	H [hoist]	P [place]			
truckEmpty	T [truck]	:	true [Boolean]	->	false [Boolean]
hoistAvailable	H [hoist]	:	false [Boolean]	->	true [Boolean]

Figure 5.5: Operator Load in Vizzard

Resulting domain expressed in the *PDDL* language can be found in the attachement B. The domain definition is using the requirement `:object-fluents` which is only available in the version `3.1` of *PDDL*. Current implementation does not provide conversion to propositional representation which would have to be used in order to retain compatibility with earlier versions of the language. The propositional version of the domain from the planning competition is included for the purpose of comparsion and can be found in the attachement A.

Planning problem. When we are content with the domain description we can set out to define some planning problem using the panel labeled *Tasks*. With the details for GUI handling we will refer to the last section of the attachement D.

We will use the example situation from the figure 5.3 in our problem definition. The finished definition of the relation `located` is on the figure 5.6. Finally the description of the initial values of the state variables based on `cratePos` together with goal specification can be seen on the figure 5.7. Now we have been through design of planning domain as well as planning problem with Vizzard.

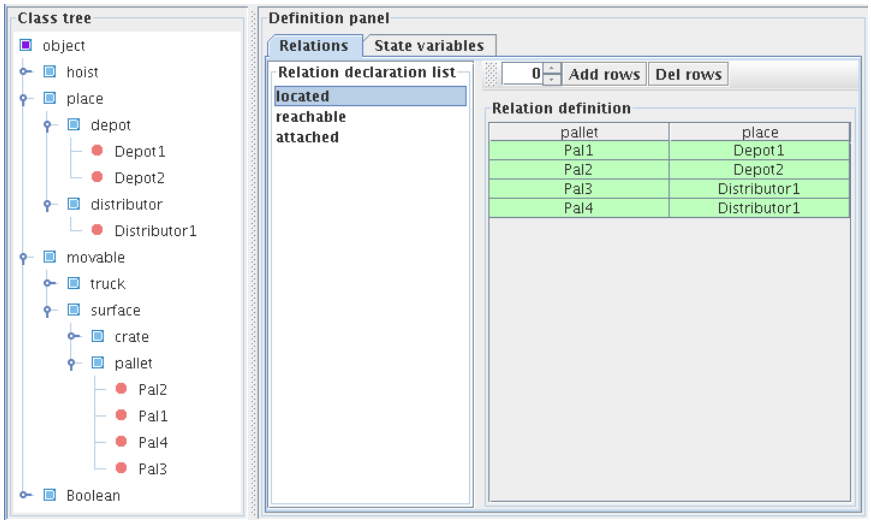


Figure 5.6: Definition of relation *located*

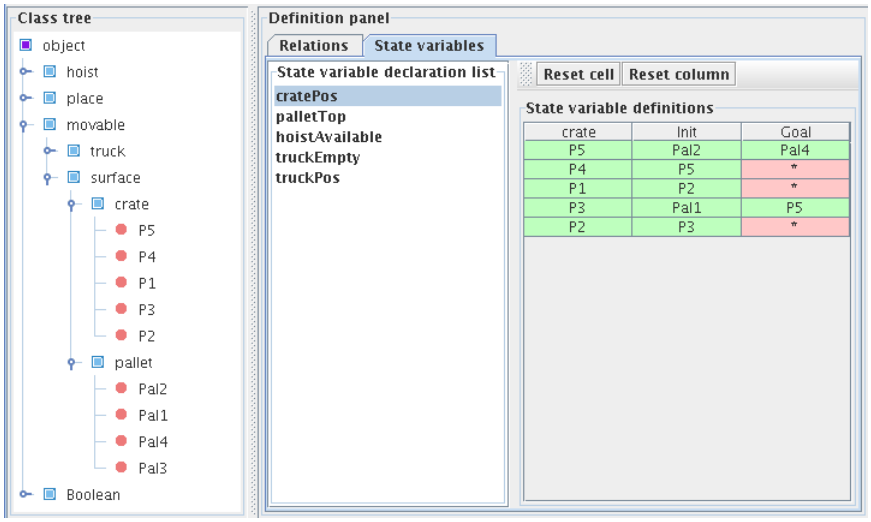


Figure 5.7: Defintions for state variable *cratePos*

5.3 Developement enviroment

For the developement of the program Vizzard we have used the enviroment with the following characteristics:

- Programming IDE: `NetBeans IDE 7.0.1`
- Java version: `1.6.0_24; Java HotSpot(TM) Client VM`
- Operating system: `Linux version 2.6.32-5-686`

6. Evaluation

6.1 Towards efficient problem modelling

The planning domain known as *towers of hanoi* is used in [9] to explain motivation for functional STRIPS formalism which is very close to our formalism. A problem from the domain features a finite number of discs of different sizes that can be moved from the top of one peg to the top of another peg. In the same time we can not place bigger disc on top of a smaller disc.

It is argued in [9] that the number of ground actions depends on the number of arguments in operators. When using functional STRIPS, we can model the operators more efficiently than with the classical propositional STRIPS. With more efficient model we can reduce the number of arguments which results in smaller number of ground actions. As a result we can expect better planner performance.

The two main differences between the functional STRIPS and our formalism are that our formalism does not explicitly define *preconditions* and *effects* in operator specification and that the functional STRIPS formalism does not describe the abstraction of planning domain without reference to problem dependent information[1].

With Vizzard we can create domain description comparable to the functional STRIPS. The formulation of a problem for three towers as presented in [9] is in the figure 6.2 whereas the description of the same problem created with Vizzard can be seen in figures 6.1 (Declarations), 6.3 (Operator move) and 6.4, 6.5 (Task[2]).

When using platform provided by Vizzard as a starting point we can implement various features that enables modelling of efficient planning domains resembling those based on the functional STRIPS.

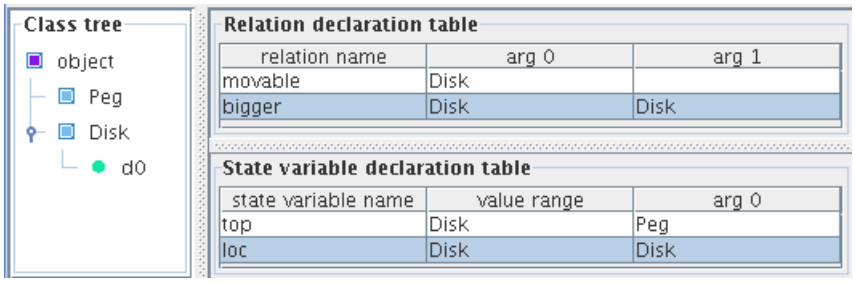


Figure 6.1: Hanoi towers - Declarations

[1] i.e. description of "the world" in functional STRIPS contains problem specific constants

[2] We have altered screenshots in order to fit in two figures.

Domains: Peg: p1, p2, p3 ;the pegs
Disk: d1, d2, d3, d4 ;the disks
Disk∗: Disk, d0 ;the disks and a dummy bottom disk 0

Fluents: top: Peg → Disk∗ ;denotes top disk in peg
loc: Disk → Disk∗ ;denotes disc below given disc
size: Disk∗ → Integer ;represents disc size

Action: move(P1,P2: Peg) ;moves between pegs
Prec: top(P1) ≠ d0, size(top(P1)) < size(top(P2))
Post: top(P1) := loc(top(P1)), loc(top(P1)) := top(P2), top(P2) := top(P1)

Init: loc(d1) = d0, loc(d2) = d1, loc(d3) = d2, loc(d4) = d3,
top(p1) = d4, top(p2) = d0, top(p3) = d0,
size(d0) = 4, size(d1) = 3, size(d2) = 2, size(d3) = 1, size(d4) = 0

Goal: loc(d1) = d0 , loc(d2) = d1, loc(d3) = d2, loc(d4) = d3,
top(p3) = d4

Figure 6.2: Hanoi towers - functional STRIPS

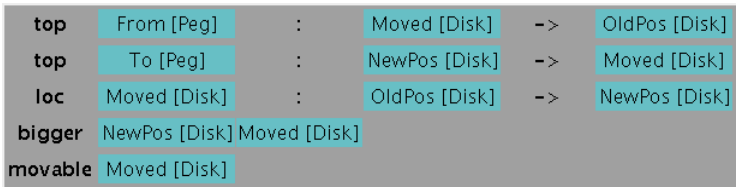


Figure 6.3: Hanoi towers - operator move

6.2 New point of view

One of the advantages of Vizzard is the possibility of direct conversion into a formalism close to the *SAS+* representation. The great deal of inspiration for this thesis came from the question if the finite state automata, as defined for example in [28], can be used for knowledge modelling in some way.

Though we did not managed to answer this question yet we have gained a different perspective while attempting to do so.

This section is included in order to demonstrate a new point of view on planning domains which is based on our formalism from the chapter 4.

We will now describe *Depots* domain with generalized finite state automata. We do not attempt to provide any formal definition of GFSA. The term "generalized" is used in order to differ these automata from FSA. Let us explain the main differences:

- There is neither initial state nor goal states in GFSA.
- The state in GFSA can be either constant symbol or set of constant symbols.
- The transition function in GFSA is based on the operators from our formalism.

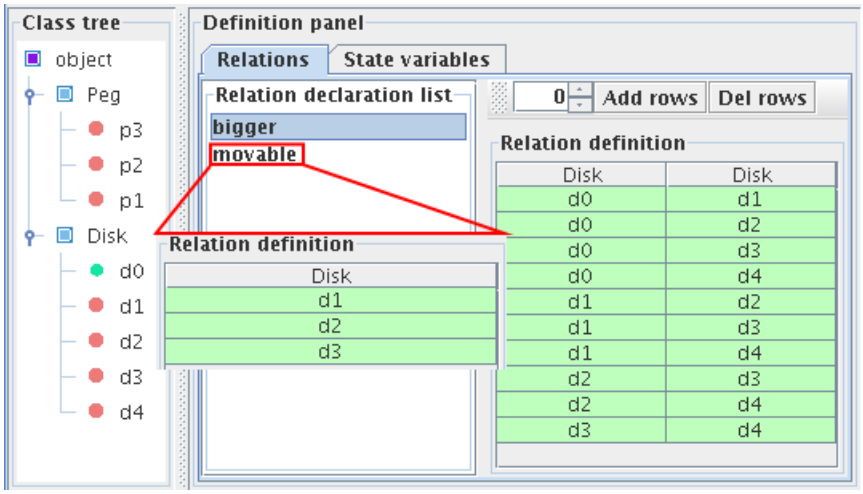


Figure 6.4: Hanoi towers - relation definitions

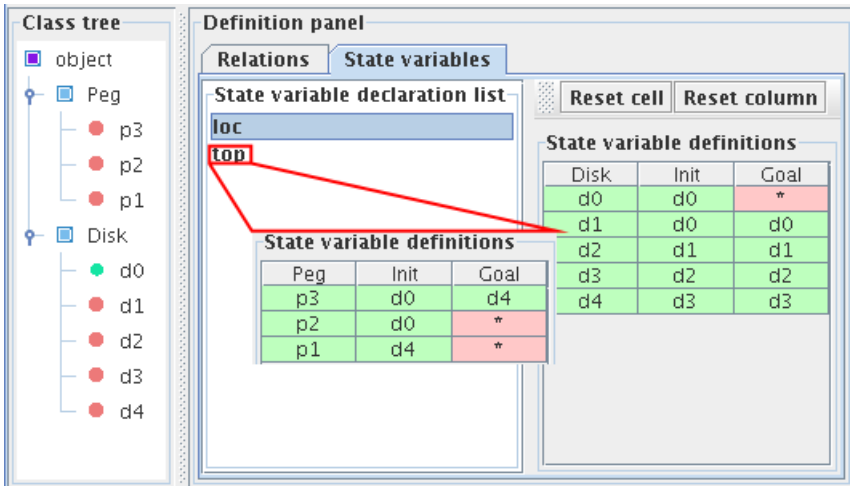


Figure 6.5: Hanoi towers - state variables

One GFSA also corresponds to one state variable declaration from our formalism and it can refer to other GFSA's through the transition function. Therefore, in non-trivial case, it makes only sense to talk about a set of GFSA's connected with a set of operators.

6.2.1 Generalized finite state automata

The diagrams in the attachement C show the generalized finite state automata for each state variable declaration from the *Depots* domain.

The edges in the diagrams are labeled with the names of the operators from the *Depots* domain. The definitions of operators can be seen in figure 6.7.

Now we will describe the aforementioned diagrams in more detail using the figure 6.6 as example for reference. The circular nodes refer to the classes from the domain. Each class determines sets of objects. The rectangular nodes mark objects represented with the problem independent constants.

The semantics of the edges in the diagrams is explained in following two points:

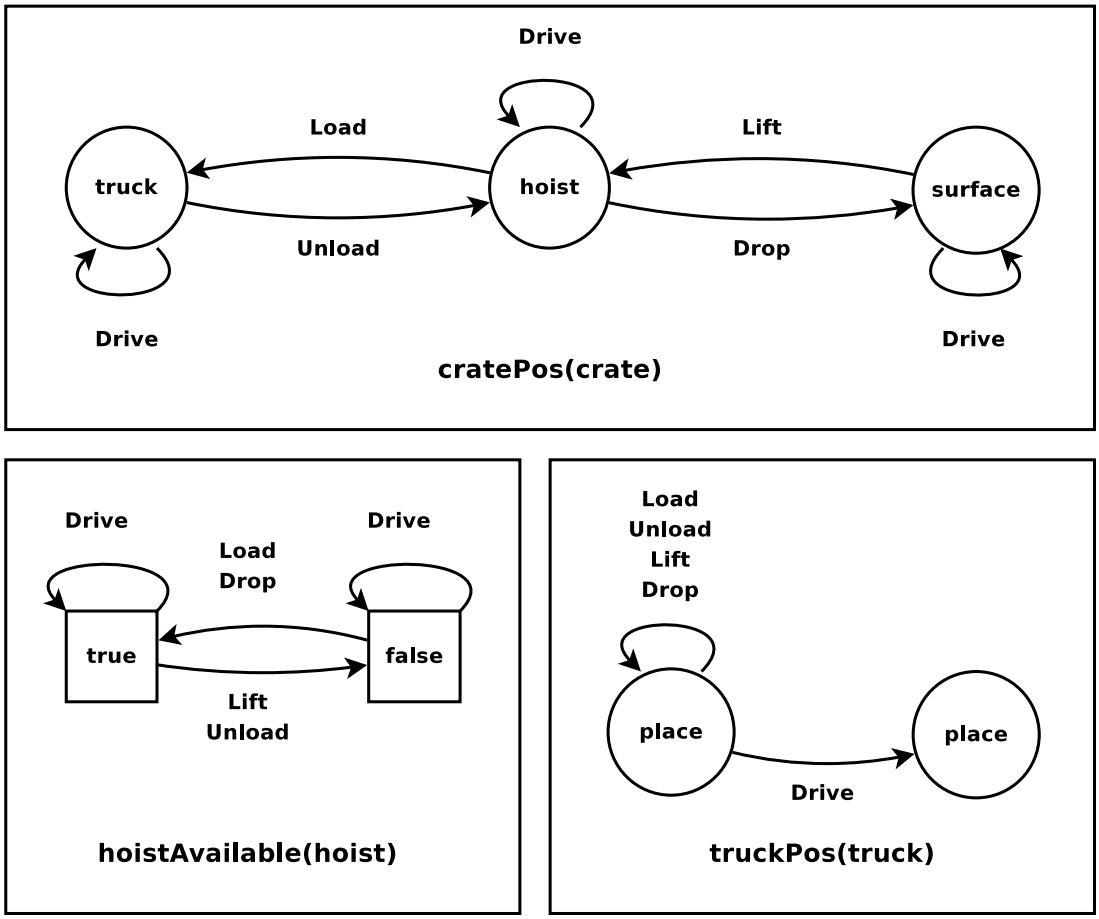


Figure 6.6: Diarams of GFSA for Depots domain

- If a loop edge is labeled with the operator O it means that either no *non-prevailing transition* is included among the expressions of O (e.g. operator **Drive** in the diagram for **cratePos** or **hoistAvailable**) or a *prevailing transition* that requires the state variable to retain its value is included (e.g. operators **Load**, **Unload**, **Lift**, **Drop** in the diagram for **truckPos**).

- A non-loop edge labeled with the operator O denote that there is a *non-prevailing transition* included among the expressions of O that change the value of the state variable from an object **A** to a different object **B**. We distinguish two cases:

  1. The class of **A** is equal with the class of **B** (e.g. operator **Drive** in the diagram for **truckPos**)
  2. The class of **A** is different from **B** (e.g. operators **Load**, **Unload**, **Lift** and **Drop** in the diagram for **cratePos** and **hoistAvailable**)

$Lift(C-crate, S-surface, H-hoist, P-pallet, L-place)$
$cratePos(C) : S \rightarrow H$ $palletTop(P) : C \rightarrow S$ $hoistAvailable(H) : true \rightarrow false$ $attached(H, L)$ $located(P, L)$

$Drop(C-crate, S-surface, H-hoist, P-pallet, L-place)$
$cratePos(C) : H \rightarrow S$ $palletTop(P) : S \rightarrow C$ $hoistAvailable(H) : false \rightarrow true$ $attached(H, L)$ $located(P, L)$

$Load(C-crate, H-hoist, T-truck, L-place)$
$cratePos(C) : H \rightarrow T$ $truckEmpty(T) : true \rightarrow false$ $hoistAvailable(H) : false \rightarrow true$ $truckPos(T) = L$ $attached(H, L)$

$Unload(C-crate, H-hoist, T-truck, L-place)$
$cratePos(C) : T \rightarrow H$ $truckEmpty(T) : false \rightarrow true$ $hoistAvailable(H) : true \rightarrow false$ $truckPos(T) = L$ $attached(H, L)$

$Drive(T-truck, L1-place, L2-place)$
$truckPos(T) : L1 \rightarrow L2$ $reachable(L1, L2, T)$

Figure 6.7: Depots domain - operators

7. Conclusion

This thesis dealt with knowledge engineering for Automated Planning. Throughout the text we have identified the main challenges in the general-purpose knowledge modelling for the classical planning and pointed out the benefits of the state variable representation.

We have proposed a new formalism based on the state variable representation which enable strict separation of the planning domain abstraction from planning problem. This feature is important in the knowledge modelling if we want to achieve knowledge reusability (i.e. one planning domain can be used for multiple planning problems).

Another important feature of our formalism is that the definition of planning operator, which is usually the most complicated part of planning domain models, is basically reduced to one set of expressions with interconnected variables. The preconditions and effects are not explicitly distinguished. We have demonstrated that the two types of expressions (*relation expressions* and *state variable expressions*) are sufficient for modeling of various operators in classical planning domains.

We have implemented the knowledge modelling tool called Vizzard which is based on the new formalism. It was demonstrated that this tool can be successfully used to model classical planning domains and problems. The representation entered by the user in program GUI can be exported into the standard language for planning domain modelling called *PDDL* which is a widespread input format for planners.

Although our tool does not provide fullfledged user interface it can be used by researchers in planning community who are already familiar with the concept of state variables.

It was argued in the last chapter that the representation based on the formalism used in the program is comparable to the efficient domain models based on the functional STRIPS and that it provides a new point of view on the knowledge modelling.

7.1 Future work

Program Vizzard presented in this thesis provide base platform for KE modelling tools. It can be extended with various tools for visualization in order to make the design comprehensible for non-experts in planning.

For example the design of a comprehensible visualization tool for planning operators can be quite challenging. The platform provided by Vizzard can be used to test and evaluate such a tool.

The program can be also extended to allow modelling of planning domains with numeric resources or time. These extensions are essential for modelling of "real world" problems.

To provide a better connection to planners it is possible to implement conversion to earlier versions of *PDDL* that does not support the `:object-fluents` requirement. Although this is a step back from a more compact domain representation it would be convenient to provide the conversion in order to retain

backward compatibility with existing planners.

Methods for the import of existing planning domains encoded in *PDDL* can be also implemented in order to enable modification of existing planning domains and to explore possibilities of conversion from the propositional representation to the representation based on the state variables.

Bibliography

[1] Various authors. Pddl online resources. http://ipc.informatik.uni-freiburg.de/PddlResources, November 2011.

[2] Christer Bäckström and Bernhard Nebel. Complexity results for sas+ planning. *Computational Intelligence*, 11:625–656, 1995.

[3] Chavier Barreiro. Nddl example: The planetary rover. http://code.google.com/p/europa-pso/wiki/ExampleRover, November 2011.

[4] Sara Bernardini and David E. Smith. Translating pddl2.2 into a constraint-based variable/value language. In *Proceedings of the Workshop on Knowledge Engineering for Planning and Scheduling (KEPS), 18th International Conference on Automated Planning and Scheduling (ICAPS)*, 2008.

[5] Yixin Chen, Ruoyun Huang, and Weixiong Zhang. Fast planning by search in domain transition graph. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI 2008)*, pages 886–891, 2008.

[6] Stefan Edelkamp and Jörg Hoffmann. Pddl 2.2: The language for the classical part of the 4th international planning competition. Technical report, Fachbereich Informatik and Institut für Informatik, 2004.

[7] Susana Fernández and Daniel Borrajo. Solving clustered oversubscription problems for planning e-courses. In *Proceedings of SPARK, Scheduling and Planning Applications woRKshop, ICAPS*, pages 36–43, September 2009.

[8] Maria Fox and Derek Long. Pddl2.1: An extension to pddl for expressing temporal planning domains. *J. Artif. Intell. Res. (JAIR)*, 20:61–124, 2003.

[9] Héctor Geffner. *Logic-based artificial intelligence*, chapter Functional strips: a more flexible language for planning and problem solving, pages 187–209. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[10] Alfonso Gerevini and Derek Long. Preferences and soft constraints in pddl3. In A. Gerevini and D. Long, editors, *Proceedings of ICAPS workshop on Planning with Preferences and Soft Constraints*, pages 46–53, 2006.

[11] Arturo González-Ferrer, Juan Fernández-Olivares, and Luis Castillo. Jabbah: A java application framework for the translation between business process models and htn. In *Proceedings of the 3rd ICKEPS Competition (collocated with ICAPS Conference)*, pages 28–37, 2009.

[12] Malte Helmert. The fast downward planning system. *J. Artif. Intell. Res. (JAIR)*, 26:191–246, 2006.

[13] Malte Helmert. Concise finite-domain representations for pddl planning tasks. *Artificial Intelligence*, 173(5-6):503–535, 2009.

[14] Sarah L. Hickmott and Sebastian Sardiña. Optimality properties of planning via petri net unfolding: A formal analysis. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 170–177. AAAI Press, September 2009.

[15] Henry Kautz. Deconstructing planning as satisfiability. In *Proceedings of the 21st national conference on Artificial intelligence - Volume 2*, pages 1524–1526. AAAI Press, 2006.

[16] Philippe Laborie and Malik Ghallab. Planning with sharable resource constraints. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 1643–1651, 1995.

[17] Derek Long, Maria Fox, David E. Smith, Drew McDermot, Fahiem Bacchus, and Hector Geffner. International planning competition 3. http://ipc.icaps-conference.org, 2002.

[18] Drew McDermott. The 1998 ai planning systems competition. *AI Magazine*, 21(2):35–55, 2000.

[19] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL - the planning domain definition language. Technical report, CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.

[20] Dana Nau, Malik Ghallab, and Paolo Traverso. *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[21] Group of contributors. Java architecture for xml binding. http://jaxb.java.net, November 2011.

[22] Sudhakar Y. Reddy, Jeremy D. Frank, Michael J. Iatauro, Matthew E. Boyce, Elif Kürklü, Mitchell Ai-Chang, and Ari K. Jónsson. Planning solar array operations on the international space station. *ACM Trans. Intell. Syst. Technol.*, 2:41:1–41:24, July 2011.

[23] R Sherwood, A Govindjee, D Yan, G Rabideau, S Chien, and A Fukunaga. Using aspen to automate eo-1 activity planning. In *Proceedings of the IEEE Aerospace Conference, Snowmass, CO*, March 1998.

[24] Ron M. Simpson, Diane E. Kitchin, and Thomas L. McCluskey. Planning domain definition using gipo. *Knowl. Eng. Rev.*, 22:117–134, June 2007.

[25] Ron M. Simpson, Thomas L. McCluskey, Donghong Liu, and Diane E. Kitchin. Knowledge representation in planning: A pddl to ocl_h translation. In *International Syposium on Methodologies for Intelligent Systems (ISMIS)*, pages 610–618, 2000.

[26] David E. Smith, Jeremy Frank, and William Cushing. The anml language. In *Proceedings of Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*, 2008.

[27] Hans-Peter Störr. Planning in the fluent calculus using binary decision diagrams. *AI Magazine*, 22(3):103–106, 2001.

[28] Daniel Toropila and Roman Barták. Using finite-state automata to model and solve planning problems. In *Proceedings of the Eleventh AI*IA Symposium on Artificial Intelligence*, pages 183–189, University of Brescia, 2010.

[29] Tiago S. Vaquero. itsimple project. http://code.google.com/p/itsimple, November 2011.

[30] Tiago S. Vaquero, Victor Romero, Flavio Tonidandel, and José R. Silva. itsimple 2.0: An integrated tool for designing planning domains. In *Proceedings of International Conference on Automated Planning and Schedulin (ICAPS)*, pages 336–343, 2007.

[31] Tiago S. Vaquero, Fernando Sette, José R. Silva, and J. Christopher Beck. Planning and scheduling of crude oil distribution in a petroleum plant. In *Proceedings of SPARK, Scheduling and Planning Applications woRKshop, ICAPS*, pages 99–106, 2009.

[32] Tiago S. Vaquero, José R. Silva, and J. Christopher Beck. A brief review of tools and methods for knowledge engineering for planning & scheduling. In *Proceedings of the Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*, pages 7–14, 2011.

[33] Tiago Stegun Vaquero, Flavio Tonidandel, and José Reinaldo Silva. The itsimple tool for modeling planning domains. In *Proceedings of the First International Competition on Knowledge Engineering for AI Planning and Scheduling (ICKEPS)*. AAAI Press, 2005.

[34] Jindřich Vodrážka and Lukáš Chrpa. Visual design of planning domains. In *Proceedings of Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*, pages 68–69, 2010.

List of Definitions

Attachments

A. Depots domain from IPC 3

Example of planning PDDL domain from International Planning Competition.

```
(define (domain Depot)
(:requirements :typing)
(:types place locatable - object
        depot distributor - place
        truck hoist surface - locatable
        pallet crate - surface)

(:predicates (at ?x - locatable ?y - place)
             (on ?x - crate ?y - surface)
             (in ?x - crate ?y - truck)
             (lifting ?x - hoist ?y - crate)
             (available ?x - hoist)
             (clear ?x - surface))

(:action Drive
:parameters (?x - truck ?y - place ?z - place)
:precondition (and (at ?x ?y))
:effect (and (not (at ?x ?y)) (at ?x ?z)))

(:action Lift
:parameters (?x - hoist ?y - crate ?z - surface ?p - place)
:precondition (and (at ?x ?p) (available ?x)
                   (at ?y ?p) (on ?y ?z) (clear ?y))
:effect (and (not (at ?y ?p)) (lifting ?x ?y)
             (not (clear ?y)) (not (available ?x))
             (clear ?z) (not (on ?y ?z))))

(:action Drop
:parameters (?x - hoist ?y - crate ?z - surface ?p - place)
:precondition (and (at ?x ?p) (at ?z ?p)
                   (clear ?z) (lifting ?x ?y))
:effect (and (available ?x) (not (lifting ?x ?y))
             (at ?y ?p) (not (clear ?z)) (clear ?y) (on ?y ?z)))

(:action Load
:parameters (?x - hoist ?y - crate ?z - truck ?p - place)
:precondition (and (at ?x ?p) (at ?z ?p) (lifting ?x ?y))
:effect (and (not (lifting ?x ?y)) (in ?y ?z) (available ?x)))

(:action Unload
:parameters (?x - hoist ?y - crate ?z - truck ?p - place)
:precondition (and (at ?x ?p) (at ?z ?p)
                   (available ?x) (in ?y ?z))
:effect (and (not (in ?y ?z))
             (not (available ?x)) (lifting ?x ?y)))
)
```

B. Depots domain generated by Vizzard

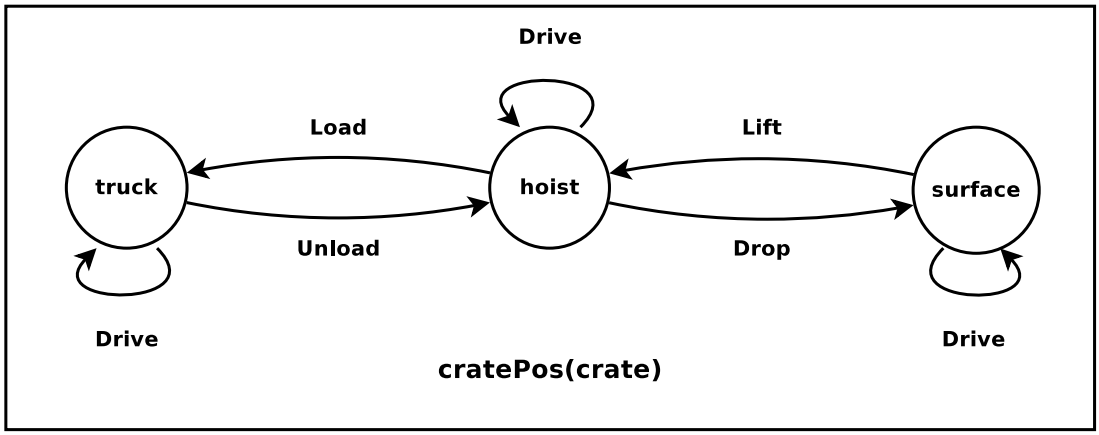
Example of planning domain generated from the model created in the program Vizzard.

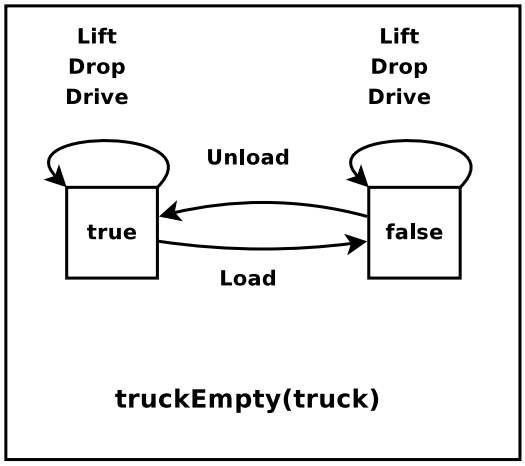
```
(define (domain Depots)
(:requirements :strips :typing :equality :object-fluents)
(:types hoist place movable boolean - object
        depot distributor - place
        truck surface - movable
        crate pallet - surface)
(:constants
        true false - boolean)
(:predicates
        (located ?A0 - pallet ?A1 - place)
        (reachable ?A0 - place ?A1 - place ?A2 - truck)
        (attached ?A0 - hoist ?A1 - place))
(:functions
        (cratePos ?A0 - crate) - (either truck hoist surface)
        (truckEmpty ?A0 - truck) - boolean
        (palletTop ?A0 - pallet) - surface
        (hoistAvailable ?A0 - hoist) - boolean
        (truckPos ?A0 - truck) - place)
(:action Drop
:parameters ( ?PL - place ?H - hoist ?CT - crate
              ?S - surface ?PA - pallet)
:precondition (and (= (palletTop ?PA) ?S)
                   (= (cratePos ?CT) ?H)
                   (= (hoistAvailable ?H) false)
                   (attached  ?H ?PL )
                   (located  ?PA ?PL ))
:effect (and (assign (palletTop ?PA) ?CT)
             (assign (cratePos ?CT) ?S)
             (assign (hoistAvailable ?H) true)))
(:action Lift
:parameters ( ?PL - place ?H - hoist ?S - surface
              ?CT - crate ?PA - pallet)
:precondition (and (= (palletTop ?PA) ?CT)
                   (= (cratePos ?CT) ?S)
                   (= (hoistAvailable ?H) true)
                   (attached  ?H ?PL )
                   (located  ?PA ?PL ))
:effect (and (assign (palletTop ?PA) ?S)
             (assign (cratePos ?CT) ?H)
             (assign (hoistAvailable ?H) false)))
```

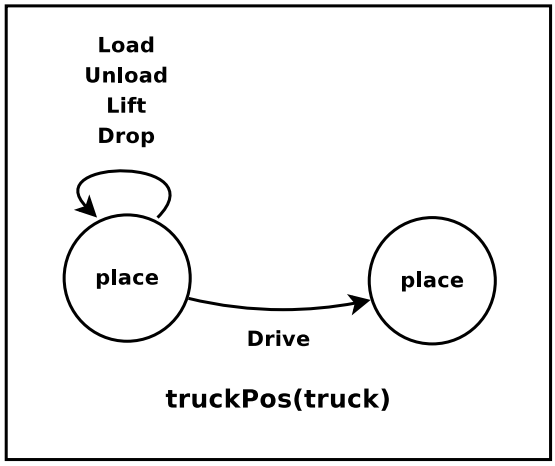
```
(:action Load
:parameters ( ?PL - place ?H - hoist
              ?T - truck ?CT - crate)
:precondition (and (= (truckEmpty ?T) true)
                   (= (hoistAvailable ?H) false)
                   (= (cratePos ?CT) ?H)
                   (= (truckPos ?T) ?PL)
                   (attached  ?H ?PL ))
:effect (and (assign (truckEmpty ?T) false)
             (assign (hoistAvailable ?H) true)
             (assign (cratePos ?CT) ?T)))
(:action Unload
:parameters ( ?PL - place ?H - hoist
              ?T - truck ?CT - crate)
:precondition (and (= (truckEmpty ?T) false)
                   (= (hoistAvailable ?H) true)
                   (= (cratePos ?CT) ?T)
                   (= (truckPos ?T) ?PL)
                   (attached  ?H ?PL ))
:effect (and (assign (truckEmpty ?T) true)
             (assign (hoistAvailable ?H) false)
             (assign (cratePos ?CT) ?H)))
(:action Drive
:parameters ( ?T - truck ?To - place ?From - place)
:precondition (and (= (truckPos ?T) ?From)
              (reachable  ?From ?To ?T ))
:effect (assign (truckPos ?T) ?To)))
```

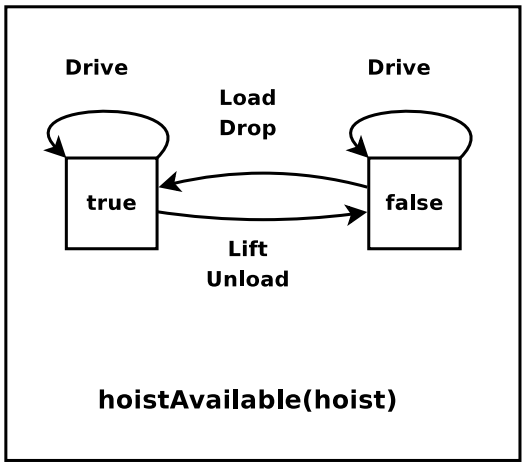
C. Generalized finite state automata

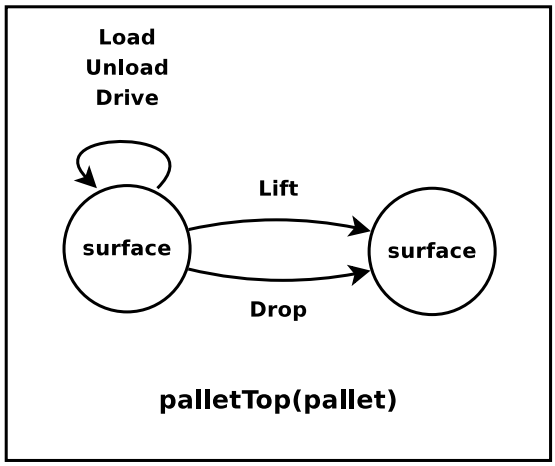
Diagrams for generalized state automata based on the example domain *Depots*.











D. Vizzard - user manual

In this attachement we will describe the program GUI. This manual is divided into four sections. Each of them refers to the specific GUI component.

1. *Main menu*
2. *Declarations*
3. *Operators*
4. *Tasks*

D.1 Main menu

The main menu contains standard options *File*, *Edit* and *Help*. Their detailed description can be found in this section.

File. Through this option we can manipulate XML files, created by *Vizzard*, that contain the domain description. Relevant menu options for this task are:

- *New* - start description of new planning domain
- *Open* - load a file with previously saved planning domain
- *Save* - save changes to existing file
- *Save as* - save the current domain snapshot to a new file

The option labeled *Export PDDL* converts current XML representation of current planning domain to PDDL format. The resulting file is limited by PDDL requirements `:typing`, `:strips`, `:equality` and `:object-fluents`. We will refer to [1] (PDDL 3.1) for details.

If we select the option *Exit*, the program asks for confirmation and terminates.

Edit. There is only one option in this section. It is labeled *Domain properties* and we can use it to change the name of the domain.

D.2 Declarations

There are basicly three tasks that can be achieved through *Declarations* panel (figure D.1). We will adress all of them in this section.

Class hierarchy declaration is the first task to do. The classes are declared in the left-most segment called *Class tree* (figure D.2). One class is always defined in the *Class tree*. Its name is `object` and it is the root of the class hierarchy.

New classes may be inserted in order to describe the class hierarchy. There is also possibility to define problem independent constants for the domain.

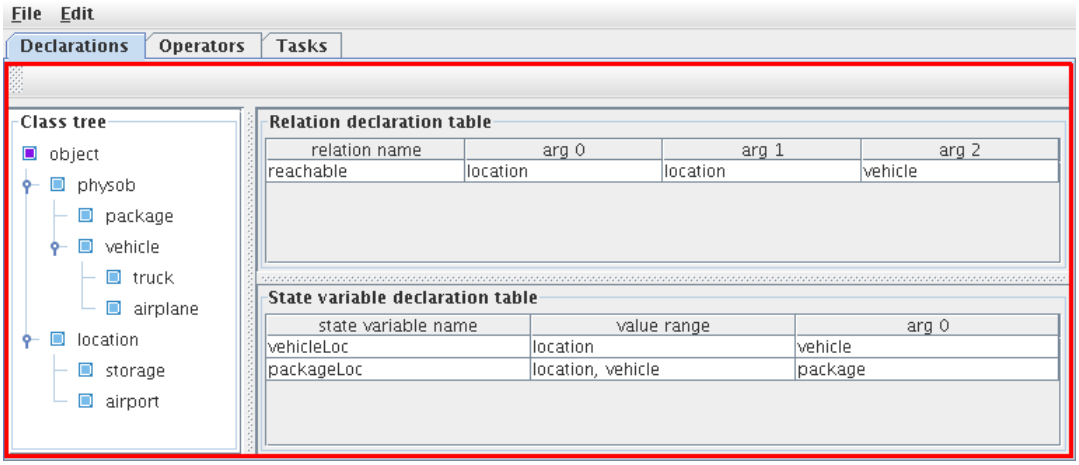


Figure D.1: Declarations panel

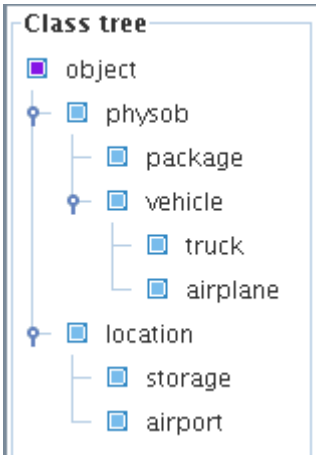


Figure D.2: Class tree

- To **add a new subclass**:

  1. select a parent class
  2. click the right mouse button anywhere in the *Class tree*
  3. select the option *Add class* in the popup-menu summoned by the click
  4. enter the subclass name in the dialog (the name has to be unique)

  An alternative is to select a parent class and press the `Insert` key. New subclass will be added with automatically generated unique name.

- To **add a new constant** to class we proceed in similar way as when adding a new subclass. Only in the pop-up menu we need to select the option *Add constant*.

- To **delete a node** from the class hiererchy:

  1. select the node we want to delete
  2. summon the pop-up menu in the *Class tree*
  3. select the option *Delete node*

Both class and constant nodes can be deleted this way. An alternative is to select the target node and press `Delete` key.

- To **rename a node**:

  1. double-click the name of the renamed node
  2. enter a new name in simple dialog

Both class and constant nodes can be renamed this way.

Relation declaration is the second task. It can be skipped if we do not want to include any rigid relations to the domain we are designing. Relations are listed in the top-most table on the right side, called *Relation declaration table* (figure D.3).

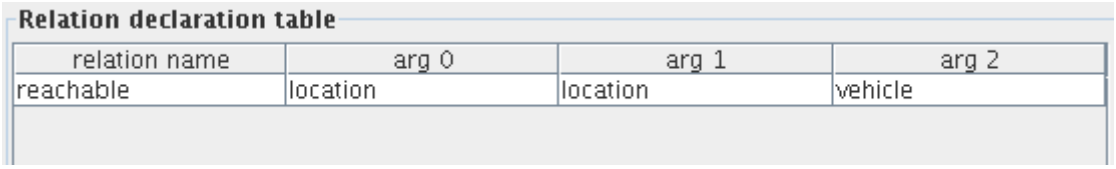


Figure D.3: Relation declaration table

A row in the table represents one rigid relation declaration. Relation name is written in the first column and the relation arguments are listed begining in the second column. In this table we define classes for relation arguments and the number of arguments is defined as a side effect.

List of possible actions for the *Relation declaration table*:

- To **declare new relation**:

  1. right click anywhere in the *Relation declaration table*
  2. select the option *Add new relation* in the popup-menu summoned by the click

  New relation will be added with automatically generated unique name.

- To **rename** a relation we just double-click on the relation name and edit the name in the table.

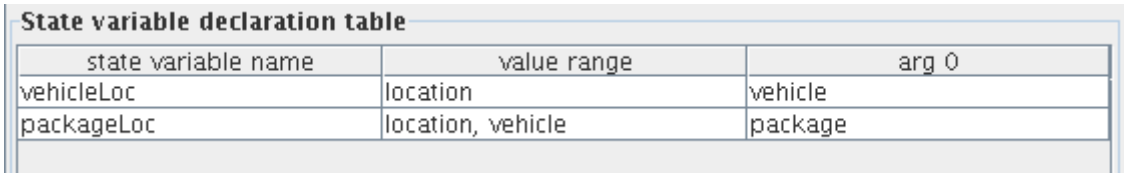
- To **declare argument class** of any relation we can drag a class from *Class tree* and drop it in the relation line to the appropriate column.

- Number of columns in the *Relation declaration table* can be manipulated by the options *Add argument column* and *Del argument column* in the pop-up menu that is summoned by clicking the right mouse button in the *Relation declaration table*.

- To **delete relation**:

  1. left click on the relation row in order to select it
  2. right click anywhere in the *Relation declaration table* to summon a pop-up menu
  3. select the option *Delete relation* from the menu

State variable declaration is the third task. Every domain aspect that is subject to any changes shall be described here. The state variables are listed in the bottom-most table on the right side, called *State variable declaration table* (figure D.4).


State variable declaration table

state variable name	value range	arg 0
vehicleLoc	location	vehicle
packageLoc	location, vehicle	package

Figure D.4: State variable declaration table

One row in the *State variable declaration table* represents declaration of a state variable set[1]. For simplicity we will say *state variable* instead of *set of state variables*.

The state variable name is displayed in the first column of the table row. The second column lists classes that altogether define the range of the state variable. The state variable arguments are defined in similar way as in the case of the rigid relations, only beginning in the third column.

The list of possible actions for *State variable declaration table*:

- To **declare new state variable**:

  1. right click anywhere in the *State variable declaration table*
  2. select the option *Add state variable* in the pop-up menu summoned by the click

  A new row in the *State variable declaration table* will be created with automatically generated name in the first column.

- To **rename** a state variable we can just double-click in the first column and edit the name in the table.

- Argument classes are defined in the same way as in the *Relation declaration table*.

- The **value range** of any state variable can be defined by dragging and droping classes from the *Class tree* into the second column of appropriate row.

- To **clear value range** of the state variable:

  1. left click on the state variable row in order to select it
  2. right click anywhere in the *State variable declaration table* to summon a pop-up menu
  3. select the option *Clear state variable range* from the menu

- To **delete state variable**:

[1]See definition 15.

1. left click on the state variable row in order to select it
2. right click anywhere in the *State variable declaration table* to summon a pop-up menu
3. select the option *Delete state variable* from the menu

D.3 Operators

The panel labeled *Operators* (figure D.5) is used to describe all kinds of changes which may occur in our planning domain. These changes are described through the operators. All the operators are listed on the left (*Operator list*). If we select an operator in the *Operator list*, details are displayed on the right in a table (*Expression table*). In this section we will descibe available actions in the *Operator list* and the *Expression table*.

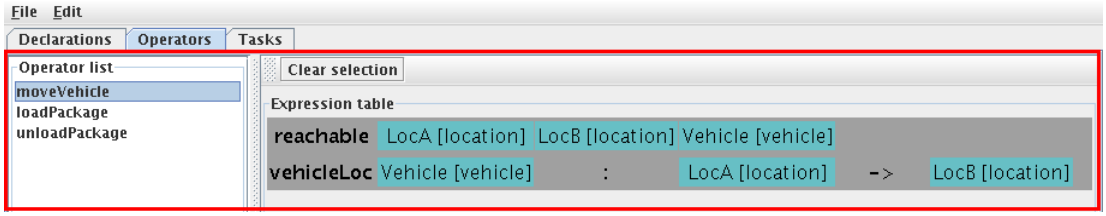


Figure D.5: Operators panel

Operator list is used for the operator management. Apart from selecting the operators for detailed display in the *expression table* we can add, remove and copy the operators using a pop-up menu raised by the right click anywhere in the *Operator list*. Possible menu items are:

- **Create operator** - create an empty operator. A unique name must be entered in the dialog.
- **Copy operator** - create a copy of the operator currently selected. A unique name for the copy must be entered in dialog.
- **Delete operator** - delete the operator currently selected.

Expression table is a component that is meant to list and edit the expressions of the operator selected in the *operator list*. Each row in the expression table represents one *expression*. There are three kinds of possible expressions:

- **relation expression** - a condition based on a rigid relation
- **prevailing transition expression** - a condition based on a state variable
- **non-prevailing transition expression** - change of a state variable

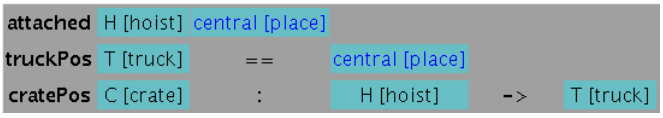


Figure D.6: Expressions

Example featuring all possible expressions is displayed in figure D.6. On the first row we can see the *relation expression*, the *prevailing transition expression* is on the second row and we can see the *non-prevailing transition expression* on the third row.

Here is the list of actions for the expression management:

- To **add a relation expression**:
  1. right click anywhere in the *expression table* to summon a pop-up menu
  2. in the menu choose the option *Insert expression* → *Add relations* to display a *Select relations dialog*
  3. The dialog features two lists (*Available relations* and *Selected relations*) and three buttons (*Include*, *Exclude* and *Done*). (see figure D.7) The buttons *Include* and *Exclude* are used to manipulate the items in the two lists (i.e. we can include and exclude selected relations). All the relations present in the *Selected relations* list are used to create new **relation expressions** in the operator as soon as the button *Done* is clicked.
- To **add a transition expression**:
  1. right click anywhere in the *expression table* to summon a pop-up menu
  2. in the menu choose the option *Insert expression* → *Add transitions* to display *Select transitions dialog*
  3. The dialog features three lists (*State variable declarations*, *State variable transitions* and *Selected transitions*), three buttons (*Include*, *Exclude* and *Done*) and one checkbox labeled *prevailing*. (see figure D.8)

     In this dialog we can select transitions that we want to use for the transition expressions creation.

     We start by selecting an item in the *State variable declarations* list. As soon as we select a state variable declaration, the *State variable transitions* list fills with the available transitions for the state variable declaration.

     Using the checkbox we can controll which kind of transitions is currently presented in the *State variable transitions* list.

     We can manipulate the content of the *Selected transitions* list with the buttons *Include* and *Exclude* (i.e. we can include and exclude selected transitions).

     As soon as the button *Done* is clicked, the items from the *Selected transitions list* are included as **transition expressions** in the *Expression table*.

4. To **delete** an expression from the *expression table* we need to select corresponding line (selected line is highlighted) in the table and summon the pop-up menu by clicking the right mouse button in the table. In the menu we select the option *Delete expression*.

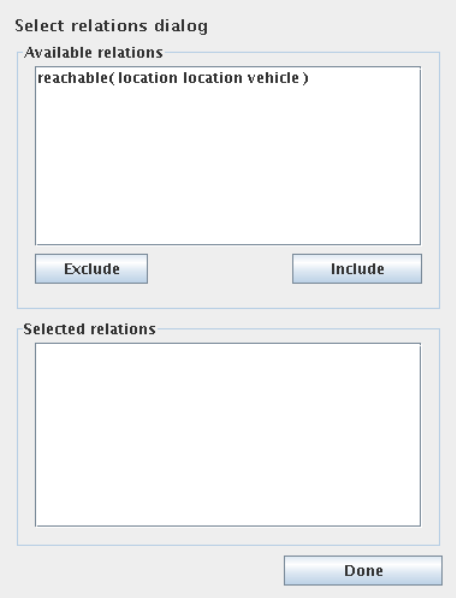


Figure D.7: Select relations dialog

Every expression contains *slots* that can be connected together by variables or assigned a constant symbol. Slot cells in the *expression table* are distiguished with different color. In each slot we can read the name of assigned variable or constant with their class in square brackets. Blue color of the text is indicating a constant. We can see an example in the figure D.6.

By clicking on the slot with the left mouse button we either *select* it (if it was not selected) or *deselect* it (if it was already selected). The figure D.9 shows two expressions - all slots are *selected* in the first expression and no slot is selected in the second one.

We can clear the selection (i.e. deselect all selected slots) by clicking the button *Clear selection* in the toolbar above the *expression table*.

List of actions for the selected slots:

- To **bind the selected slots** by common object variable:

  1. right click anywhere in the *expression table* to summon a pop-up menu
  2. select the option *Set value* → *Variable*
  3. enter the variable name in the dialog

  The entered name has to be unique in the current operator.

- To **assign a constant symbol** to selected slots:

  1. right click anywhere in the *expression table* to summon a pop-up menu
  2. select the option *Set value* → *Constant*

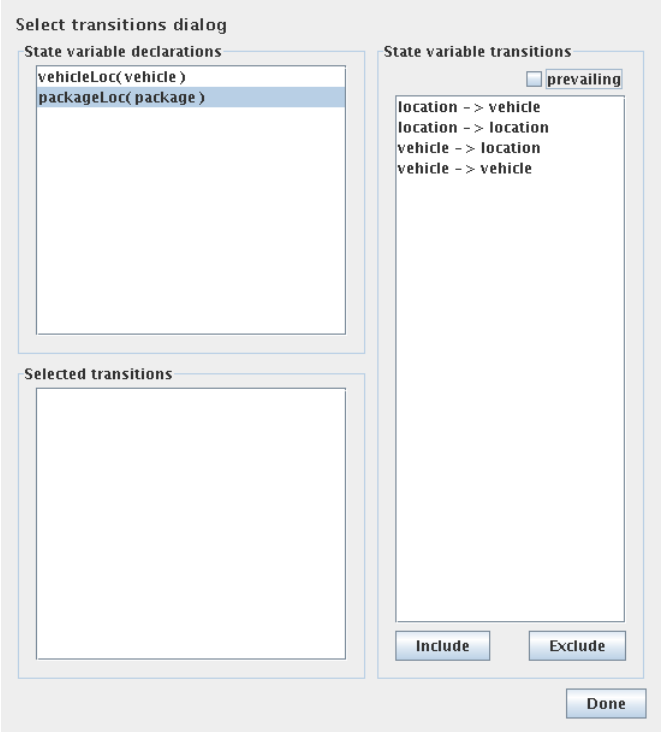


Figure D.8: Select transitions dialog

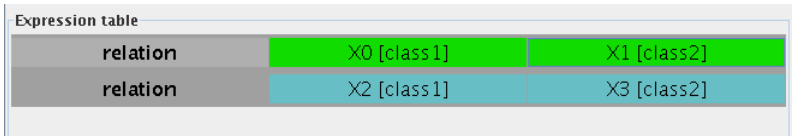


Figure D.9: Slot selection example

3. select a constant from the dialog offered

Some constant, which is compatible with all selected slots, has to be defined in the domain in order to perform this action.

D.4 Tasks

This panel (figure D.10) is meant to provide support for modelling of planning problems. We suppose that a planning domain was previously defined using the *Declarations* panel and the *Operators* panel.

The panel has three main components:

- *task management toolbar* - task handling procedures
- *class tree* - management of constant symbols
- *definition panel* - definition of rigid relations and specification of initial state together with goal conditions

In this section we will describe their purpose and associated actions in more detail.

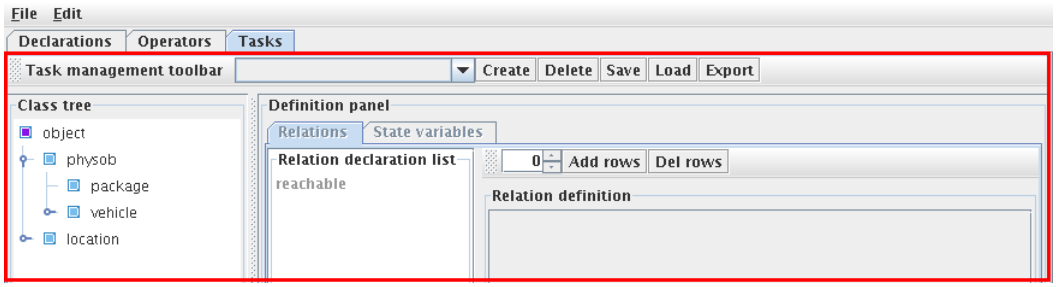


Figure D.10: Tasks panel

Task management toolbar features one combo box and buttons *Create*, *Delete*, *Save*, *Load* and *Export*.

The combobox displays name of the task that is being edited at the moment. We will refer to such task as *current task*. We can change the *current task* by selecting another item in the combo box.

The functionality provided by the buttons is descibed below:

- *Create* button is used to create a new task. Clicking this button will summon a dialog asking for the new task name. When the name is confirmed we are able to select the associated task in the combo box.
- *Delete* button is used to remove the current task. Its name is removed from the combo box.
- *Save* button is used to write the current task on disk in XML format. A file select dialog is presented after clicking the button.
- *Load* button is used to read a task which was previously saved using the *Save* button.
- *Export* button is used to write the current task on disk in *PDDL* format. A file select dialog is presented after clicking the button.

The class tree placed on the left side of the *Tasks* panel contains the exact copy of the class hierarchy defined in the *Declarations* panel. However in this case the classes can not be manipulated. The *Class tree* in the *Tasks* panel is used to define *task dependent* constant symbols.

Actions for manipulation of the *task dependent* constants:

- To **define a new constant**:
  1. select some class in the *Class tree*
  2. right click anywhere in the *Class tree* to summon a pop-up menu
  3. in the menu select the option *Add constant*
  4. enter the name for the constant in the offered dialog

  A new node, with the name entered in the last step, will appear in the tree under the class that was selected in the first step. Another alternative is to select a class and press the `Insert` key. New constant will be defined with automatically generated name.

- To **rename** constant just doubleclick on the target constant and enter new name in the offered dialog.
- To **delete constant**:
  1. select target node in the *Class tree*
  2. press `Delete` key

Only problem dependent constants can be removed this way. They can be distinguished from the problem independent constants by color.

Definition panel placed on the right side od the *Tasks* panel (see figure D.11), features two tabs labeled *Relations* and *State variables*. Both tabs have similar structure - a list on the left side and a table on the right side.

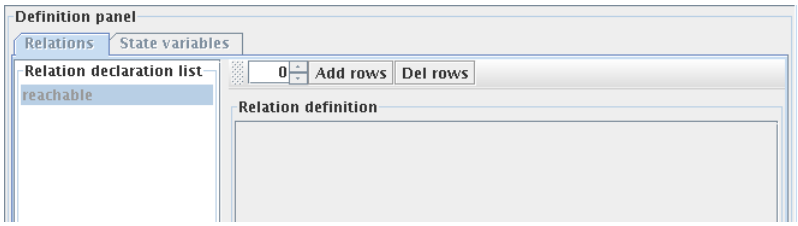


Figure D.11: Definition panel

In the case of the *Relations* tab, the list contains the names of the relations declared in the domain, whereas in the case of the *State variables* tab, the list is filled with the names of the state variable declarations. In both cases we can use these lists to select a particular relation or a state variable name.

The tables in both tabs can be edited by draging constant symbols from *Class tree* or using toolbars above them. Most of the cells in the tables initially contains the symbol "*" to indicate undefined value (those cells have red color). The value of a cell can be defined by draging a constant symbol from the *Class tree* and droping it to the cell (the cells with defined values are green). The example situation is displayed in figure D.12. Both tables are described in more detail in following paragraphs.

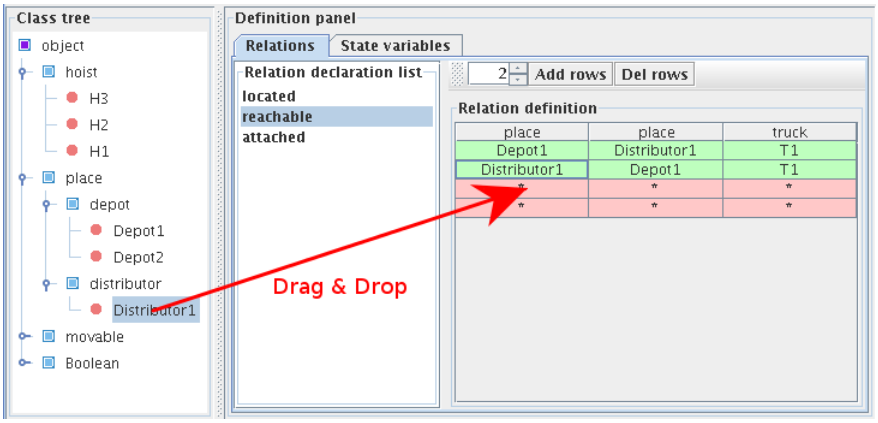


Figure D.12: Definition table cells

Relation definition table displays definition of the relation that was previously selected in the *Relation list*. The number of columns in the table corresponds with the number of arguments of the relation. Columns are labeled with the names of corresponding argument classes.

Initially there are no rows in the table. We can add new rows using the toolbar above the table. Each row in the table defines the tuple of constants in the relation. We can define such tuples using the *drag & drop* approach suggested in the figure D.12 to fill empty rows or to change values in existing rows.

The toolbar can be used to:

- **add specified number of empty rows** - use the spin box to set the number and then click the button *Add rows*
- **delete rows** - while holding the `Ctrl` key, click the left mouse button on rows you wish to delete. When done, click the button *Del rows*.

State variable definitions table displays the list of all the state variables defined in the current task that are based on the state variable declaration, currently selected in the *State variable list*. The columns in the table are labeled with the names of the state variable argument classes begining in the first column.

The two last columns are labeled *Init* and *Goal*. The cells int the *Init* column contains initial values for the state variables. They are used to specify an initial state of the system.

The cells in the *Goal* column can contain sets of constants. The conditions for a goal state are restricted through these sets.

The *drag & drop* approach suggested in the figure D.12 can be used in similar way as in the case of the *Relation definition table* to fill the cells in both *Init* and *Goal* columns.

The main difference is that we can not add or delete any rows[2] and we can change values only in the two last columns[3].

The toolbar above the *State variable definitions table* can be used to:

- **reset selected column** - click to either of *Init* or *Goal* column to select it and then click on the *Reset column* button. All the cells in the column will be will be reseted to the default value "*****".
- **reset single cell** - click on the toggle button *Reset cell*. As long as the button is toggled you can reset any single cell to the default value "*****" if you click on it.

[2]There is a finite number of state variables based on the number of defined constants in the problem.

[3]Each state variable is uniquely defined by its arguments.

E. XSD scheme files

Schemes for XML files used by the program Vizzard.

E.1 Domain scheme

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:jxb="http://java.sun.com/xml/ns/jaxb"
            jxb:version="2.0">

    <!— root element —>
    <xsd:element name="domain" type="domainType"/>

    <!— root element definition —>
    <xsd:complexType name="domainType">
        <xsd:sequence>
            <xsd:element name="properties"
                type="domainProperties" minOccurs="1" maxOccurs="1"/>
            <xsd:element name="classes"
                type="classTreeType" minOccurs="1" maxOccurs="1"/>
            <xsd:element name="relations"
                type="relationListType" minOccurs="1" maxOccurs="1"/>
            <xsd:element name="stateVariables"
                type="stateVariableListType" minOccurs="1" maxOccurs="1"/>
            <xsd:element name="operators"
                type="operatorListType" minOccurs="1" maxOccurs="1"/>
        </xsd:sequence>
    </xsd:complexType>

    <!— domain properties START —>
    <xsd:complexType name="domainProperties">
        <xsd:sequence>
            <xsd:element name="name" type="xsd:string"/>
            <xsd:element name="requirements" type="pddlRequirementsListType
                "/>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="pddlRequirementsListType">
        <xsd:sequence>
            <xsd:element name="requirement" type="xsd:string" maxOccurs="
                unbounded"/>
        </xsd:sequence>
    </xsd:complexType>
    <!— domain properties END —>

    <!— classes definition START —>
    <xsd:complexType name="classTreeType">
```

```
    <xsd:sequence>
        <xsd:element name="node" type="nodeType"/>
    </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="nodeType">
    <xsd:sequence>
        <xsd:element name="type" type="xsd:string"/>
        <xsd:element name="children" type="childrenListType"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string"/>
</xsd:complexType>

<xsd:complexType name="childrenListType">
    <xsd:sequence>
       <xsd:element name="node" type="nodeType"
       maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>
<!— classes definition END —>

<!— relations definition START —>
<xsd:complexType name="relationListType">
    <xsd:sequence>
    <xsd:element name="relation" type="relationType"
       maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="relationType">
    <xsd:sequence>
        <xsd:element name="argument" type="argumentType"
        maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string"/>
</xsd:complexType>

<xsd:complexType name="argumentType">
    <xsd:attribute name="number" type="xsd:int"/>
    <xsd:attribute name="class" type="xsd:string"/>
</xsd:complexType>
<!— relations definition END —>

<!— stateVariables definition START —>
<xsd:complexType name="stateVariableListType">
    <xsd:sequence>
    <xsd:element name="stateVariable" type="stateVariableType"
       maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>
```

```
<xsd:complexType name="stateVariableType">
    <xsd:sequence>
        <xsd:element name="argument" type="argumentType"
        maxOccurs="unbounded"/>
        <xsd:element name="valueRange" type="valueRangeType"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string"/>
</xsd:complexType>

<xsd:complexType name="valueRangeType">
    <xsd:sequence>
        <xsd:element name="class" type="xsd:string"
        maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>
<!— stateVariables definition START —>

<!— operators definition START —>
<xsd:complexType name="operatorListType">
    <xsd:sequence>
        <xsd:element name="operator" type="operatorType"
            maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="operatorType">
    <xsd:sequence>
        <xsd:element name="expression" type="expressionType"
            maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string"/>
</xsd:complexType>

<xsd:complexType name="expressionType">
    <xsd:sequence>
        <xsd:element name="slot" type="slotType"
            maxOccurs="unbounded"/>
    </xsd:sequence>
    <!— expression type: relation/prevailing state var./non–prevailing
        state var.—>
    <xsd:attribute name="type" type="xsd:string"/>
    <xsd:attribute name="delegate" type="xsd:string"/>
</xsd:complexType>

<xsd:complexType name="slotType">
    <xsd:sequence>
        <xsd:element name="content" type="xsd:string"/>
    </xsd:sequence>
    <!— content type: variable name/constant name/wildcard—>
    <xsd:attribute name="contentType" type="xsd:string"/>
    <xsd:attribute name="slotIndex" type="xsd:int"/>
```

```
        <xsd:attribute name="contentClass" type="xsd:string"/>
    </xsd:complexType>
    <!— operators definition END —>
</xsd:schema>
```

E.2 Problem scheme

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:jxb="http://java.sun.com/xml/ns/jaxb"
            jxb:version="2.0">
    <!— root element —>
    <xsd:element name="task" type="taskType"/>
    <!— root element definition —>
    <xsd:complexType name="taskType">
        <xsd:sequence>
            <xsd:element name="properties" type="taskProperties"/>
            <xsd:element name="constants" type="constantsSectionType"/>
            <xsd:element name="relations" type="relationSectionType"/>
            <xsd:element name="stateVariables" type="
                stateVariablesSectionType"/>
        </xsd:sequence>
    </xsd:complexType>
    <!— task properties START —>
    <xsd:complexType name="taskProperties">
        <xsd:sequence>
            <xsd:element name="name" type="xsd:string"/>
            <xsd:element name="domain" type="xsd:string"/>
            <xsd:element name="requirements" type="pddlRequirementsListType
                "/>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="pddlRequirementsListType">
        <xsd:sequence>
            <xsd:element name="requirement"
                type="xsd:string" maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>
    <!— task properties END —>
    <!— constants definition START —>
    <xsd:complexType name="constantsSectionType">
        <xsd:sequence>
            <xsd:element name="constantList"
                type="constantListType" maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="constantListType">
        <xsd:sequence>
            <xsd:element name="constant"
                type="constantType" maxOccurs="unbounded"/>
        </xsd:sequence>
```

```
    <xsd:attribute name="className" type="xsd:string"/>
</xsd:complexType>
<xsd:complexType name="constantType">
    <xsd:attribute name="name" type="xsd:string"/>
</xsd:complexType>
<!— constants definition END —>
<!— relations definition START —>
<xsd:complexType name="relationSectionType">
    <xsd:sequence>
        <xsd:element name="relationDef"
            type="relationDefType" maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="relationDefType">
    <xsd:sequence>
        <xsd:element name="signature" type="signatureType"/>
        <xsd:element name="table" type="tableType"/>
    </xsd:sequence>
    <xsd:attribute name="relationName" type="xsd:string"/>
</xsd:complexType>
<xsd:complexType name="signatureType">
    <xsd:sequence>
        <xsd:element name="argument"
            type="argumentType" maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="argumentType">
    <xsd:attribute name="number" type="xsd:int"/>
    <xsd:attribute name="class" type="xsd:string"/>
</xsd:complexType>
<xsd:complexType name="tableType">
    <xsd:sequence>
        <xsd:element name="row"
            type="rowType" maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="rowType">
    <xsd:sequence>
        <xsd:element name="column"
            type="xsd:string" maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>
<!— relations definition END —>
<!— stateVariables definition START —>
<xsd:complexType name="stateVariablesSectionType">
    <xsd:sequence>
        <xsd:element name="stateVariableDef"
            type="stateVariableDefType" maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="stateVariableDefType">
```

```
        <xsd:sequence>
            <xsd:element name="signature" type="signatureType"/>
            <xsd:element name="table" type="tableType"/>
        </xsd:sequence>
        <xsd:attribute name="stateVariableName" type="xsd:string"/>
    </xsd:complexType>
    <!— signatureType and tableType defined previously for relation
        definition —>
    <!— stateVariables definition END —>
</xsd:schema>
```

F. CD contents

The compact disk included with the thesis has following structure:

- `thesis.pdf` - the electronic version of this thesis.
- `vizzard_user_manual.pdf` - the user manual for the program *Vizzard* (standalone attachement D)
- `Vizzard` - *NetBeans* project directory containing the program source files
- `Vizzard.jar` - the executable file of program *Vizzard*
  - To run the program you need JRE installed on your system.
  - The minimal version required is `Java 1.6.0`
  - To run the program from the command line enter:

    ```
    java -jar Vizzard.jar
    ```
- `Examples` - example domain and problem files in XML format used by *Vizzard*