Comenius University
Faculty of Mathematics, Physics and Informatics
Department of Applied Informatics

Michal Čertický

An Architecture for Universal Knowledge-based Agent

Diploma thesis

Thesis advisor: doc. PhDr. Ján Šefránek, CSc.

Bratislava 2009

An Architecture for Universal Knowledge-based Agent

Diploma thesis

Michal Čertický

Comenius University
Faculty of Mathematics, Physics and Informatics
Department of Applied Informatics

Bratislava, Slovakia

9.2.11. Cognitive Science

Advisor: doc. PhDr. Ján Šefránek, CSc.

Bratislava, May 2009

Declaration

By this I declare that I wrote this diploma thesis by myself, only with the help of the referenced literature, under the careful supervision of my thesis advisor.

Bratislava, May 2009 Michal Čertický

Abstract

Typical approach to agent-oriented programming results in slight limitations of agent's usability. One agent is typically usable only in single environment type and for a small, strictly given set of tasks. We introduce a new term - Universal Knowledge-based Agent (UKA) - and propose an architecture and needed set of algorithms, providing complete methodology for its implementation.

One single implementation of this agent is usable in any environment and for various practical tasks, with modifications only in its hardware interface, thanks to environmentally-independent knowledge representation and general motivational mechanism inspired by a model from psychology. Convenient formalism for symbolic knowledge representation and reasoning is described in detail in this thesis, together with significant modifications of classical approaches.

Keywords: Knowledge Representation, Symbolic Reasoning, Agent-oriented Programming, Multi-agent Systems, Agent Architectures

Abstrakt

Typický prístup k agentovo orientovanému programovaniu má za dôsledok mierne obmedzenie použitel'nosti agentov. Jeden agent je zvyčajne vytvorený iba pre konkrétny typ prostredia a pre malú, striktne zadanú množinu úloh. V tejto práci definujeme nový pojem - univerzálneho znalostného agenta (Universal Knowledge-based Agent, UKA) - a navrhujeme aj jeho architektúru a potrebné algoritmy, čím poskytujeme kompletnú metodológiu pre implementáciu takýchto agentov.

Jediná implementácia takéhoto agenta je použitel'ná v akomkol'vek prostredí a pre rôzne praktické úlohy, s modifikáciou iba v jeho hardwarovom interface. To je dosiahnuté reprezentáciou znalostí nezávislou na prostredí a použitím adaptácie motivačného mechanizmu inšpirovaného modelom zo psychológie. Vhodný formalizmus pre symbolickú reprezentáciu znalostí a usudzovanie je v tejto práci detailne popísaný, spolu s rozsiahlymi modifikáciami klasických prístupov.

Kl'účové slová: Reprezentácia znalostí, Symbolické usudzovanie, Agentovo-orientované programovanie, Multi-agentové systémy, Agentové architektúry

Acknowledgements

A very big thanks goes to my advisor Ján Šefránek for his supervision and priceless constructive criticism, without which this thesis wouldn't exist in its present form.

I would also like to thank all my close friends and relatives, who had to give up the time I spent working on this thesis, that would otherwise belong to them.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

The goal of this work is to define a new term - universal knowledge-based agent (UKA) and propose a general architecture for its implementation. In the context of previous work our architecture can be understood as a novel approach agent-oriented programming based on well-known *belief-desire-intention* (BDI) model introduced by Rao and Georgeff in 1992 [3]. BDI model gets its name from the fact that it recognizes the primacy of beliefs, desires, and intentions in rational action [7]. It is based on a widely respected theory of rational action in humans developed by the philosopher Michael Bratman [10].

Bratman's theory is a theory of *practical reasoning* — the process of reasoning that we all go through in our everyday lives, deciding moment by moment which action to perform next [7]. Understanding this process is necessary to understand the concept of BDI and UKA agents.

Intuitively, an agent's *beliefs* in BDI model correspond to information the agent has about its environment. These beliefs may be incomplete or incorrect. Agent's *desires* represent states of affairs that the agent would, in an ideal world, wish to be brought about. Finally, agent's *intentions* represent desires that it has committed to achieving [7].

We will understand beliefs as *models of agent's environment* represented by symbolic knowledge-representation language, desires as similar (but less detailed) *models of goal state of environment* and intentions as *plans* (sequences of actions with certain parameters) leading to certain goal states.

1.2 Problems

Serious problem of typical approach to agent-oriented programming represent agent's usability limitations. Even if an architecture is general enough (like for example BDI), its single implementation can be used only for a small set of tasks (usually only one task) in a specific environment type. We are trying to propose an architecture, enabling its single implementation to be used in various environments and for various tasks with only minor (hardware-related) modifications. In other words, we are trying to grant an agent built on our architecture two novel traits: *environmental universality* and *task universality*.

This ambition is in contrast with typical practice of implementing different agent programs for different situations or tasks. On the other hand, it correlates to living agents (e.g. humans), who are able to perform more or less effectively in various kinds of environments (urban area, jungle, arctic tundra, etc.) with one implementation of cognitive system (neural system).

It is obvious, that if agent's implementation is supposed to be able to function with different sets of sensors and effectors (hardware), it will have to process extremely wide range of information. For that it will need a *flexible knowledge representation language*.

Also, since it is not specialized for any certain kind of environment, it needs to be able to *learn about causal rules* of its current environment in order to know how certain actions affect it (and use this knowledge while planning its own actions).

In dynamic or non-deterministic environments an agent cannot fully rely on his own plans, and needs to monitor the success of their execution. Mechanism for plan *execution monitoring* and *dealing with discrepancies* is therefore needed.

To ensure the task universality an agent needs to be given goal descriptions in its knowledge representation language. However, it is desirable to also ensure agent's autonomy and continuous knowledge expansion. For that we will need some convenient *motivational (goal-generation) mechanism*.

Since we want an implementation of our architecture to be usable in complex environments and for non-trivial tasks, we must solve the computational complexity issues by employing various *heuristic methods* (possibly inspired by living agents).

1.3 Solutions

Solution of *environmental universality* problem proposed by UKA architecture is philosophically based on Marvin Minsky's modular scheme - *Society*

of mind [16]. Minsky tried to explain "how mind works" by splitting it into many simpler processes, which he called mental agents. According to his scheme, each of those agents could only do a simple thing that needed no mind or thought at all. However, joining them into "societies" in certain special way, leads to emergence of what he called "true intelligence" [16]. Inspired by Minsky's modular approach, agent with UKA architecture will be represented by a multi-agent system - each significant cognitive process will be carried out by one simple agent, called *sub-agent*. To ensure universality, form of communication between these sub-agents will be environmentally independent and only two of them (directly controlling sensors and effectors) will be environment-specific (together they are called *hardware interface*). Respecting the idea of BDI model, communication between sub-agents consists of exchanging beliefs, desires, and intentions. Schematic description of UKA architecture and communication between its individual parts can be found in chapter 3.

To achieve *task universality*, *autonomy*, and continuous *knowledge expansion*, we need some kind of flexible goal-generation mechanism. Inspired by motivational process of living agents, we propose an algorithm generating goals according to Alderfers psychological ERG theory [4]. Those goals are ordered in 3-level hierarchy: Existence needs with highest priority, Relatedness needs with middle priority and Growth needs with lowest priority. Detailed information about this algorithm can be found in chapter 8.

The result of this approach is that our agent in fact does what it wants. Getting it to carry out our orders is possible because of the middle level of ERG hierarchy - we just need to pass our agent an information, that we wish something to be done (basically we are submitting a middle-priority goal).

Readers without significant psychological knowledge will perhaps be more familiar with Maslow's model of hierarchy of needs [1], which is similar to ERG model, only older and more complex.

As a *knowledge representation language* we have chosen extended logic programs and their modifications described in chapter 4.

For *learning about causal rules*, we introduce an algorithm called *Categorized Causal Hypothesis Generator*. It is described in the section 7.5.3.

Monitoring of plan execution and *dealing with discrepancies* is (among other things) carried out by subset of sub-agents (sensory sub-agent, effector sub-agent and planning sub-agent). Specific description can be found in chapters 6 and 7.

Heuristic approaches can be found on many places throughout the UKA architecture. As relevant examples we should mention the use of extended logic programs with three truth values instead of perhaps more realistic fuzzy logic (chapter 4), forgetting of older unused knowledge (section 7.5.1), conversion of hypotheses to background knowledge after their validation (section 7.5.2), introduction of modification of STRIPS formalism for representation of actions (section 4.5), heuristic strategy for dealing with discrepancies in plan execution (section 6.1), etc.

Chapter 2

Definition of UKA

In this chapter, we will define the term of *UKA - Universal Knowledge-based Agent*. Our definition is built on general definition of an agent, so we must choose one of many alternatives, since the term is often used in different meanings and various contexts.

For a common definition of the term *agent*, we can cite Russel and Norvig's textbook *Artificial Intelligence: A Modern Approach* from 1995:

Definition 1. *An agent* is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors [5].

We will now take this very general definition and use it to build a new term by specifying the set of additional conditions, that need to be met by an agent to be called universal knowledge-based agent.

Definition 2. *Universal Knowledge based agent* (UKA) is an agent which:

1. is usable in various environments *(environmental universality)*
2. is usable for various tasks *(task universality)*
3. has knowledge base representing its knowledge about environment
4. enlarges its knowledge base by observation and reasoning
5. is capable of making plans to fulfill its goals
6. is autonomous

First and second condition is sufficiently explained in "Problems" section of chapter 1.

The central component of any knowledge-based agent is it's knowledge base, or KB [5]. It can be intuitively understood as a set of sentences (or *"beliefs"* in terminology of BDI model) representing the information an agent has about its environment. Agent's knowledge may be incomplete and incorrect because of sensoric limitations.

Reasoning in fourth condition can be any appropriate algorithmic process modifying agent's knowledge base - for example causal hypothesis generation described in section 7.5.3. Enlarging KB by observation is self-explanatory.

By making plans we understand finding a sequence of actions leading from a given initial state of an environment to a given goal state.

Agent's autonomy is a capability of ensuring its continuous existence without an external assistance.

Chapter 3

UKA Architecture

We have chosen a paradigm of multi-agent systems to build UKA architecture. Main source of inspiration came from Marvin Minsky's modular theory of cognition [16]. Individual cognitive processes are simulated by separate agents (called sub-agents in UKA architecture), which can be understood as separate programs able to communicate with each other and with all the other parts of architecture. It is also possible for sub-agents to run on different computers and communicate via network. Cooperation of these sub-agents should result in emergence of intelligent behaviour.

The most important aspect of this approach is simplicity – complex problem is separated into several much simpler parts. It also makes the whole system versatile, since the there is a possibility to add new sub-agents for more cognitive functions, or to replace the old sub-agents with more effective versions.

In figure 3.1 we can see seven sub-agents and three non-agent software components of UKA architecture - Knowledge Base (called also Belief Base), Desire Base and Intention Base - named using the terminology of BDI model [3].

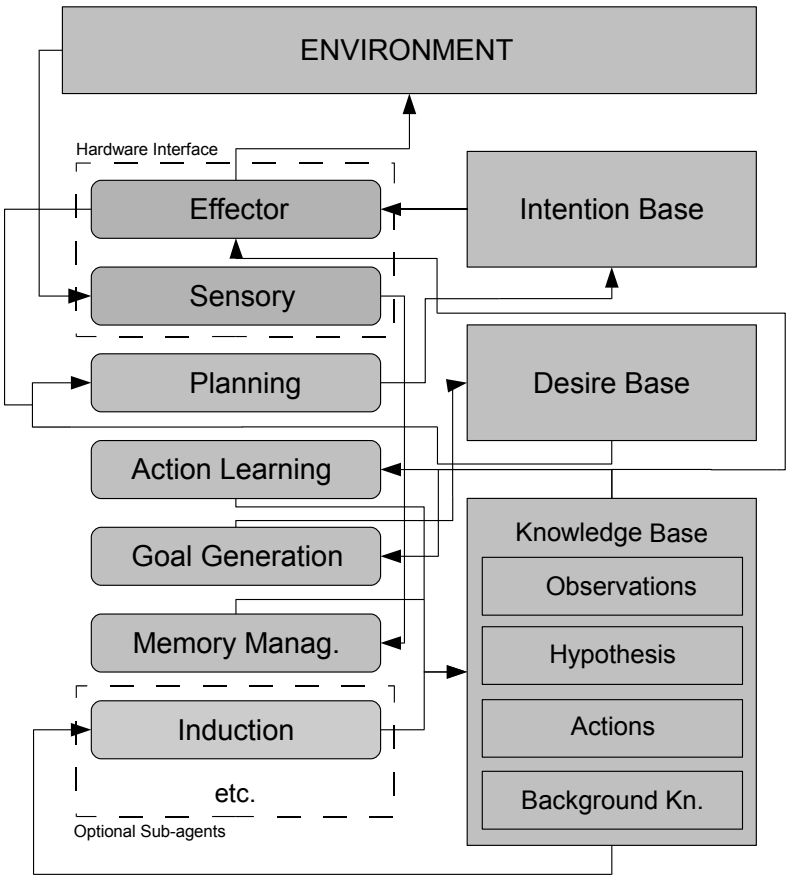

Figure 3.1: Scheme of UKA Architecture.

There is a possibility to employ various approaches from all fields of AI in the implementation of sub-agents (neural networks, evolution algorithms, etc.) and non-agent parts of architecture. Furthermore, each sub-agent can be implemented by a lower-level multi-agent system of its own.

Communication between separate parts of architecture is displayed in figure 3.1 by arrows. We can see that only two sub-agents communicate with environment, and hence are environmentally dependant: Sensory sub-agent gathers quantitative information from agent's hardware sensors, processes the data and outputs sets of environment-independent logical literals, which are sent to memory management sub-agent. Second one, called effector sub-agent, takes environment-independent plan and accordingly controls agent's hardware effectors. These two sub-agents together are called *Hardware interface*.

Remark 1. We must note, that implementation of those two sub-agents in non-trivial environments (e.g. real world) is potentionally complex and difficult task, depending on the set of hardware sensors and effectors. In general, conversion from perceived quantitative data into qualitative and environmentally independent interpretation represented by extended logic programs is a challenging problem. Similarly is the execution of formally represented plans by hardware effectors.

Implementation of each other part of the system is completely independent on environment, since the form of every other sub-agent's input and output is general enough to be usable in any environment. Communicated semantic information is represented by extended logic programs (definition 3) or their modifications and therefore consists of logical rules, facts or constraints composed of finite and strictly given set of predicate symbols, where only parameters are environment-specific, but sub-agents work with them only as with character strings.

See an example 1, where observed world-specific individual is represented by string *dog01*, and the value of world-specific attribute represented by string *color* is a string *black*.

Example 1.

$$has\_attribute(dog01, color, black)$$

We will describe individual sub-agents and the form of communication between them in corresponding sections of chapter 7.

Besides six compulsory sub-agents, UKA architecture allows us to add any number of optional sub-agents performing additional cognitive tasks. In our scheme (figure 3.1) we can see an example of such optional sub-agent - induction sub-agent communicating only with knowledge base.

Knowledge base, called also belief base in BDI terminology, is a database storing all the information agent has about its environment and itself. There are four different types of information stored in KB. Knowledge base and individual information stored in it will be described in detail in chapter 4.

Desire base is a software component storing agent's goals represented by incomplete logical descriptions of desired world states. It is described in chapter 5. Goals are added to desire base by goal generation sub-agent and read by planning sub-agent.

Intention base contains plans generated by planning sub-agent represented by finite sequences of actions leading to goals from desire base. Plans from intention base are then executed by effector sub-agent. More information about intention base can be found in chapter 6.

Chapter 4

Knowledge Base

As the name suggests, knowledge base (KB) is the central component of universal knowledge-based agent. KB in agent-oriented programming is a data structure representing the information an agent has about its environment. In terminology of BDI model, it is called "belief base". In our architecture, KB is a software component (possibly, but not necessarily, a standalone program) storing this information. Knowledge base of UKA needs to be able to communicate with certain sub-agents of UKA architecture according to client-server paradigm, where sub-agents are considered clients reading, adding, and modifying stored data.

Remark 2. Since several sub-agents must be able to access and modify stored information, we must remember to solve the synchronization issues when implementing KB. If we want to implement KB as a stand-alone program communicating with sub-agents via network, we should be able to use one of many third-party database implementations. Language for communication between sub-agents and KB could then be for example SQL.

Question arises, if KB should not be considered just one of the sub-agents, but calling KB an agent would be in conflict with definition of an agent (def. 1), since it doesn't in any way affect its environment. "Software component" is therefore more appropriate term.

Knowledge base of universal knowledge-based agent contains several types of information: observations, hypotheses, actions and background knowledge. Those four individual types of stored information will be described throughout this chapter.

4.1 Knowledge Representation

We have chosen an *extended logic programs* - ELP (definition 3) with a *closed world assumption* (definition 12) as a knowledge representation language for universal knowledge-based agent.

In addition to knowledge represented by ELP, KB must be able to remember various kinds of metainformation for each stored record. Types of these metainformation are specified further in the following sections of this chapter.

Definition 3. Let $n \geq 0$ and $m \geq 0$. An *extended logic program (ELP)* is finite set of rules of the form

$$c \leftarrow a_1, \ldots, a_n, not\ b_1, \ldots, not\ b_m$$

where a_i, b_j and c are literals, i.e., either propositional atoms, or such atoms preceded by explicit negation sign. The symbol "not" denotes negation by failure (default negation), "¬" denotes explicit negation [13].

Part of the rule before "←" sign is called *head* and part after it is *body*. Rule with empty body ($n = m = 0$) is called a *fact* and rule without head is a *constraint*. Facts can be written in abbreviated form without the implication symbol ("←").

If r is a rule of the form as above, then c is denoted by $head(r)$ and $\{a_1, \ldots, a_n, not\ b_1, \ldots, not\ b_n\}$ by $body(r)$.

As we can see, extended logic programs provide two kinds of negation operators. Explicit negation of an atom a ($\neg a$) is used, when we have explicit information (e.g. observation) of given atom a being false. Default negation of an atom a ($not\ a$) is based on the closed world assumption (definition 12)

and means that we are unable to resolve a from our current knowledge. In other words it means, that we don't know about a being true.

Representing knowledge with two kinds of negation (explicit and default negation) is useful for reasoning with incomplete knowledge, since it creates more expressive truth value system - see figure 4.1.

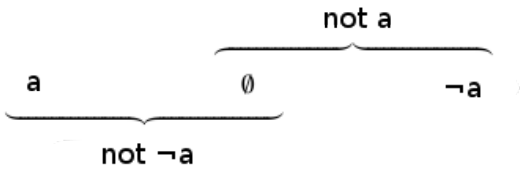

Figure 4.1: Truth values in ELP.

Remark 3. Note, that even when representation by extended logic programs allows us to have any finite amount of literals in the body of each rule, body of most common stored clauses - observations - is empty.

Following set of definitions builds up necessary terminology needed throughout the rest of this chapter. Specifically, we need to have a definition of stable model of ELP in order to introduce needed concept of SM-following (def. 11).

Definition 4 (Extended Herbrand base)**.** Let a be an atom of the form $p(t_1, \ldots, t_n)$, where p is n-ary predicate symbol and each t_i is a term (constant or variable). If no variable occurs in each t_i then the atom a is said to be *ground*. *Extended Herbrand Base* is the set of all literals of the form a or $\neg a$, where a is a ground atom.

Definition 5 (Grounding of a rule)**.** The *grounding* of a rule r, denoted by $ground(r)$, is the set of all rules obtained from r by all possible substitutions of constants for the variables in r. Similarly for a set of rules.

Definition 6 (Herbrand interpretation)**.** A *herbrand interpretation* of a logic program P is any coherent (not containing literals a and $\neg a$ together) subset $I \subseteq H(P)$ of its extended herbrand base. We say an interpretation $I \in S$ is *minimal* among the set of interpretations S if there does not exist an interpretation $J \in S$ such that $J \subset I$.

Definition 7 (Satisfying)**.** We say that interpretation I *satisfies* a rule r iff:

$$body(r) \subseteq I \Rightarrow head(r) \in I$$

Interpretation *satisfies* an extended logic program P, iff it satisfies each rule $r \in P$. We can also say, that rule r or extended logic program P is satisfied in interpretation I.

Definition 8 (Herbrand model)**.** A *herbrand model* of a logic program P is a herbrand interpretation of P such that it satisfies all rules in P.

Definition 9 (Program reduct)**.** Let P be a ground extended logic program. For any interpretation I, let *program reduct* P^I be a program obtained from P by deleting

- each rule r such that $body^{not}(r) \cap I \neq \emptyset$
- each default literal *not* a such that $a \notin I$,

where $body^{not}(r)$ denotes a set of all literals b, such that *not* $b \in body(r)$.

Definition 10 (Stable model)**.** An interpretation I of extended logic program P is a *stable model* iff I is a minimal model of program reduct P^I. We will denote a set of all stable models of program P as $SM(P)$.

Definition 11 ($\models_{SM}$)**.** A literal L *SM-follows* from a program P iff for each $S \in SM(P), L \in S$ (notation: $P \models_{SM} L$). A set of literals X SM-follows from a program P, iff each literal $L \in X$ SM-follows from P (notation: $P \models_{SM} X$). A rule r SM-follows from P iff r is satisfied in each $S \in SM(P)$. If U is a set of rules then $P \models_{SM} U$ iff $\forall r \in U\ P \models_{SM} r$.

Definition 12. *Closed World Assumption* with given logic program P:

$$CWA(P) = \{not\ a \mid P \not\models_{SM} a\}$$

4.2 Observations

First kind of information stored in KB are simple environmental observations (definition 13) provided by agent's sensors. Sensory sub-agent observes the environment with all available sensors in short time intervals, interprets those observations as ELP facts and sends them to memory-management sub-agent. After some processing (described in chapter 7) it sends the observations to KB along with the time of the observation. KB also stores metainformation about the time of last access to each observation, which is later used again by memory-management sub-agent (in a process of forgetting). Every single observation is hence stored in KB in the form of an ELP fact with additional 2 numbers.

Definition 13. An *observation* in KB of UKA architecture is a data structure consisting of three parts:

1. ELP fact $\mathcal{F}$ corresponding to fact observed by sensory sub-agent,
2. numeric representation of time when the fact was observed by sensors,
3. numeric representation of time when the observation was last accessed by sub-agents,

where fact $\mathcal{F}$ has one of the following forms:

1. "$has\_attribute(o, a, v)$" with **o** identifying observed object (individual), **a** observed attribute of the object **o**, and **v** attribute's value
2. "$finished\_action(o, a, p_1, p_2)$" with **o** identifying object (also called actor), which performed some action **a**, and $\mathbf{p_1}$ and $\mathbf{p_2}$ parameters of performed action

Example 2. *Observations stored in KB of UKA architecture:*

1212630372; 1212630567; $finished\_action(john, switch, button03, on)$.
1212630372; 1212630372; $has\_attribute(light\_bulb01, on\_off\_state, off)$.

First observation in example 2 means that in time 1212630372 (unix timestamp in this case, but there can be any other representation of time) observed individual represented by string "john" switched on the button number 3, and that last time this information was accessed by sub-agents is 1212630567.

Fact *finished_action* describes the observation of some individual (actor) finishing certain action. First parameter identifies an actor performing the action. Second parameter identifies the type of observed action and other two are parameters of performed action. Last two parameters can be left blank for some actions. Such action describing clauses are inspired by human speech - four parameters should be sufficient for description of almost every commonly observed action. Architecture could be however easily modified to allow larger number of action parameters.

Remark 4. Actor can also be an agent itself. In that case the actor identifier is string "*me*".

Second observation in example 2 is simpler and much more frequent. Fact *has_attribute* describes any attribute of observed object (given as the first parameter). In our example, the attribute is "*on_off_state*" and value is "*off*" (both strings), what means that light bulb number 1 is turned off. Similarly like with the finished_action clause, the object of observation can be an agent itself (identified as "*me*").

Remark 5. Since UKA architecture works with all the predicate parameters simply as with strings, there can be any observable attributes, objects, actions or action parameters. Set of those strings is different for each environment and consists of anything, the sensory sub-agent outputs.

We have chosen those two forms of clauses for our observations, because they provide the possibility to produce hypotheses about actions and their effects on the objects, which enables an agent to learn new ways of affecting the environment, and are general enough to keep environmental universality.

The reason to remember *last access time* of each observation is forgetting. Inspired by the human cognition, memory management sub-agent erases observations, that haven't been used for long enough time (specified by constant), from knowledge base in order to make the reasoning processes faster.

4.3 Hypotheses

Hypotheses are products of agent's reasoning and resemble classical *rules* of extended logic programs (with one literal in the head and one or more literals in the body). In the KB of UKA architecture, hypotheses are represented by a data structure consisting of two such rules and two numeric records - number of observations that validate the hypothesis, and observations that falsify it (definition 14).

Definition 14. Let $n > 0,$ *or* $m > 0$. A *hypothesis* in KB of UKA architecture is a data structure consisting of four parts:

1. rule $\mathcal{R}$ of a form $c \leftarrow a_1, \ldots, a_n, not\ b_1, \ldots, not\ b_m$ representing a current hypothesis,
2. rule $\mathcal{O}$ of the same form as $\mathcal{R}$, representing the original hypothesis before any modifications by sub-agents,
3. number of its validations,
4. and number of its falsifications.

Representation of hypotheses slightly differs from other types of knowledge stored in KB - while the all the facts and rules of observations or background knowledge are always considered true, the truth value assigned to rules of hypotheses is a real number from interval $\langle 0, 1 \rangle$. This value is determined by the number of falsifications and validations (definition 15).

Definition 15. Let v be a number of validations, and f number of falsifications of hypothesis h. We call $t_h \in \mathbb{R}$ a *truth value of hypothesis* h when:

$$t_h = \frac{v}{v + f}$$

Number of validating and falsifying observations is refreshed by memory-management sub-agent always when a new related observation is added. When truth value of hypothesis is high enough ($t > c$, where c is a well chosen constant - for example 0.9), and the amount of related observations sufficient ($v + f > n$, where n is another constant - for example 10), then our agent considers the hypothesis validated and memory management sub-agent can move it to background knowledge part of KB (it is wise to keep an amount of hypothesis low, because they are all checked when the new observation is added).

Details about this process can be found in "Memory Management Sub-agent" section of chapter 7. In the same section, there is also a description of an algorithm called *CCHG* (Categorized Causal Hypotheses Generator), which works with categorization of environment and appropriately generalizes or specifies the hypotheses (using the record of original rule $\mathcal{O}$ - second item in the definition 14).

This approach makes learning possible in a non-deterministic and dynamic environments, where results of actions are not always the same and observations can be inaccurate. UKA agent is basically building new (background) knowledge over a time period by first *forming the hypotheses* and after that *continuously deciding* if they are true or false by observations. If hypotheses were bivalent, they could be easily incorrectly validated or falsified by one inaccurate observation, while truth value assigned by continuous function more closely resembles the way humans work with their hypotheses.

4.4 Background Knowledge

Background knowledge is the simplest type of knowledge stored in KB of UKA architecture. It consists of ELP rules, facts and constraints, which are used in the process of reasoning [8] and planning [14]. Unlike hypotheses, they are always considered true. Also we don't need to remember any additional metainformation - see definition 16.

Definition 16. A *background knowledge* in KB of UKA architecture are data structures representing ELP rules, facts or constraints.

Background knowledge consists of information of two different types and origins: validated hypotheses and basic clauses submitted by programmer of an agent, including also categorization-related knowledge.

Remark 6. An agent built on this architecture alone is not capable of creating any categorization system, but this capability can be easily added in future, taking advantage of modularity of UKA architecture, by adding a new subagent to the system. For now, we will only use categorization system created by programmer. This fact however, is consistent with universality condition, since categorization system is optional and agent is functional without it.

First part consists of all the hypothesis agent produces and later validates by sufficient number of observations. As described in chapter 7, they are converted into background knowledge rules by memory-management subagent in order to lower the computational complexity.

Second part of background knowledge consists of categorization-related knowledge and other basic rules or constraints. Categorization of objects encountered in the environment is important (for example) for producing hypotheses

about the world, because these are based on observations of individual objects. To make the hypothesis usable, it needs to say something about larger set of objects, not just one – it needs to be generalized. Reasonable generalization is possible through categorization. Algorithm using categorization knowledge for producing hypotheses is also described in chapter 7.

Example 3.

$$category(creature).$$
$$category(human).$$
$$category(male).$$
$$category(female).$$
$$belongs\_to\_category(john, male).$$
$$subcategory(male, human).$$
$$subcategory(female, human).$$
$$subcategory(human, creature).$$

As we can see in example 3, there are three types of ELP facts, which are sufficient for creation of potentially complex categorization systems for objects encountered by agent in it's environment.

Figure 4.2 is a visualisation of simplistic categorization system defined by example 3. There are three various categories defined by unary fact "category". Also we can see three relations (binary facts "belongs" and "subcategory") defining the structure of our categorization system. As we can see, individual objects can belong into categories (like object "john" in the example), and categories can be subcategories of other categories, creating acyclic graphs - in this example a tree.

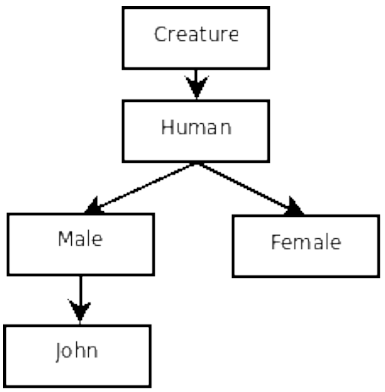

Figure 4.2: Simple categorization graph.

However, just those eight facts are not sufficient to compose the whole desired system. For example object "john" doesn't belong to category "human", which is not exactly what we wanted. In order to achieve something like that, we should add two rules into KB:

- $subcategory(C_3, C_1) \leftarrow subcategory(C_3, C_2), subcategory(C_2, C_1)$
- $belongs(O, C_1) \leftarrow belongs(O, C_2),\ subcategory(C_2, C_1)$

Those rules are necessary for our categorization system to work properly. First rule says, that if any category is a subcategory, it is also a subcategory of higher categories. Second rule says, that if an individual belongs to a category, it also belongs to higher categories. Rules like this are typical examples of background knowledge submitted to KB during the implementation by programmer. They can, but don't have to be all environment dependant (example above is environmentally independent). Also they don't have to be categorization-related. Another good example (example 4) of a clause submitted by programmer is a constraint saying, that one parameter of an object cannot have two different values.

Example 4.

$$\leftarrow has\_attribute(O, A, V_1), has\_attribute(O, A, V_2), V_1 \neq V_2.$$

Remark 7. Note that this constraint is just an example and is not compulsory part of background knowledge. Also it is an abbreviated form - last literal "$V_1 \neq V_2$" is an abbreviation for formally correct literal $not\_equal(V_1, V_2)$.

4.5 Actions

For our purposes of reasoning about actions, we need to have their representation in agent's knowledge base. It will be based on a classical representation language STRIPS. However, we needed to modify it to be usable along with ELP-based representation of other kinds of knowledge, and for reasoning with incomplete knowledge.

Remark 8. It is important to distinguish between the original STRIPS *planner*, and the STRIPS representation language. Due to its power of representation, the language developed for describing STRIPS' operators and world models was since then used, with minor modifications, in a large number of traditional (but more advanced STRIPS planners) [2]. UKA architecture uses partial modification of this language in combination with language of extended logic programs to represent agent's actions in KB.

We represent every agent's action on two levels: *action operator* and *action routine*. Abstract *operators* have their corresponding *routines* - procedures whose execution causes an agent to take actual actions in it's environment. Significant difference is, that operators are applied to world models during the planning (chapter 7), while execution of routines actually causes an agent to affect the world by it's effectors. Planning can therefore be understood as an attempt to find a sequence of operators whose associated routines will lead to a desired state of the world. Just like routines have their procedural arguments, operators have their own corresponding parameters.

Since every action's routine is a programming procedure directly controlling agent's effectors, it is necessary to have unique initial small set of very simple actions represented in KB corresponding to current agent's effector set. Language of routines used for representation of those actions can be any procedural programming language runnable by effector sub-agent. Operators however, are abstract and environment/hardware-independent.

Representation of operators in STRIPS language [15] consists of three major components :

1. Name of operator and it's parameters
2. Preconditions, and
3. Effects.

First component consists merely of name of the operator and its parameters (arity ≥ 0). Those parameters are variables in action's representation in KB. However, in the process of planning, the operator is applied on world models with certain constants instead of variables. Therefore when deciding the applicability of an operator, or computing the world model after its application, all the corresponding variables in preconditions or effects are first replaced by constants. We can see, that one operator can be applicable on world model with one set of constants as parameters, and inapplicable with different one. Also one action can modify the same world model differently, depending on its parameters. From now on, we will talk about applicability and effects of an operator, but we should bear in mind, that we are always talking about an operator with its own variable parameters substituted by certain constants in advance.

Second component in classical STRIPS is a formula in first-order logic. We will however, since we are working with an incomplete knowledge, use two ELPs consisting of facts only (definition 3), instead of one FOL formula: let's call them $preconditions^+$ and $preconditions^-$. Operator will then be applicable on a world model M, iff each fact from $preconditions^+$ and no fact from $preconditions^-$ SM-follows from $M \cup B$, where B is agent's background knowledge.

Definition 17 (Applicability of operator / action)**.** Let B be a set of ELP rules representing agent's background knowledge. Let P^+ and P^- be two sets of ELP facts representing $preconditions^+$ and $preconditions^-$ of operator O (action A). Operator O (action A) is then *applicable* on a world model M iff holds:

$$\forall p \in P^+ : M \cup B \models_{SM} p \ \wedge \ \nexists n \in P^- : M \cup B \models_{SM} n$$

Third component of an operator description defines its effects represented by so-called *effect descriptors* (definition 18) which are stored in two sets called (like in STRIPS [15]) *add list* and *delete list*. Effects in classical STRIPS are represented simply by a single literals and are applied to any world model. However, we want our effects to have some conditions for their applicability saying when (on which world model) they can be applied. Therefore we introduced *effect descriptors* consisting, besides the actual description of effect (called *modifier*), also of their own conditions of applicability (called *positive and negative requirements*). There is one more improvement over classical STRIPS, where one effect could only add/remove one literal with always the same constant parameters. Our effect descriptors can contain variables in both modifier and requirements, which allows them to add/remove literal with different constants according to current world model.

Definition 18 (Effect descriptor)**.** Let e be an ELP literal and R^+, R^- two coherent sets of ELP literals. Triple (e, R^+, R^-) is called an *effect descriptor* of operator (action). We will call e a *modifier* and R^+, R^- sets of *positive and negative requirements* of effect descriptor (e, R^+, R^-). All literals in effect descriptor can contain variables as parameters, but if there is a variable in modifier, there has to be the same variable also in R^+ or R^-.

Remark 9. Effect descriptor (e, R^+, R^-) intuitively says, that if R^+ is true and R^- is false, we modify our current world model by adding or deleting literal e. Classical STRIPS doesn't provide such conditions for applicability of effect. However, they allow us to represent actions more intuitively: In example 5, in add list of an action $shoot(X, Y)$, we can see an effect descriptor $(wumpus\_dead(yes), \{wumpus\_position(X, Y)\}, \emptyset)$ saying, that if *wumpus* is in position X, Y, it dies. If we were using classical STRIPS, we would have to assume that *wumpus* dies even if we didn't know that it was in position X, Y, which is not true.

Those effect descriptors specify how to create a new world model, using their modifier literals. However, since there can possibly be some variables in effect descriptors, first thing to do is to get rid of them by substitution. This can produce several instances of one effect descriptor for each world model, because there can be more than one possibility for such substitution.

Definition 19 (Effect instance)**.** Let M and B be two sets of ELP rules representing a world model and background knowledge. Let (e, R^+, R^-) be an effect descriptor, possibly containing variables as parameters of some of its literals. Let $\theta_1 \dots \theta_k$ be all possible (substitution) functions $\theta_i((e, R^+, R^-)) = (e_i, R_i^+, R_i^-)$ such that each variable of (e, R^+, R^-) is substituted by a constant in (e_i, R_i^+, R_i^-) and holds:

- $\forall l \in R_i^+ : M \cup B \models_{SM} l$
- $\nexists l \in R_i^- : M \cup B \models_{SM} l$

Then for each $i \in \{i \dots k\}$ we call (e_i, R_i^+, R_i^-) an *instance of effect descriptor* (e, R^+, R^-). We say that an effect descriptor is *applicable* on a world model M w.r.t. background knowledge B iff it has at least one instance.

In other words, we have some effect descriptors for each action. Each of those descriptors has several (possibly zero) instances for current model (w.r.t. BK). We produce those instances by first substituting its variables for constants and then checking positive and negative requirements. In each effect instance, those requirements are met, and all the parameters are constants. Thus we consider them applicable and we are able to use their modifiers to compute new world models.

Computation of a world models by one effect descriptor is different when it is in an add list or in delete list of an action.

Definition 20 (Computing the world models)**.**
Let O be an action operator of action A. Let M_1 be an ELP representing a world model. Let L_A be a set of all the modifier literals from all the instances (for model M_1, w.r.t. BK) of effect descriptors contained in *add list* of operator O (action A). Similarly, let L_D be a set of all the modifier literals from all the instances (for model M_1, w.r.t. BK) of effect descriptors contained in *delete list* of operator O (action A). M_2 is a world model *computed by applying operator O (or action A)* on the world model M_1 iff holds:

$$M_2 = (M_1 \setminus L_D) \cup L_A$$

Notation: $M_1 \rhd_O M_2$ or $M_1 \rhd_A M_2$

Remark 10. Representation of action's effects is evidently different than in classical STRIPS. It provides individual requirements for applicability of each effect of one action and allows effects to modify different world models differently, by using variables as parameters, thus making representation of actions richer and more intuitive - see example 5.

Now, that we have defined the concepts of effect descriptors, their instances, and applicability of operator, we can finally define a term "action" for UKA architecture.

Definition 21. An *action* in KB of UKA architecture is a data structure consisting of two parts:

1. action's *routine* represented by procedure in any programming language runnable by effector sub-agent,
2. and corresponding action's *operator* consisting of:
   (a) action's name and parameters,
   (b) preconditions represented by by two ELPs - $preconditions^+$ and $preconditions^-$ - which define a conditions of applicability and inapplicability,
   (c) and two sets of effect descriptors - *add list* and *delete list*.

In example 5 we will see an operator of simple action described in our modified STRIPS language. It is one of a few actions from early computer game called Hunt the Wumpus. This game makes, as Michael Genesereth suggested, an excellent environment for intelligent agents [5].

In *Wumpus world* an agent explores a dark cave consisting of two-dimensional array representing interconnected rooms. Lurking somewhere in the cave is the wumpus - a beast that eats anyone who enters its room. To explain our example, we don't need to fully specify the rules of such environment. It is sufficient to say, that an agent can perceive (smell) wumpus if it is in the room next to it and can shoot an arrow into rooms adjacent to its position (if it still has some arrows left).

Example 5. *Operator of an action in Wumpus world environment.*

- Action's name and parameters:
  $shoot(X, Y)$.
- Preconditions$^+$:
  $P^+ = \{adjacent\_position(X, Y)\}$
- Preconditions$^-$:
  $P^- = \{arrows(0)\}$
- Add list:
  $AL = \{\ (wumpus\_dead(yes), \{wumpus\_position(X, Y)\}, \emptyset),$
  $(arrows(A), \{arrows(B), succ(A, B)\}, \emptyset)\ \}$
- Delete list:
  $RL = \{\ (arrows(C), \{arrows(C)\}, \emptyset)\ \}$

Notice, that we have variables X and Y as parameters of our operator. However, we will always try to apply an operator with X and Y substituted by constants. Let's say that we want to shoot to position a-3 in our world. We will replace each X in *preconditions*$^+$, *preconditions*$^-$, *add list*, and *delete list* by constant "a". The same goes for Y and "3". Then we will check the applicability by trying the dependence of "$adjacent\_position(a, 3)$" and "$arrows(0)$" on our model w.r.t. background knowledge.

After we found out, that our action is applicable on current world model, we can proceed to computation of a new world model by finding instances of effect descriptors and applying them.

In add list we have an effect descriptor, whose modifier is unary literal $wumpus\_dead(yes)$. It has only one requirement - $wumpus(X, Y)$, where

variables X and Y are already substituted by constants "a" and "3". Next descriptor is more interesting. Modifier is $arrows(A)$ with requirements $\{arrows(B), succ(A, B)\}$. We need to find such constant replacements for variables A and B, that those requirements would be SM-modelled by current world model with BK in order to compute effect instances. Let's say, that there is literal $arrows(5)$ in our world model and literal $succ(4, 5)$ in our background knowledge. By substituting constant 5 for A, and 4 for B, we get one instance of such effect. Since we are considering an effect descriptor from add list, we add substituted modifier $arrows(4)$ to our world model.

Remark 11. Example 5 serves only as an explanation of proposed modification of STRIPS representation language. Literals displayed there are not typical literals used in UKA architecture.

Remark 12. To conclude this section about action representation we must note, that each action of UKA agent is observed by the agent itself. We take this fact into account when planning. Therefore add list of each action $a(X, Y)$ in UKA must contain effect descriptor saying that this very action is performed by our agent - $(finished\_action(me, a, X, Y), \emptyset, \emptyset)$ - and its delete list must contain another effect descriptor saying that no other action is performed by our agent at that moment - $(finished\_action(me, A, B, C), \{finished\_action(me, A, B, C)\}, \emptyset)$. As we will see in chapter 8, some agent's goals lead to validation of its hypotheses. Making plans leading to such goals requires these effect descriptors.

Chapter 5

Desire Base

Desire base is one of three non-agent software components of UKA architecture. Its purpose is to store current agent's desires, called also *goals*, each represented by two sets of ELP literals (definition 3) - D^+ and D^- - together describing desired state of the world.

We can understand each literal in D^+ as one condition that an agent wants to be met and each literal in D^- as a condition that it wants not to be met. A goal is reached in any world state modelled by M, where $\forall p \in D^+ : M \models_{SM} p$, and $\nexists n \in D^- : M \models_{SM} n$.

Each goal also has its numeric priority assigned by goal-generation sub-agent.

Definition 22. *Goal* (desire) in UKA architecture is a data structure consisting of:

1. two sets of ELP literals (D^+ and D^-) describing desired state of the world,
2. and numeric priority assigned to the goal.

Goal is an incomplete description of a world state. In typical case one of the sets D^+ and D^- contains only one literal, while other set is empty - see example 6.

Example 6.

- $D^+ = \emptyset$
- $D^- = \{has\_attribute(me,\ energy,\ low)\}$

In example 6 we can see two sets of literals describing a simple goal. Set D^- contains a single literal, saying that an agent has low energy. Since this literal is in D^-, the goal is reached in any world state, from whose model this literal doesn't SM-follow (definition 11).

Goals are generated by goal generation sub-agent according to ERG motivational model described in chapter 8. Then they are accessed by planning sub-agent, which tries to create plans leading an agent to such desired states.

Example 6 describes typical high-priority goal (more information about priorities can also be found in chapter 8). Naturally, a sequence of actions leading an agent to this goal state would include getting to some kind of power source and recharging its energy supply.

Name "desire base" is chosen for this software component according to Rao and Georgeff's BDI model [3], where desires represent states of affairs that the agent would, in an ideal world, wish to be brought about [7].

It is currently accessed by two sub-agents: goal generation sub-agent, which can add new goals to desire base, and planning sub-agent, which reads and deletes the goals and tries to create plans leading to corresponding world states.

Remark 13. One of many possible implementations of desire base is, similarly as in case of knowledge base, a standalone database server. Communication language could in that case also be SQL and desires could be sorted by their priority. Another possibility is any implementation of priority queue, as long as sub-agents can access and modify its contents.

Regardless of selected way of implementation, programmer needs to solve synchronization issues, just like in case of knowledge base, since at least two sub-agents can possibly access (read or modify) a desire base at a same time.

Chapter 6

Intention Base

Intention base is the last non-agent software component of UKA architecture. It stores plans (definition 23) added by planning sub-agent and executed by effector sub-agent.

Each plan is a sequence of several stages, consisting of a description of one single action and intended state of the world after its execution. Since we want our agent to be functional in nondeterministic environments, we need this description in order to be able to check if the action was executed successfully.

Definition 23 (Plan)**.** Let A be one of actions from agent's KB, denoted by an atom of a form $a(p_1, \ldots, p_n)$, where a is action's name, and $p_1 \ldots p_n$ are its non-variable parameters. Let $I^+ = \{a \mid a$ is a modifier of any instance of any effect descriptor $e \in add\ list\ of\ A\}$ and $I^- = \{d \mid d$ is a modifier of any instance of any effect descriptor $e \in delete\ list\ of\ A\}$ be a pair of sets describing intended state of the world after execution of action A.
Triple (A, I^+, I^-) is then called a *plan stage*. A *plan* in UKA architecture is

any finite sequence of plan stages.

Definition 24 (Leading)**.** Let M_i be a model of initial state of the world and M_g a model of desired (goal) state. Let $A_1, A_2, \ldots, A_n$ be actions corresponding to individual stages of a plan P. We say a plan P *leads* from initial world state (modelled by M_i) to goal world state (modelled by M_g) iff:

$$M_i \rhd_{A_1} M_1,\ M_1 \rhd_{A_2} M_2,\ \ldots,\ M_n \rhd_{A_n} M_g$$

Definition 25 (Correct plan)**.** We say, that a plan P consisting of stages $S_1, \ldots, S_n$ leading from a world state modelled by M_1 to a world state modelled by M_n is *correct*, iff for each subsequent stages S_i, S_{i+1} with corresponding world models M_i, M_{i+1} holds:

$$M_i \rhd_{A_i} M_{i+1} \textit{ and } A_{i+1} \textit{ is applicable on } M_{i+1}$$

where A_i and A_{i+1} are actions of stages S_i and S_{i+1} respectively.

Intention base contains *correct* plans *leading* (definitions 25 and 24) from current state of the world, to one of the goal states from desire base. Intuitively it means, that from the model of current world state, we can compute (definition 20) a model of goal state by subsequently applying all the actions of our plan.

Each plan in intention base has assigned a numeric priority equal to the priority of a goal it leads to.

Intention base is, similarly to desire base, accessed by two sub-agents: planning sub-agent, which adds new plans, and effector sub-agent, which always takes the plan with highest priority and executes it.

Just like in case of knowledge base and desire base, there are many possible implementations of intention base, but a programmer has to keep in mind a

need for synchronisation, since more than one sub-agent at a time can try to access (read or modify) data stored in it.

6.1 Plan Execution and Discrepancies

In real-world domains, executing a plan without monitoring is a fragile strategy, since non-determinism might cause unintended effects of actions which, for instance, may prevent the execution of the rest of the plan from reaching a goal state [6].

Therefore it is important for an agent to be able to monitor the execution of it's actions and also to have some strategy for dealing with discrepancies. It is a good idea to check, after each stage of the plan, if it was successful (definition 26). This check is performed by effector sub-agent, which consults the "observations" part of knowledge base.

Definition 26. Let O be a set of ELP literals representing agent's observations after execution of a plan stage (A, I^+, I^-). We say that a plan stage (A, I^+, I^-) was *successful* iff:

- $I^+ \cup O$ is a coherent (not containing literals a and $\neg a$ together) set,
- and $I^- \cap O = \emptyset$.

Effector sub-agent should however always wait until the new observations are added to KB after executing each stage, in order to have up-to-date information enabling it to decide if the stage was successful. In case the last stage was successful, effector sub-agent can move on to next stage of the plan, while deleting finished stage from the plan.

However, if up-to-date observations show, that a stage was unsuccessful, or that an action of next stage is not applicable on $M \cup O$ (where M is current world model and O current observations), we have reached so called "point of failure" [6] and the strategy for failure recovery is needed.

Suggested strategy differs in two cases: if the remaining plan is too long (longer than a given constant), and if it is short enough (shorter than a given constant).

If the plan is short enough, there is a good chance that planning sub-agent can quickly find an alternative plan to achieve a goal state from the current one. In that case the planning sub-agent gets the request to do so, and replaces the current plan with an alternative (if it cannot be found, the new goal is chosen).

In other case, if the remaining plan is too long, it is less computationally expensive to deal with discrepancy and then continue executing the rest of the plan. Planning sub-agent, in this case, needs to find only a short plan leading from the current state of the world to a non-failure state described by the current unsuccessful stage of the plan and preconditions for an action of a next plan stage (thus $D^+ = I^+ \cup P^+$ and $D^- = I^- \cup P^-$, where D^+ and D^- describe a non-failure state, I^+ and I^- intended state after current stage, and P^+ and P^- are preconditions of an action of the following plan stage). This plan is then added before the remainder of our original plan, and executed (if it cannot be found, the procedure is the same as in previous case – an alternative plan to get to the goal is found and replaces the original one).

See figure 6.1 for a flow chart depicting this strategy.

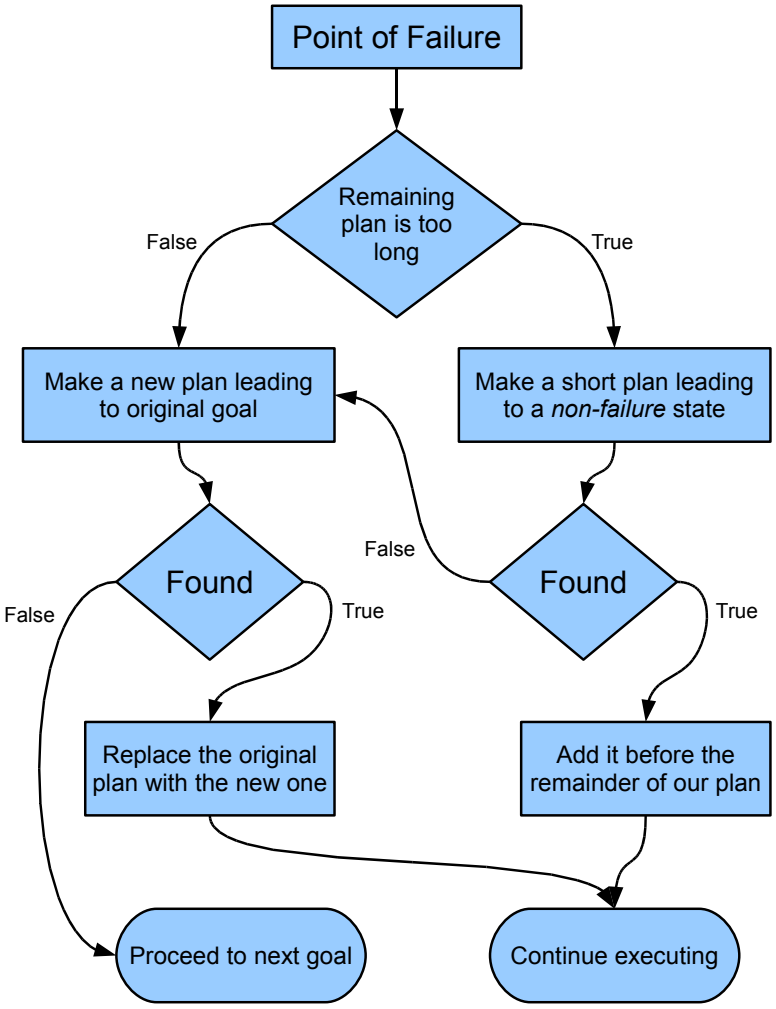

Figure 6.1: Heuristic strategy for dealing with discrepancies.

This heuristic approach is meant to shorten the time spent by finding the plans, since the plan-finding is very computationally expensive, but on the other hand the plans produced like that can in some cases be not the optimal, shortest plans. Also we shouldn't forget, that in the non-deterministic world and with incomplete knowledge, an agent can get to the situation, when its plan is not executable and it gets into infinite loop trying to complete it. Therefore it is a good idea to limit the number of attempts to recover from the discrepant states, and stop executing the plan after that limit is reached. For example, if an agent gets to the point of failure 5 times, it could consider the plan inexecutable and move on to the next goal. Alternatively, the number of attempts doesn't have to be constant and can depend on the priority of current goal (higher priority means higher number of attempts).

Remark 14. There is also a possibility for an agent to learn some extra information from the fact, that he got to the point of failure. Information can be produced by abductive reasoning: in the point of failure, we have a set of observations, which are different than the expected state of the world, and we should somehow explain these observations. By abductive reasoning we can (on the basis of our previous knowledge) produce the hypothesis about the causes of these observed facts. This kind of reasoning could be in future performed by some sort of optional sub-agent (lets say an "abductive sub-agent").

Chapter 7

Sub-agents

In this chapter we will describe all the individual sub-agents of UKA architecture (figure 3.1), one by one. For each of them, we will define their input, output and function. Environment of all the sub-agents is the architecture itself.

Input of each of them can be understood as sensoric input - perception of certain kind of information from specified part of architecture.

Output is understood as sub-agent's way of affecting its environment - other parts of architecture.

Function of a sub-agent is a description of its purpose in the architecture.

First two described sub-agents will be sensory and effector sub-agent, which are two components of "hardware interface" - only part of UKA architecture communicating directly with hardware, thus dependant on environment. Rest of the chapter will describe completely environmentally independent sub-agents.

7.1 Sensory Sub-agent

Input

Input of sensory sub-agent is any kind of information received from hardware sensors. Our agent's environment can be either real world or some kind of artificial environment (e.g. simulation). In case of real-world environment, typical sensors can be for example a camera, IR sensor, microphone, and so on. In artificial environments, agent's sensors will probably be some program components (object, interface...).

Output

Output of sensory sub-agent are sets of ELP facts of one of two forms:

$$has\_attribute(o, a, v)$$

where o denotes observed object, a name of observed attribute of object o, and v value of this attribute. All three parameters are character strings. An agent can possibly observe itself - in that case $o =$"me".

$$finished\_action(o, a, p_1, p_2)$$

where o denotes an observed object (called also an *actor*) which performed action a with two parameters p_1 and p_2. Two parameters should be sufficient for description of most observable actions - in most cases they are not even necessary (and can be an empty string). If an agent observes its own actions, $o =$"me".

In fact, it is compulsory for sensory sub-agent to observe each action performed by our agent and output this observation. This is later used for generation and validation of agent's hypotheses.

Another observations compulsory for sensory sub-agent are observations about *emergency states*. Autonomous agent needs to be able to make hypotheses about what situations are dangerous for it, in order to be able to avoid damage. Since there are different kinds of dangerous situations with different hardware (e.g. high temperature or low energy), it is a task of sensory sub-agent to perceive those hazards and send this information to KB. A typical example of observation about emergency state is described by an ELP fact $has\_attribute(me, emergency, overheating)$. Observations of this kind are then used by inductive sub-agent to create hypotheses about the causes of emergency states. If such hypothesis is validated, an agent knows what causes the emergency state and can make a high-priority plan leading it out of that state. Read more in chapter 8.

Function

Sensory sub-agent continuously checks the output of all available sensors, converts this information into ELP facts of a form specified above, and in short time interval sends a set of all gathered facts to memory-management sub-agent.

This task is in general potentially very complex (especially in real-world environments) and UKA architecture itself doesn't provide any universal algorithm for it. Being a part of hardware interface, sensory sub-agent must be implemented separately for each set of hardware sensors, and conversion of quantitative sensoric percepts into qualitative ELP output is different problem in each case. That is the reason, why it is not possible to provide detailed information about the function of this sub-agent.

7.2 Effector Sub-agent

Input

Main input of effector sub-agent are plans from intention base, which are described in definition 23. It always reads the plan with highest priority and tries to execute it. In order to be able to execute any actions and monitor their execution, effector sub-agent must also have access to knowledge base.

Output

Output of effector sub-agent are actual actions physically (at least in real-world environments) performed by agent's hardware effectors.

Function

Effector sub-agent performs the plans from intention base in successive steps. There is always only one plan at a time being executed by it - chosen according to priority after finishing previous plan. In each step, effector sub-agent takes following plan stage (if there is any - if not, it proceeds to next plan with currently highest priority), gets needed routine corresponding to an action of that specific stage from KB, and tries to perform it with the same parameters (sensory sub-agent needs to be able to interpret this routine represented by an algorithm in one concrete procedural programming language). After the execution, it waits until sensory sub-agent adds the next set of observations to KB (it is sufficient to wait for time in which observations are added - communication with sensory sub-agent is not necessary), and checks if the it hasn't reached the point of failure (section 6.1). In such case, it needs to

deal with discrepancy before executing next plan stages. Strategy for dealing with discrepancies is described in detail in section 6.1.

7.3 Goal-generation Sub-agent

Input

Goal generation sub-agent takes as its input an information about observed emergency states ($has\_attribute(me, emergency, E)$), background knowledge about causes of those states, and unvalidated hypotheses from agent's knowledge base.

Output

Output of goal-generation sub-agent are goals represented, accordingly to definition 22, by two sets of ELP literals (D^+ and D^-) describing desired state of the world, and numeric priority assigned to the goal.

Function

Goal-generation sub-agent produces goals according to low level and top level of ERG hierarchical motivational model (Existence needs and Growth needs). Those goals are assigned priority $P = 3$ for existence needs and $P = 1$ for growth needs. They are then added to agent's desire base. Actual generation of goals is described in detail in chapter 8.

7.4 Planning Sub-agent

Input

Input of planning sub-agent consists of three parts:

- goals from agent's desire base (always the one with highest priority),
- data from knowledge base consisting of observations used to model initial world state, and actions together with background knowledge used to compute next world models,
- and requests from effector sub-agent for short plans needed to deal with discrepancies in plan execution.

Output

Planning sub-agent produces *correct* plans *leading* from initial state to goal state of the world. Those plans are either added to agent's intention base with assigned the same priority as was the priority of a goal, or sent to effector sub-agent as a reply for its request.

Function

The need for modeling the behavior of robots in a formal way led to definition of logic-based languages for reasoning about actions and planning. They allow us to specify a planning problems of the form "find a sequence of actions leading from a given initial state to a given goal state"[14].

Formal logic-based language for reasoning about actions used in UKA architecture is based on STRIPS language and is described in detail in section 4.5

and throughout the rest of chapter 4. Our agent must have a capability to solve the planning problems of this kind and choose actions on the basis of generated plans.

Goal states are stored in agent's desire base and current initial state is always described by up-to-date observations stored in knowledge base. Planning sub-agent always takes the goal with highest priority from desire base and tries to produce a (preferably shortest) plan leading from current state of the world to any state of the world from which the goal state SM-follows (w.r.t. background knowledge). As described in definition 23, plan is a sequence of plan stages - triples (A, I^+, I^-), where A is an action (stored in KB) with certain non-variable parameters and I^+, I^- are sets describing intended state of the world after execution of action A. Moreover, plans generated by planning sub-agent need to be correct and leading from initial to goal state (definitions 25 and 24). Naturally, not every goal from desire base is reachable. Therefore planning sub-agent can fail to produce a plan leading to such goal state. In that case it moves on to next goal chosen according to priority. However, if any plan is produced, planning sub-agent adds it into intention base with the same priority as has the goal it leads to.

Since planning is computationally complex operation, we will probably need to limit the time spent by computing a plan. Time available to planning sub-agent should depend on the priority of a goal. The higher the priority is, the more time planning sub-agent should have. Also it is recommended to employ heuristic methods when implementing this sub-agent. There are many possibilities to improve performance of planning algorithms. Since planning includes frequent computation of stable models of a world states (to test SM-following of various literals), it is a good idea to heuristically reduce a model representing initial state by using only such observed facts,

that can be resolved from the goal by a sort of backward chaining using actions as inference rules ($preconditions \rightarrow effects$) - this is can however lead to loss of some solutions. This is only one of many possible heuristic modifications a programmer can choose from.

Planing sub-agent is also used to deal with discrepancies in plan execution. Besides plans leading to goals from desire base, it also creates short plans requested by effector sub-agent. If planning sub-agent receives such request, it either interrupts current planning task, or starts a new thread for computation of requested plan. This process of dealing with discrepancies is described in section 6.1.

7.5 Memory-management Sub-agent

Input

Memory management sub-agent receives its input from sensory sub-agent: sets of observations represented by ELP facts of a form $has\_attribute(o, a, v)$ or $finished\_action(o, a, p_1, p_2)$, described in section 7.1. All the parameters of received ELP facts are character strings. Those observations are received in short time intervals and represent latest percepts from all agent's sensors.

In addition to observations received from sensoric sub-agent, memory management sub-agent has read-write access to knowledge base.

Output

Output of this sub-agent are observations with corresponding time and newly created hypotheses submitted to agent's KB, as well as modifications of existing hypotheses or their numbers of validation and falsification. It is also capable of reorganising KB by moving (converting) the information between separate parts of KB, or deleting observations which were not used for long time.

Function

Memory management sub-agent performs parallely three separate tasks: it deletes (forgets) old and unused observations from KB, organizes knowledge in KB, and inserts new observations received from observation sub-agent into KB. Since those tasks are supposed to run parallely, the most intuitive implementation is to run each of those tasks in a separate thread.

The rest of this section describes those three tasks/functions of memory management sub-agent in detail.

7.5.1 Forgetting Observations

In certain constant time intervals, memory management sub-agent consults the knowledge base and if there are any observations with time of their last access (definition 13) older than given constant (chosen accordingly to agent's hardware performance), it deletes them. This keeps computational complexity (for example of hypothesis consistency check) on a low level, since the amount of remembered information doesn't grow uncontrollably.

7.5.2 Migration of Knowledge

Second function of memory management sub-agent is migration of knowledge. Specifically it moves the hypotheses with sufficient number of validations and truth value (definition 15) high enough from hypotheses part of knowledge base to background knowledge, so they can be used for later reasoning. Any hypothesis, with number of validations higher than a constant c_1 (for example 10) and truth value higher than a constant c_2 (for example 0.9) is deleted from the hypotheses part of KB, and ELP rule representing it is added to background knowledge part of KB. This rule is from this moment considered true in all the reasoning processes.

Remark 15. You can see, that this is purely heuristic approach aimed to lower the computational complexity. Once the hypothesis is considered to be true, it is moved to background knowledge, and cannot be falsified despite the fact, that it is potentially incorrect. On the other hand it is not necessary

to check its validity with every new related observation. If the computational complexity wasn't a real issue, it would be better not to move any hypotheses to background knowledge and compute a set of all hypothesis with truth value momentarily high enough for each reasoning process.

7.5.3 Insertion of Observations

The most complex function of memory management sub-agent is the insertion of new observations recieved from sensory sub-agent. Besides simple adding of observations to KB it includes creation of new causal hypotheses, their generalisation or specification and validation or falsification.

Causality Inference

One of the algorithms designed for UKA architecture takes care of understanding the causal rules of agent's environment with the help of categorization (part of agent's background knowledge). We call this algorithm *CCHG (Categorized Causal Hypothesis Generator)*. It takes agent's observations as an input and produces hypotheses (ELP rules) about causal relation between observed actions and their effects. Also it validates or falsifies former hypotheses, thus modifying their truth value, and modifies old hypotheses when their truth value is too low. Categorization of observed individuals is used when determining how general the produced hypothesis should be.

Every time, when memory management receives a new set of observation from sensory sub-agent, it runs CCHG algorithm with each observation from this set as an argument. Figure 7.1 depicts this algorithm called with received observation O in time t.

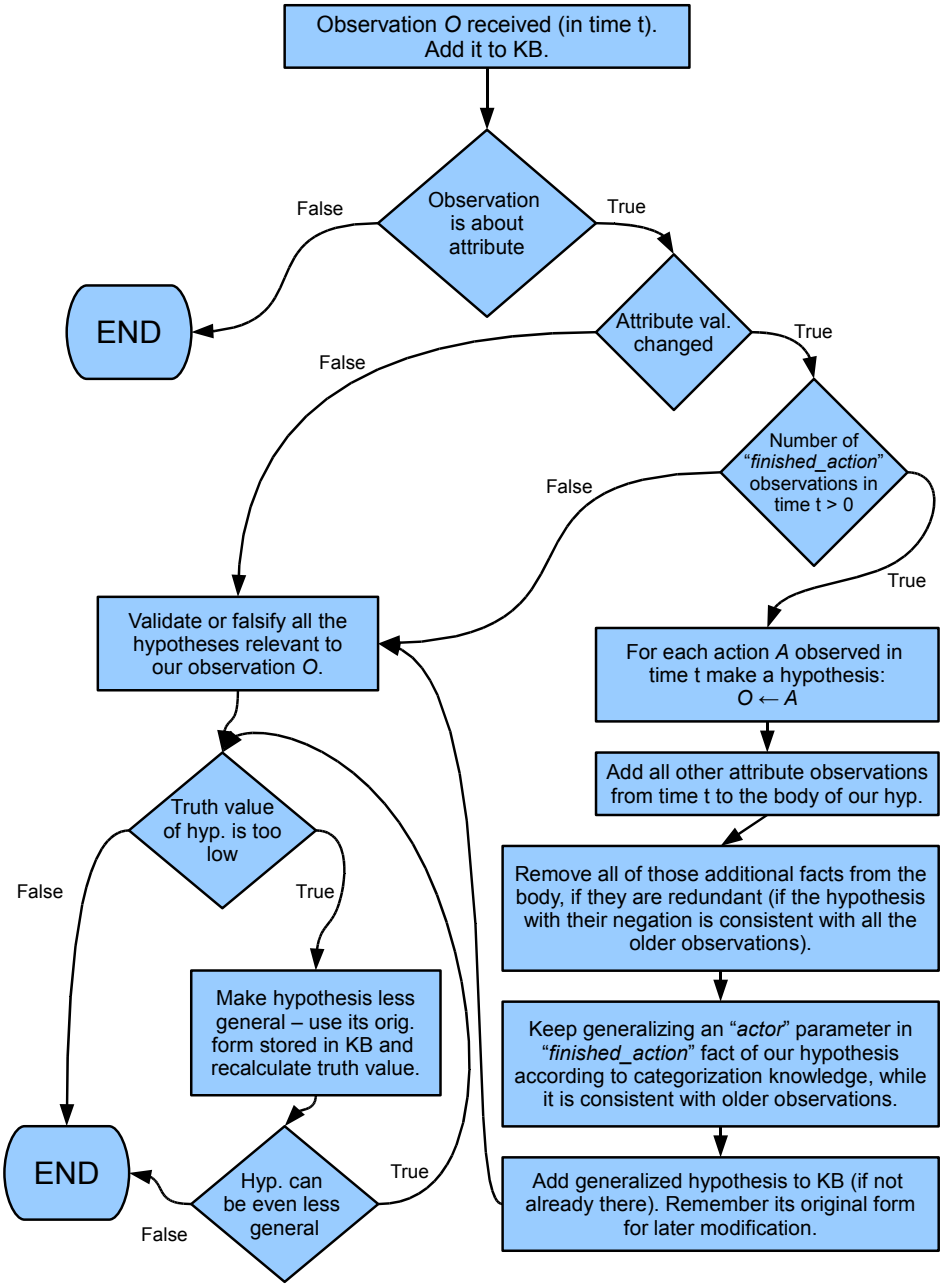

Figure 7.1: CCHG algorithm adding an observation into KB.

After simply adding the observation into KB, our algorithm checks if it is of a form $has\_attribute(obj, att, val)$. If it is, it finds out, whether the value of that observed attribute has changed since the last observation.

If the attribute value has changed, it takes all the observations of a form $finished\_action(actor, action, p_1, p_2)$ observed in time t and produces a hypothesis of a form:

$$has\_attribute(obj, att, val) \leftarrow finished\_action(actor, action, p_1, p_2)$$

After that, it adds all the attribute observations except our original observation O from time t to body of our new hypothesis and removes the ones that are redundant. Attribute observation O' is redundant in hypothesis H, if the same hypothesis with O' replaced by $\neg O'$ is consistent with all the older observations in KB.

After this step we have a set of hypothesis, each with one attribute observation (O) in head, and body consisting of one action observation (A) and finite set (C) of attribute observations. In natural language it means, that when conditions C are met, an action A causes O.

Last thing to do with such hypotheses is to use available categorization knowledge to generalize them, thus making them applicable in wider range of future situations. In order to generalize the hypothesis, we check if an actor of action A belongs to some category c, and if it does, we replace it with a variable V and add a literal $belongs\_to\_category(V, c)$ to the body of a hypothesis. However, after that we must check whether our generalized hypothesis is still consistent with our older observations. If it is, and if c is a subcategory of some higher category c', we can generalize even more and replace c with c'. We repeat this until we reach inconsistency with older observations.

When we have such generalized hypothesis, we add it to KB under *hypotheses* part (if it is not already there). Subsequently we validate or falsify all hypotheses related to our current observation O by adding 1 either to their *validations* or *falsifications*. If their truth value is not too low after this, there is nothing left to do. However, if the truth value of hypothesis is suddenly too low, we can try to make it less general, using its remembered original form, and then recalculate its number of validations and falsifications.

Remark 16. Despite the fact, that this algorithm takes advantage of agent's categorization knowledge, such knowledge is not absolutely necessary. In case that observed actor doesn't belong to any category, it simply stays ungeneralized, but hypothesis is still usable.

7.6 Action-learning Sub-agent

Input

Input of action learning sub-agent are observations about actions performed by an agent itself stored in knowledge base.

Output

Action learning sub-agent produces new, more complex actions composed from the repeated sequences of previously used simpler actions, and adds them into "actions" part of knowledge base.

Function

Action learning sub-agent provides an agent built on UKA architecture a capability to learn new actions by repetition of older, simpler ones. UKA agent learns new actions by monitoring the sequences of its own actions for repetitive patterns and composing repeated sequences of simple actions into more complex ones. This process is inspired by implicit learning of living agents and lowers the computational complexity of planning tasks by making the produced plans much shorter (e.g. complex action *walk* can be composed of sequence of actions *move_left_leg* and *move_right_leg*).

We can expect, that certain actions will be carried out in agent's "lifetime" often in more or less repetitive patterns, as certain repeated tasks require always the same sequences of actions. So if we provide an agent with just a small set of very simple actions, it is optimal for it to learn using these repeated sequences as a more complex actions. Again let's take a human as

an example (but let's forget about common theories of human cognition – our hypothetical human serves just as a metaphor for action learning of UKA agent). Let's say that a small child only knows simple actions like moving it's left and right foot forward and lifting it up (it is of course very simplified case, but sufficient for our explanation). It's goal would be changing the position in the world (movement). In the beginning, the movement is very complex task for it, and it would have to plan every leg movement. However, later after some amount of attempts, it (unconsciously) realizes that it repeats the sequence of lifting left foot, moving it forward, putting it down and doing the same with right foot, which leads to effective movement. This movement is called walking and it is an example of complex action created by combination of several simple actions. Walking later becomes a subconscious skill and requires no planning. Knowing complex actions like this significantly shortens agent's plans (as for example the long sequence of moving feet is replaced by single action "walk" in a plan) and makes planning faster. In UKA architecture, the combination of repeated actions into more complex ones is accomplished by action-learning sub-agent.

Since an agent observes all of its own actions, we have an information about their execution stored in "observations" part of knowledge base. Therefore it is possible for action learning sub-agent to access this information and analyse it in order to find any repetitive patterns. If such pattern is found, action learning sub-agent composes a new action from the sequence of repeated actions (composition joins both operators and action routines). Such composed action is then added into "actions" part of knowledge base by action learning sub-agent.

7.7 Inductive Sub-agent

Input and Output

Induction sub-agent communicates only with knowledge base. It regularly checks the newest observations together with background knowledge and produces hypotheses that it adds to knowledge base. Those hypotheses are later validated or falsified by memory-management sub-agent by the same algorithm, that validates or falsifies the causal hypotheses generated by it.

Function

Planning sub-agent enlarges knowledge base by the means of inductive reasoning. Using the data mining techniques, induction sub-agent gathers additional information from subsequent sets of agent's observations stored in knowledge base (also with use of agent's background knowledge).

The most important (and necessary for agent's full function) is extraction of knowledge about reasons of emergency states. Agent needs to learn which of situations it gets into are dangerous, in order to be able to make plans leading it to safety.

If an agent for example repeatedly observes emergency state saying that its body is overheating - $A = has\_attribute(me, emergency, overheating)$ - along with observation saying that it is near to a fireplace denoted by string $fireplace01$ - $B = has\_attribute(fireplace01, near\_to\_me, yes)$ - it can produce a hypothesis of a form:

$$A \leftarrow B$$

Of course, causes of emergency states can be more complex, so there can be more than one literal in the body of such rule. Also there can be more different causes for one type of emergency.

This process resembles the cognitive processes of living agents: observed emergency states can be seen as a perception of pain caused by pressure, heat, cold, etc. Those percepts warn living agents about various kinds damage dealt to their bodies. Observation of emergency state by UKA agent also warns about possible damage to its hardware and there also are various kinds of perceivable emergency states. Inductive learning of causes of emergency states resembles the way that living agents learn what causes the pain, so they can avoid it.

There are various data mining techniques usable for extraction of such knowledge. However, since they are too complex to be extensively described in this text, we will at this moment leave the choice for a programmers implementing UKA architecture.

Chapter 8

Motivation and ERG Model

Common approach in agent-oriented programming is to design an agent just for finite set of task types. However, if we want to meet the requirement of task universality, we need more progressive approach. Inspiration is taken from the ability of living agents to perform more or less effectively in different kinds of environments. Living agents (e.g. humans) can with the same implementation of cognitive system (neural system) live in various conditions (urban area, jungle, arctic tundra, etc.) quite effectively if they possess enough knowledge about it.

Our ambition is to more closely model the motivational process of living agents. Inspiration is therefore taken from a psychological theory of human motivation - ERG motivational model.

ERG Theory appeared in 1969 in an article titled "An Empirical Test of a New Theory of Human Need" [4] in Psychological Review. It is a revision of classical Abraham Maslow's Hierarchy of Needs by Clayton Alderfer, and it was created to align Maslow's motivation theory more closely with empirical

research. Acronym ERG stands for three levels of this hierarchical model: Existence needs, Relatedness needs, and Growth needs.

Alderfer reduced the number of levels of original Maslow's Hierarchy of Needs to three. This simplified model not only aligns better to human motivational system, but is also more convenient to use for creation of artificial intelligent agents. Generation of different types of goals with numeric priority according to those three levels, results in autonomous behaviour of agents, possibility to formulate various orders on the run, and spontaneous behaviour leading to enlarging agent's knowledge.

Like Maslow's model, the ERG motivation is hierarchical, and creates a pyramid or triangle appearance. Existence needs motivate at a more fundamental level than relatedness needs, which, in turn supercedes growth needs.

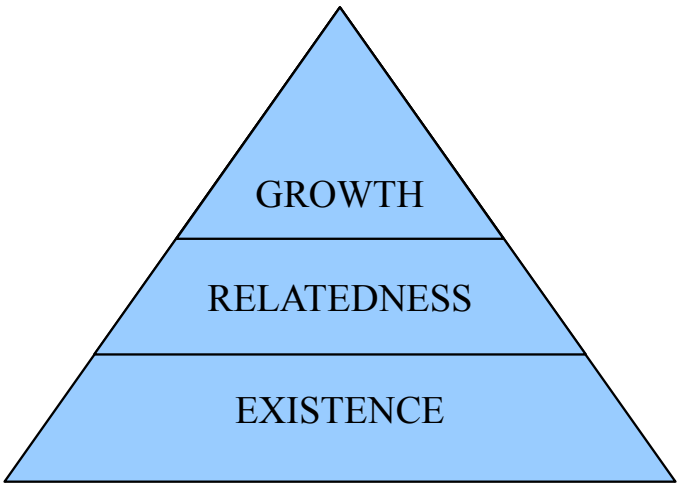

Figure 8.1: Alderfers ERG model of motivation.

- *Existence needs* refer to human's concern with basic material existence motivators. Existence needs of universal knowledge-based agent make goal-generation sub-agent produce goals leading out of dangerous situations. Example of dangerous situation is a world state, where an agent's hardware is overheating, or its energy supply is dangerously low. To be able to produce plans leading out of this situation, an agent needs to have knowledge about its causes. Goals of this level have highest priority in UKA architecture and plans leading to them are executed first.

- *Relatedness needs* refer to the motivation humans have for maintaining interpersonal relationships. This level gives us opportunity to formulate any middle-priority goals (in given knowledge representation language) for our artificial agent. If there are no goals with higher priority, an agent tries to make a plan leading to one of those goals and execute it.

- *Growth needs* refer to an intrinsic desire for personal development of humans. For UKA, growth needs result in generation of goals leading to enlarging of practically usable part of its knowledge base by validating or falsificating agent's hypotheses. Goals leading to such exploratory behaviour have lowest priority in UKA architecture.

Remark 17. Note, that this motivational model can cause an agent to behave differently than we may want in case of emergency. Since existence needs have highest priority, safety of an agent is its main concern in dangerous situations and it therefore ignores other goals - including our orders. In some cases, this behaviour can be considered incorrect, but it shouldn't be a problem to solve this by modifying the motivational model.

Now, that the motivational model is explained, we can move on to implementation issues. Main idea is, that goals are generated by a static algorithm of *goal-generation sub-agent*. This sub-agent consults agent's knowledge base in time intervals and generates goals with priority $p \in \{3, 2, 1\}$ corresponding to three levels of ERG model, and adds them to desire base.

Planning sub-agent then takes always the goal with highest priority and tries to generate a plan leading to it. If it succeeds, the plan is added to the intention base. Otherwise, it moves on to next goal (again according to priority). Since the planning phase can be time consuming, it might be a good idea to limit a time of plan-generation accordingly to goal's priority.

When the plan is added to intention base by planning sub-agent, it can be accessed by effector sub-agent. It always takes the plan with highest priority and starts the execution. Again, an execution of a plan can also be unsuccessful - in that case, effector sub-agents moves on to next plan with highest priority.

8.1 Low level - Existence needs

Low-level goals are meant to satisfy the existence needs of an agent. If there is an observation about an emergency state in KB (see section 7.1), an agent is in a dangerous situation. In that case a goal-generation sub-agent adds a goal leading the agent out of that situation. In order to generate such goals, we need to have some knowledge about the causes of that emergency. This knowledge is typically gained by validating hypotheses produced by inductive sub-agent (part of such knowledge can also be added by a programmer). With this information about causes of emergency, generation of a goal is simple:

For each observation o about emergency state found in KB, goal-generation sub-agent searches background knowledge for ELP rules of a form "$o \leftarrow l_1, \ldots, l_k, not\ l_{k+1}, \ldots, not\ l_n$". For each found rule, a generated goal is described by positive conditions $D^+ = \{l_1, \ldots, l_k\}$, negative conditions $D^- = \{l_{k+1}, \ldots, l_n\}$, and priority $P = 3$ (see definition 22 for explanation of such goal description).

8.2 Middle level - Relatedness needs

Relatedness needs of universal knowledge-based agent make it produce a plans leading to goals received from outside (submitted by human). They provide us with a possibility to control an agent. When a goal is submitted, it goes right into agent's desire base and there is no need for planning sub-agent's intervention. Submitted goals are incomplete descriptions of a desired world states represented in a language of ELP accordingly to definition 22. They have assigned a middle priority: $P = 2$.

8.3 Top level - Growth needs

Growth needs of universal knowledge-based agent are satisfied by enlarging its usable knowledge. When there aren't any more important things to do, an agent tries to learn something new. This learning is understood as validating or falsificating of hypotheses stored in agent's knowledge base. Each hypothesis in KB is represented (besides some metainformation) by an ELP rule of a form "$H = a \leftarrow b_1, \ldots, b_k, not\ b_{k+1}, \ldots, not\ b_n$". In order to validate or falsify such hypothesis, an agent needs to get to a world state, where the

body of hypothesis is observable. Then if it observes the body and also the head part of the hypothesis, it can add a validation. If the body is observed, but the head is not, one falsification is added to the hypothesis. Therefore the obvious goal leading to a validation of hypothesis H is described by positive conditions $D^+ = \{b_1, \ldots, b_k\}$, negative conditions $D^- = \{b_{k+1}, \ldots, b_n\}$, and priority $P = 1$.

Chapter 9

Conclusion

To conclude this thesis, we will first summarize the whole work, then enumerate most important of its contributions, briefly describe the inspiration drawn from fields other than artificial intelligence, and finally establish topics for future research related to this work.

Summary

Introduction of a term UKA. We have introduced a new term of Universal Knowledge-based Agent. Our definition of such agent (def. 2), built on classical definition by Russel and Norvig [5], describes all its key attributes and explains differences between UKA and other agents.

Design of UKA architecture. Throughout the whole thesis, we managed to design a modular architecture for universal knowledge-based agent, including important algorithms necessary for its function, and providing complete methodology for its implementation (apart from environmentally dependant hardware interface). An architecture is designed as multi-agent system,

where all the internal communication is environmentally independent. Any implementation of this architecture is thus usable in any environment and for various sets of (practical) tasks, with changes only in hardware interface (two sub-agents of multi-agent system).

Solution to problems of environmental and task universality. Environmental universality is possible because of completely environmentally-independent knowledge representation and communication of sub-agents, where specific individuals encountered/observed by an agent are represented by constants and denoted by character strings. Task universality is achieved by using an adaptation of psychological model of human motivation (ERG [4]) for our purposes and making it usable with symbolic knowledge representation (This adapted model allows user to submit descriptions of desired states of the world as goals for our agent. Whether the goal is reachable for our agent or not depends on the set of actions it can perform.).

Modularity and possible improvements. Since an architecture is mildly inspired by Marvin Minsky's philosophy presented in Society of Mind [16], it consists of a multi-agent system where several sub-agents perform individual cognitive processes. This approach leads to modularity of our architecture, and provides us with possibility to replace any sub-agent with more effective version, or add new sub-agents for different cognitive processes.

Knowledge representation. Choice of extended logic programs as a knowledge representation language allows us to effectively represent incomplete knowledge about the environment, by using three truth values (two kinds of negation operator).

Contributions

Improvement of STRIPS language. In chapter 4 we introduced a modification of STRIPS formalism, better usable in combination with extended logic programs and incomplete knowledge. Unlike the original STRIPS, our modification allows us to decide whether each action is executable even with incomplete description of world model. Also it allows us to specify applicability conditions for each effect of a certain action and to use variables in effect descriptions, which results in fact, that each effect can modify different world models differently. Our modification also takes into account agent's available background knowledge when computing world models and deciding applicability of actions/effects.

CCHG algorithm. In chapter 7 we proposed an algorithm called CCHG (Categorized Causal Hypotheses Generator), which produces hypotheses about observed actions and their effects (learning by observation), generalizes those hypotheses according to agent's available categorization information, and validates or falsifies them by later observations.

Algorithm for recovery from discrepancies in plan execution. Another algorithm proposed in this thesis uses heuristic approach to deal with any discrepant situations encountered in execution of agent's plans. It is necessary for effective function in dynamic or non-deterministic environments.

Commercial potential. Implementation of UKA architecture is, thanks to its usability in different environments and for various tasks, commercially interesting, since one program can be used for many purposes (let's say in robotics or various simulations) with only a minor changes.

Conversion of psychological model into AI. Interesting is also an attempt to use psychological model of human motivation (ERG) in artificial intelligence for production of agent's goals.

Multidisciplinary inspiration

Marvin Minsky's philosophy. Inspiration for modular UKA architecture was drawn from cognitive model proposed in *Society of Mind* [16].

Psychological ERG motivational model. We adapted Clayton P. Alderfer's model of human motivation, originally based on Maslow's classical Hierarchy of Needs.

Learning by observation. Generation of causal hypotheses is originally based on learning processes of living agents.

Learning by repetition. Production of new actions by repetition of sequences of older, simpler ones is also inspired by living agents.

Forgetting of old and unused knowledge. Heuristic reduction of computational complexity by deleting unused knowledge is based on the memory loss phenomenon of living agents.

Future research

Implementation and testing. First phase of future research concerning UKA architecture is obviously its implementation and testing in different environments. This phase can possibly uncover additional weaknesses of such architecture, which would lead to more improvements.

Human-agent communication. Communication between UKA agent and human is needed for submission of orders by human. UKA architecture currently doesn't describe this communication, leaving the space for improvement. In ideal case, orders could be submitted by natural language (phonetic or textual representation).

Agent-agent communication. Future research should be directed also to the possibilities of cooperation of UKA agents, communication between them and team work in problem solving.

Emergence of new categories. In its current state, UKA architecture doesn't have any mechanism producing a categorization system for an agent (categorization is optional part of knowledge base submitted by a programmer). This provides another area for improvements.

Abductive reasoning. An example of currently unsupported type of reasoning is diagnostic/abductive reasoning. It could be used for example to diagnostically uncover the reasons of encountered discrepancies in plan execution.

Predicting the development of current situation. In dynamic environments, it would be useful for an agent to be able to predict the changes in the world not directly caused by its own actions. It could be used for example to detect dangerous situations sooner, and therefore avoid damage more effectively.

Research on goals/tasks. In relation with task universality, further research should be done on agent's goals, their different classes, approaches/patterns leading to achieving those goals, and even their performability.

Bibliography

[1] Maslow, A. H. 1943. *A Theory of Human Motivation. Psychological Review*: 1943.

[2] Cocosco, C. A. 1998. *A Review of "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving by R.E. Fikes, N.J. Nillson, 1971".*

[3] Rao, A. S. - George, M. P. 1992. *An Abstract Architecture for Rational Agents. Knowledge Representation and Reasoning.* 1992.

[4] Alderfer, C. P. 1969. *An Empirical Test of a New Theory of Human Needs. Psychological Review*: 1969.

[5] Russell, J. - Norvig, P. 1995. *Artificial Intelligence: A Modern Approach.* Englewood Cliffs: 1995.

[6] Eiter, T. - Erdem, E. - Faber, W. 2004. *Diagnosing Plan Execution Discrepancies in a Logic-based Action Framework. INFSYS Research Report 2004.*

[7] Hendler, J. - Kitano, H. - Nebel, B. 2008. *Handbook of Knowledge Representation.* Elsevier: 2008.

[8] Muggleton, S. - Raedt, L. 1994. *Inductive logic programming: Theory and Methods. Journal of Logic Programming 1994.*

[9] Šefránek, J. 2000. *Inteligencia ako výpočet.* IRIS Bratislava: 2000.

[10] Bratman, M.E. 1987. *Intention, Plans, and Practical Reason.* Harvard University Press, Cambridge: 1988.

[11] Šefránek, J. 2006. *Irrelevant updates and nonmonotonic assumptions. Logic in Artificial Intelligence : Proceedings.* Berlin: Springer-Verlag. 2006.

[12] Fitting, M. 1992. *Kleene's logic, generalized. Journal of Logic and Computation.* 1992.

[13] Dix, J. - Brewka, G. 1996. *Knowledge Representation with Logic Programs.* Dept. of CS of the University of Koblenz-Landau: 1996.

[14] Eiter, T. - Faber, W. - Leone, N. - Pfeifer, G. - Polleres, A. 2000. *Planning under Incomplete Knowledge. Lecture Notes in Computer Science 2000.* Springer Berlin / Heidelberg: 2000.

[15] Fikes, R. E. - Nilsson, N. J. *STRIPS: A new approach to the application of theorem proving to problem solving. Artificial Intelligence 2*: 1971.

[16] Minsky, M. 1986. *The Society of Mind.* Simon and Schuster, New York: 1986.