Erik Sandewall

Artificial Intelligence and the Design of Cognitive Systems

Volume I:

Domain Modelling and Cognitive Architectures

Version 1.0, completed 2012-01-12

Contents

Copyright Notice

Preface

At present there is an important trend towards synergy between research in Artificial Intelligence and in Cognitive Systems. From the Artificial Intelligence point of view this represents a return to the original goal of Artificial Intelligence research, namely, the design of software artifacts that exhibit intelligence of a similar kind as what we observe in humans; not necessarily intelligence to the same extent, but of a similar kind. Several other branches of Artificial Intelligence (A.I.) study specific algorithmic or formal-logical techniques that are proposed to be relevant for the design of intelligent artifacts. Cognitive systems research, especially in Europe, has taken a holistic perspective on its topic using a multidisciplinary approach where Artificial Intelligence has been only one of the participating disciplines.

The Cognitive Systems approach in Artificial Intelligence takes the view that the current fragmentation of the field needs to be balanced by new efforts towards reintegration. One important step is to study *cognitive architectures,* ie. the overall structure and design of computational systems that exhibit human-level intelligence. The construction of such systems requires knowledge of the results from specialized areas in A.I., but it also requires an understanding of how such formal and mathematical results – and which of them – can be used in the overall design of systems with human-level intelligence, and of what other types of results are also needed, and of how the different parts can be made to work together.

The Cognitive Systems approach in Artificial Intelligence has two complementary aspects, therefore: techniques for *knowledge representation* that provide the information structures and the computational methods that are the building blocks of cognitive systems, and the design of the *systems architecture* for cognitive systems whereby the parts are assembled into a whole.

The present book is intended as the first volume in a series of a few moderately-sized books that introduce the Cognitive Systems approach in Artificial Intelligence. The current major textbooks and handbooks reflect the fragmentation of the field in the sense that they consist of a number of chapters that introduce separate topics, but which do not rely on each other and do not connect very well. I started writing this book series with a belief that it is possible to present Artificial Intelligence in a more coherent way.

With this approach it is important that the books shall present languages, architectures, and software tools that constitute the concretely available software technology for the construction of cognitive systems, and not only the algorithmic and logic-based methods. However, like in other parts of computer science there exists a variety of alternative languages and alter-

native softwares, which raises the question whether this book series should restrict itself to cover one of each kind, or whether it should strive to present an overview of all major alternatives?

The solution to this problem has been to separate the material between *main volumes* and *annex volumes,* where the main volumes show the overall picture using a generic view of architectures and one particular representation language, albeit a fairly conventional one. Each chapter in a main volume has a corresponding chapter in an annex volume, where the annex chapter contains an exposé of existing approaches and software solutions for the topic in the main chapter.

The reason for this arrangment is that it makes it possible to show the cross-connections between the various functions in a cognitive architecture as clearly as possible. Allowing catalogs of alternative designs for the same function in the main text could easily have confused the overall picture.

Another design choice for this work was whether it should aim to be a textbook or an academic handbook, in the sense of a systematic account of the field for use by professionals there. The difference is that a handbook should be quite concise, whereas a textbook is expected to be more easy to read, it should contain illustrations, problems to solve, solutions to the problems, and so forth. Moreover, a handbook does not need to contain an introduction to the field since its readers are supposed to be familiar with it already.

In the present case I felt that the concise style of a handbook is desirable, again in order to communicate the overall structure of its topic as well as possible. At the same time I could not omit the introductory material since to my knowledge there does not exist any textbook that takes a similar approach. The present text is therefore a hybrid between the two alternatives.

Hence, if the reader of this book is already familiar with the field of Artificial Intelligence, he or she will have to bear with introductory chapters that express what he/she already knows and takes for granted. It is my hope that these chapters will be of interest anyway as a new way of looking at the material. Similarly, if the book is used for a course in Artificial Intelligence, then it will have to be complemented with the additional student-friendly materials that are needed for that purpose. In due time this may be the topic for a second type of annex volumes, or for extended versions of the main volumes.

In the longer perspective it is foreseen to also add a third type of volumes besides main volumes and annexes, namely, volumes that provide in-depth accounts of topics that have been introduced in the main volumes, and based on the same overall framework. If this can be done, with the contributions of additional authors, then the goal of having a coherent handbook for the field can be achieved.

Given the structure of main volumes and annex volumes, it is natural as well to locate the historical overview and the references to original contributions in the (first kind of) annex volumes. Main volumes will frequently contain clickable links to websites containing related information, but literature references in the conventional sense are rare in the main volumes and will in principle be located in the annex volumes.

It is intended that annex volumes shall be able to evolve more rapidly than

main volumes, due to the successive addition of new material. While readers are certainly invited to contribute their comments and suggestions to all aspects of this work, there is a particular invitation to contribute suggestions for additional accounts of, and references to original work, for inclusion in the appropriate annex chapters.

The reader who compares this book series with standard textbooks in Artificial Intelligence will first of all notice the emphasis on basic knowledge representation principles and on architectures in Volume I, and the use of this material in Volume II and onwards. He should also notice the emphasis on precise definitions of basic concepts, in particular in Chapter 2 of Volume I, and the consistent use of these concepts in later chapters.

The establishment of a consistent terminology throughout the work means that the presentation of a particular topic sometimes ends up using other terms than what is customary in the current literature for that topic. This has been necessary when different branches of A.I. use different terms for the same thing, in which case I have chosen one of the terms for use throughout. It has also sometimes been necessary when different branches use the same term for different things, in which case I have tried to use separate terms in order to avoid confusion.

In addition, I have sometimes taken the liberty to substitute new terms in the place of odd, historically inherited terms such as *A-box, T-box,* and *version space.* However, in all cases where my terminology differs from the standard one there shall be a comment or footnote that clarifies the situation.

The unification of areas goes further than the mere unification of terminology, however. For the topic of Reasoning about Actions and Change, for example, the Handbook of Knowledge Representation [1] devotes separate and unrelated chapters to Time and Action Logic, the Event Calculus, and the Situation Calculus. Of course, Time and Action Logic and the modern Event Calculus have the same core, apart from their use of different nonmonotonic preference methods. The present work treats them together, but they are also integrated with the presentation of the Situation Calculus since the latter can be obtained from the former merely by the introduction of one function (the situation successor function) and one single axiom that characterizes this function. This connection is defined in Subsection 4.5.3. of the present volume.

Finally, there is considerable commonality between Action Planning and Reasoning about Actions and Change, and also between the latter field and Qualitative Reasoning. These commonalities are exploited in the present work, in particular in its Volume II. These are only some of the examples of the conceptual and textual economies that can be obtained through a coherent exposition of the various parts of classical Artificial Intelligence.

The present version of this volume is a preliminary one, and it is circulated to colleagues in the hope of obtaining comments and suggestions for how it may be improved, as well as how to proceed with the additional volumes. Major parts of Volume II have been written and although it is not yet ready for circulation, the table of contents for the manuscript for Volume II is available on the AICA website.

[1] Frank van Harmelen, Vladimir Lifschitz and Bruce Porter (eds): Handbook of Knowledge Representation. Elsevier, 2008.

Both main volumes and annex volumes will be available on-line and without charge from the website for this work, called the AICA website, which is

`http://www.ida.liu.se/ext/aica/`

Free availability will also apply to the finished work and not merely to the preliminary versions. The work is intended as an Open Educational Resource (OER) and it will be published (preliminary choice) under the GNU Free Documentation License. [2]

I have developed a software platform for cognitive systems, called the *Leonardo system,* which uses the same approach and the same notation as is used in this book series. One of the later volumes will describe Leonardo. Current information about Leonardo is available from its website, which is

`http://www.ida.liu.se/ext/leonardo/`

This system is of course also freely available and under the same license.

Both the present book sequence and the Leonardo system are the result of work in the Cognitive Autonomous Systems Laboratory (CASL) at Linköping University, Sweden. The website for this group (the term 'laboratory' should be interpreted as 'research group') is located at

`http://www.ida.liu.se/ext/casl/`

All comments and suggestions concerning this work are welcome and may be sent to my email address at:

`erik.sandewall@liu.se`

The approach and the goal of the present work is in line with several current initiatives both in Europe and in North America. I hope that it will be a useful contribution to these initiatives as well as to Artificial Intelligence in general.

Erik Sandewall

[2] http://www.gnu.org/licenses/fdl.txt

Chapter 1

The Word *Intelligence* as in *Artificial Intelligence*

1.1 Intelligence is a Graded Concept

The goal of research in Artificial Intelligence is to develop a technology whereby computers behave intelligently, for example:

- Intelligent behavior by simulated persons or other actors in a computer game
- Mobile robots in hostile or dangerous environments that behave intelligently
- Intelligent responses to user requests in information search systems such as Google or Wikipedia
- Process control systems that supervise complex processes and that are able to respond intelligently to a broad variety of functional failure and other unusual situations
- Intelligent execution of user requests in the operating system and the basic services (document handling, electronic mail) of standard computers.

The words "intelligent" and "intelligence" are then used in the same sense as they have in ordinary language. Several other scientific disciplines, such as psychology and philosophy, have developed fairly precise notions as well as considerable internal disagreement about what intelligence "really is" or what is the appropriate meaning of the term. This has influenced artificial intelligence research a bit, but not very much. Similarly, when an aeroplane designer says that a particular aircraft "flies," he or she realizes of course that the performance of the aircraft differs in fundamental ways from the flying of a bird or a bee, and he or she is not concerned about what is the "true meaning" of the concept of flying.

It is important to keep in mind that intelligence is a graded concept: a person may be more or less intelligent, and it is not possible to divide the world's population into one group of people that are intelligent and others that are not. Just like "hot" and "cold" describe the two ends of a

temperature scale, similarly "intelligent" and "stupid" describe the two ends of the intelligence scale. The behavior of conventional computer programs is manifestly at the stupidity end of that scale, [1] and the goal of artificial intelligence is to move software away from pure stupidity and in the direction of being more intelligent. This is a gradual process and one should not think of intelligence as a magical spirit or mystical power that can be placed in computers once and for all.

1.2 Scenarios with Children that Involve Intelligence

This section will characterize the kind of intelligence that artificial intelligence research is concerned with by describing a number of scenarios involving small children, referred to as Ronnie, Valerie and Gustaf, who exhibits a certain degree of intelligence. These are not invented stories, they are actual scenarios that I have observed in small persons, but they have been selected since they are good examples of the type of capabilities that artificial intelligence research addresses. In this book we shall show how these capabilities that one can observe when intelligence emerges in young children, are also relevant and needed when a certain level of intelligence is embedded in computer software.

1.2.1 The Freezer Door Scenario

Ronnie and her older brothers like icecream, and their parents allow them to go and get icecream bars by themselves in the kitchen freezer but only after having received permission for this on each particular occasion. This time the brothers are away and Ronnie is allowed to pick up an icecream bar herself. The freezer door is too tall for her to reach the handle at the top of the door, so she runs to the nearest chair in the kitchen, pushes it to a position in front of the refrigerator that is next to the freezer, climbs onto the chair, and opens the freezer door. This is her own idea at this occasion, since usually it is one of her brothers that opens the freezer door without needing a chair and that is able to get icecream bars for all the children.

This scenario involves several aspects of intelligence. First of all, Ronnie understood that she needed an *instrument* that made it possible for her to open the freezer door. She was able to select an appropriate instrument and to operate it for the intended purpose.

Moreover, Ronnie made a smart move by placing the chair in front of the refrigerator door, and not in front of the freezer door which was after all at the center of her attention. If she had put it in front of the freezer door then it would have blocked that door so that she could not open it. Similarly, if she had put it to the right of the freezer door, instead of to the left, then

[1] Examples of the utter stupidity of ordinary computer software can be found in the so-called balloons with messages to the user that are produced in particular in the Windows environment. For example, "Updates are ready for your computer. Click here to download these updates" even if there is no Internet connection at the time. Or "Wireless network is not connected" which is obtained as a message even when the *wired* network is already connected.

she would maybe have been able to open that door, but she would not have been able to reach the icecream bars.

How did Ronnie have this insight, and how could a computer program (in a computer game, or in a household robot, for example) have the same insight? Given that this was a new situation that Ronnie had not encountered before, the most plausible explanation is that she was able to *simulate* her successive actions in her own mind before starting out, or maybe while she was already pushing the chair, and as a way of anticipating the effects of her continued actions. If she first imagined pushing the chair to the position in front of the freezer door, the simulation would show how she climbs the chair and tries to open the freezer door but fails, and then she is able to revise the plan before it has taken effect.

In artificial intelligence terms, Ronnie has showed the capability of *planning, anticipating* the effects of a proposed plan, and *plan revision* where her first plan was modified in order to pass the anticipation test, and of course also *plan execution* where the selected plan is performed.

1.2.2 The Revised Freezer Door Scenario

Now let us modify the scenario so that it applies to Ronnie's friend, Valerie, who is a little younger and not yet as smart. Valerie actually pushes the chair to the place in front of the freezer door. She climbs the chair, tries to open the freezer door but fails, climbs off the chair, moves it to the position in front of the refrigerator door instead, climbs it again, opens the freezer door, and gets the icecream bar.

In this case Valerie had not been able to anticipate the full effects of her original plan so that the execution of that plan failed. This called for another important aspect of intelligent behavior, as understood in artificial intelligence, namely for *diagnosis* which means finding out the reason for why something has gone wrong: a plan does not work, or an available device does not work properly. Diagnosis leads to making a new plan, where typically some of the effects of the original plan are undone and a new plan is made.

Notice an important difference between this scenario and the first one: In the original scenario Ronnie may first have thought of moving the chair to in front of the freezer door, and then changed to a better plan, but the revised plan *did not* consist of first moving the chair to the freezer door, climbing it, unclimbing it, and moving it to the refrigerator door. In other words, the plan did not replicate the planning process; it was a better plan that took the obstacles into account.

Now suppose Valerie had performed once according to the revised scenario, and the next day the situation is repeated: Valerie is allowed to pick up an icecream bar, and she has to do it herself. She is no smarter than the day before, so if her anticipation capability was not sufficient yesterday, there is no reason that it should be so today. However, this time Valerie makes the right move and takes the chair to the refrigerator door. Why? - because she has *learnt* from the experience of yesterday. The scenario involving the plan failure and plan revision was retained in her memory, and she was able to use it for doing the right thing. However, she certainly did not repeat yesterday's sequence of action, so she did more than just

repeating a memorized sequence of actions: she *revised* yesterday's plan so as to eliminate the false start, obtaining the same plan as was obtained by Ronnie in the original scenario.

Moreover, it turned out that the second day all the chairs were out of the kitchen, so Valerie was not able to apply the revised plan. There were two possibilities: find something else to climb on, or get a chair from the next room. Valerie used the second possibility and got her icecream bar. Plan revision is therefore used not merely for simplifying away false starts from a plan in plan memory; it is also used for modifying a plan to fit the current situation.

The technique of maintaining a "knowledgebase" of problems that one has encountered before as well as their solutions, and to adapt them and apply them to forthcoming situations is an important one in artificial intelligence, and it goes by the name of *case-based* techniques.

1.2.3 The Doll Problem Scenario

This day Valerie is visiting Ronnie and the two girls are playing. They are not playing together; each of them is doing her own thing. Ronnie is putting her two dolls to bed, which involves first putting diapers on them. She is busy with one doll (Ada) when Valerie comes into the room, a comb in her hand since she has just been combing the hair of one of her own dolls. Valerie notices that Ronnie's other doll also needs to have her hair combed, and picks her up. Father observes the situation but decides not to intervene, but to watch the scene to see what will happen.

Ronnie also notices the situation of course, frowns, and goes to the sofa and picks up Valerie's beloved little handbag, and brings it to Valerie. Valerie receives the handbag, lets go of Beda, and Ronnie proceeds to putting diapers on Beda as well.

This scenario is very similar to the freezer door scenarios in some ways, and very dissimilar in other ways. A major difference is that here we are dealing with social actions and their effects on the state of mind of persons, whereas the previous scenarios only concerned physical things: moving chairs, climbing onto them, opening freezer doors. However, in spite of these differences there is also an important similarity: both scenarios are concerned with *actions and their effects,* and with *making a plan* that is expected to achieve effects that are desired by the person in question.

The making and the execution of the plan can be thought of as *reasoning:* one thinks about different possibilities, different combinations of actions, and all aspects of the reasoning can be expressed clearly and it can even be represented as formulas in formal logic. However, this reasoning aspect is accompanied by other aspects that do not have the same rational character, and which may be considered as more or less intuitive when performed by a person. The freezer door scenario required a sense of positions and movements in the three-dimensional world of the kitchen; the doll problem scenario required Ronnie to have an understanding of Valerie's likely response to her plan. In fact, the success of the plan may well have depended on a positive understanding and a will to compromise on Valerie's side, and if Ronnie understood this then she might have responded differently if it had been one of her brothers that had taken the doll, rather than Valerie.

Is Ronnie's problem-solving competence in the doll problem scenario the *same capability* as her problem-solving competence in the freezer door scenario? Is it reasonable to refer to both of them using the same term, such as "intelligence," or should one refer to one as "mechanical intelligence" and the other as "social intelligence," for example, and should one consider them as two different phenomena each with its own characteristics?

This question is not very important from the point of view of artificial intelligence, since both types of scenarios are included in the topic of interest of artificial intelligence as a field. The question whether it is essentially the same capability or essentially different capabilities will then be answered on technical grounds, by comparing the software solutions for different aspects of intelligence in order to see how much the differ or overlap.

If one should ask the same question for intelligence in people, then there is little chance of obtaining a definite answer. It is not possible to extract "the intelligence" from a human person in order to inspect it and analyze it, of course, so there is not direct way of saying what (human) intelligence *really is*. The use of the term intelligence is a linguistic convention, and one may advance various kinds of arguments in favor of saying that there is just one kind of intelligence, or there are several kinds.

1.2.4 The Bath Assistance Scenario

The third scenario takes place when Ronnie is a bit older and is able to speak, although this is only since a few months. Father is watching Ronnie as she plays with her favorite doll, Ada, and now she has taken off the doll's clothes. Guessing what she has in mind, he asks "are you going to give Ada a bath?" Ronnie answers yes; father says "are you going to take her to the bathroom then?" Ronnie answers yes again, and waits. Father waits. Ronnie says: "Come you also, I can not myself open the water." The reason for this is that although Ronnie can easily reach into the wash basin, the faucet handle is located in such a position that she can not reach it.

This scenario resembles the freezer door scenario since there is an instrumental aspect. Just as Ronnie realized that she needed a chair in order to open the freezer door, she now realizes that she needs father's help in order to "open the water." She knows that he is able to do that since she has seen him do it before, or since it is one of the things that adults in general are able to do according to her experience. She is also able to use the powerful medium of language in order to get him to move to the bathroom.

However, besides this instrumental side of the scenario, there is also a linguistic side: How did Ronnie formulate the phrase "Come you also, I can not myself open the water." It is fair to assume that there is a large portion of case-based reasoning in this case as well, since we learn language – children do, in particular – by listening to many examples of language, imitating them, and adapting them to new situations, for example by putting together sentences in new ways.

The phrase "open the water" can not be understood merely as a memorized phrase or a rearrangement, however, since it is not the normal way of expressing oneself in English. One must assume that Ronnie has a notion of the operation of "opening" that she has learnt and used in the context of

opening doors, drawers and boxes, and that she can apply it by analogy to the action of allowing the stream of water to flow into the wash-basin.

Furthermore, notice that as Ronnie wanted father to come with her, she did not say exactly what she wanted; she did not say "come you also and open the water for me." Instead, she provided an explanation why father's assistance was needed, in order to explain why she could not perform the task herself. This was all she needed to say; she could assume that this explanation was sufficient for father to understand that his assistance was needed and that she wanted him to come along.

Like in the doll problem scenario, this interaction must be understood by an analysis of the actions and the reactions of both the people involved. When father hears "I can not myself open the water," and when he has understood what Ronnie means by "open the water," nominally he has just received a piece of information about the state of the world, and this information he probably knew already. If you think of linguistic utterances as means of transmitting information, questions and requests, then it was a redundant utterance. However, in the given context it was clear what was Ronnie's intention when saying this: besides being a piece of fact, it also served a purpose namely being the *motivation for* something. Understanding language is not merely understanding immediate content, it is also understanding the reasons for why phrases are being said, and those reasons are often the real meaning of the utterances.

This kind of analysis of dialogs is the subject of *speech act theory* which has aspects of both linguistics, psychology and philosophy. Speech act theory is also used in artificial intelligence, both for the design of dialog software and as the basis of techniques for message exchange between autonomous agents. The latter topic is addressed in Chapter 2.

1.2.5 The Swimming Scenario

Ronnie's older brother, Gustav, is just learning to swim. He is standing eagerly at the side of the swimming pool, jumps into the water and makes it across the pool to the other side, although it's clear that this is close to the limit of his ability. After only a moment's hesitation he turns around and swims back to the first side, where his grandfather waits and compliments him for the accomplishment. Grandfather also says: "let us go over to your father and tell him that you have crossed the pool by yourself," but Gustav answers "No! I want to show him." He goes and asks his father to come, and swims across the pool a third time.

What is the mechanism that leads Gustav to decide to swim back at once, and what is the mechanism that leads him to not accept the suggestion to just go and tell his father what he has just done? A few possible answers can easily be rejected. His behavior can obviously not be explained merely in terms of minimization of effort. The behavior of this kind can also not be explained only in terms of emotional reward and encouragement, since one must assume that the child also takes his own physical limitations into account and does not attempt something that is clearly impossible or too painful.

The previous examples concerned how to achieve a specific goal, but in this scenario the question is rather how the goal was selected, whereas when this

decision had been made it was quite clear how to go about it. Anticipation of the effects of the selected actions would seem to be of marginal importance compared to the previous examples.

One may have different opinions about whether Gustav's decisions in this scenario are manifestations of 'intelligence' or not, but in any case it is clear that this kind of *spontaneous decisions* are very common in human cognitive behavior, so that at least they coexist with intelligent deliberation and interact with it. For example, the common wisdom that you should "think before you act" can be understood as a recommendation to moderate spontaneous decisions with a certain amount of deliberation before action is taken, in particular, deliberation about the likely consequences of the act that was one's "first thought."

It is therefore natural to include a software mechanism for producing spontaneous decisions in an architecture for intelligent behavior in computers. The major candidate for this purpose is the technique of *decision trees* which will be described in Volume II of the present book series.

1.2.6 Can Intelligence be Implemented?

The examples in this section represent relatively simple examples of some important aspects of intelligence in the sense that this term is used in artificial intelligence: planning, anticipation, plan revision, diagnosis, case-based reasoning, learning, linguistic competence. Several of these require an underlying capability of *imagining* other situations than the one that the person is in at present, that is, creating mental structures whereby different situations can be compared and potentially forthcoming situations can influence immediate behavior. A software system that implements similar capabilities must therefore have a way of creating and using a mental *model world* where alternative possible actions and alternative possible explanations for observed phenomena can be tested and evaluated.

Several of these capabilities have the character of mental *deliberation* which bears many similarities with *formal reasoning methods* that are studied in the field of Formal Logic as well as in Artificial Intelligence. It is a topic of debate and disagreement to what extent logic-based methods are sufficient and to what extent other types of computational methods are also required, but in any case it is clear that both are needed.

For example, the decisions that Gustaf made in the Swimming Scenario can hardly be considered as the result of logic-based deliberation. Both his decisions were made instantly and without any indication of step-by-step reasoning activity. To the extent that one can speculate about it, it would seem that a certain number of factors must have contributed to his decisions, but there are good reasons to think that the combination of these factors had a quantitative aspect rather than just binary choices.

Implementing capabilities such as the ones described here is a complicated task, especially when one proceeds to situations that are larger in the sense of containing many more persons and aspects. Practical implementations also tend to be further complicated by the requirements of the "non-intelligence" aspects of the entire system, such implementing an "understanding" of the movements of objects in a three-dimensional world, in

such a way that it can be effectively connected to the core intelligence part of the system.

However, although it is complex, this task is by no means impossible. There is no need for magic in it; all the subtasks can be approached, and in particular since we think of intelligence as a graded concept with "stupid" at one end and "intelligent" at the other, there is not need to feel overwhelmed by the daunting task of creating human adult-level intelligence; we can proceed gradually. This is what research in artificial intelligence is about.

1.3 Scenarios with Grownups that Involve Intelligence

The examples of intelligence in the previous section can be exhibited by small children. What additional aspects of intelligence do we develop during the continued years until adulthood, and to what extent can they also be implemented in computer software?

A comprehensive answer to that question would require much more space than what can be devoted to it here, and we shall only give an example that illustrates one aspect of an answer.

1.3.1 The Party Preparation Scenario

Consider the task of preparing your livingroom so that it is ready when your guests arrive, the assumption being that you will start there with a drink and then go elsewhere for the meal. Two main activities are involved: the activity of preparing the livingroom, and the activity of entertaining the guests. The latter activity is only partly known, since surprises may come up (one guest may report an allergy that you were not aware of before, another guest may fall and hurt himself, and so on) but still you have sufficient knowledge about it that you can determine what needs to be done in advance. This involves bringing a number of things to the livingroom, such as beverages, glasses, napkins, probably a corkscrew and a bottle opener, and whatever you will offer to eat or to nibble. This knowledge is sufficient for making a plan for the preparation phase and for carrying it out. Contingencies may arise, not only in the entertaining phase but also in the preparation phase, but these have to be taken care of if and when they happen since you can not plan for everything.

There are a number of analogous scenarios in ordinary life, such as putting things in order in the kitchen before cooking according to a complicated recipe, putting a conference room in order for a meeting, or preparing a surgery room for an operation.

Intelligent behavior in such preparation scenarios require the same capabilities as were discussed in the previous section. One difference is however that the capability of imagining alternative situations is put to much bigger test in these preparation scenarios, compared to what was needed for the freezer door scenario or the doll problem scenario. However, this is arguably only a difference in scale.

Suppose one wished to implement a software system that could perform the party preparation task in the scenario just described. How should this problem be approached? One must assume that the following information is available to the system at the outset, besides a description of the environment as a whole:

- A description of the performance stage (e.g., the entertaining activity in the initial example) that can be executed in the model world. This will be a nondeterministic action sequence or a set of typical, deterministic action sequences. It does not span the entire set of possible developments in the performance stage, but it can generate a number of typical examples.
- A specification of the place where the performance stage is to occur.
- A specification of the objects that must be available in that place before the beginning of the performance stage, and of any location constraints for these objects within the place of performance.
- A specification of the present location of objects that may be needed, or of actions that can produce them if they are not presently available in the home. Those actions may for example include calling a delivery service.
- A specification of the actions that can be used for moving objects from their present location to the place of performance.

The following procedure may produce an initial plan hypothesis for the preparation stage:

- Identify the set of objects that will be needed and their place at the time of performance.
- Identify their present location.
- Identify move actions that will get each required object to the required place.
- Aggregate the move actions, for example using actions that can take several actions at the same time, instead of one by one. This also includes e.g. using a tray or a cart in order to be able to bring several objects at the same time.

Then check out the preparation plan in the following way:

- Make a mental construction of an *imagined situation* ie. a model world in the sense of an earlier section here.
- Execute the aggregated tentative plan in lookahead mode in the model world. If it fails at some point, for example because some action is determined not to be applicable, then revise the plan and iterate.
- When the tentative plan executes correctly in the model world, then execute the performance stage as well in the model world and verify that it works correctly. If it fails at some point, then consider two possibilities: revising the preparation plan, or postponing the problem until it really occurs (if it is going to occur) in the performance stage.

- Execute the preparation plan in the outside world. If problems arise in spite of the checkout in the preceding steps then assess the situation at hand, and revise the plan.

Some activities recur several times in these steps and one can in fact identify the following three major types of activities in them:

- Initial analysis of the situation in order to identify what objects are needed and where they are presently located.
- Making a plan, and modifying an existing plan according to information about some situation where it fails.
- Robust execution of a script, i.e. a sequence of actions, which means executing it in such a way as to respond adequately if some of the actions fail. This includes both the test-execution of a plan in the model world, test-execution of the execution-phase script in the model world, and execution of these in the outside world.

These three kinds of activities have one thing in common, namely, they are based on the use of a *goal* . The first activity identifies a goal, the second activity finds or changes a plan for achieving a goal, and the third activity relies on the available goals when there is a problem, and also when there is an uncertainty about exactly how to execute an action in the script at hand.

One may object that the script for the performance stage is not accompanied by a goal: it just specifies what may happen during that stage. However, the concept of a "goal" must be understood in two ways. There can be specific goals, such as "cool these beer bottles in time for the arrival of the guests," but there can also be standing goals such as "make sure that the glasses of all guests are at least half-filled unless the guest has declined a refill." In that sense there are goals even for the performance stage.

1.4 Non-Deliberating Intelligence

The intelligence scenario in Section 1.3 must be understood as an example of *deliberation:* the person performing the party planning thinks systematically (more or less) and rationally (more or less) about the task at hand. This deliberation can to some extent be observed by introspection, and work in Artificial Intelligence sometimes uses introspection about one's own deliberative behavior in order to obtain suggestions for how to organize software performing similar cognitive functions.

This leads to an obvious danger of methodological bias, namely: if actual cognitive activity consists of *both* deliberation that is accessible to introspection, at least to some extent, and *also* of spontaneous cognitive behavior like in the Swimming Scenario above, which is very little accessible to observation by introspection, then there is a risk that the role of deliberative activity is over-emphasized and the role of spontaneous activity is under-emphasized.

In order to compensate for this risk it is important to use other sources of information about cognitive behavior as well, and in particular about the mechanisms for spontaneous decisions. (Other types of non-deliberating

behavior include biologically based *reflex actions* and mentally based *trained rapid responses,* but they are outside the topic of the present discussion). One line of research concerning spontaneous decisions proposes that these may be the results of combining values for a limited number of *features* of the situation at hand, to the order of 10 or 20 of them. The computer implmentation of such a decision process will therefore involve mechanisms for the selection of relevant sets of features, processes for obtaining the current values for each of them, and a process for combining the obtained values so as to obtain a decision.

This line of research has been described in an easily accessible book [2] that describes examples of this kind of decision as well as what is known about the cognitive processes that make them possible. Many of the examples in this book concern decisions of major importance, such as determining whether a particular classical painting is a fake or not. However, it is also easy to imagine how the same mechanism could be used for modelling Gustaf's decisions in the Swimming Scenario, and more generally for modelling many of the very large numbers of minor decisions that we make in daily life.

It is also plausible that spontaneous decisions of the this kind are used as components for deliberating actions. In terms of the scenarios in Section 1.2, how did Ronnie decide to use a chair as an instrument for being able to open the freezer door? How did she know that she could push the chair? How did she decide on how to grip the chair with her hands so that she could climb onto it? These are a multitude of small decisions without which the deliberating behavior would not be possible, but they can not all be understood in terms of deliberating behavior, compiled or not.

Artificial intelligence systems must therefore also contain aspects or components that can make spontaneous judgements. The use of *decision trees* is one of the techniques for this purpose.

With respect to terminology, should spontaneous judgements be considered as an integrating aspect of intelligence, or as another kind of intelligence besides ordinary intelligence, or as a separate phenomenon that does not go by the name of intelligence at all? This is largely a matter of definition, since there is no objective way of extracting the intelligence out of a person and inspecting it for its contents. Some people argue strongly that one should identify different kinds of intelligence, and there may be certain empirical arguments in favor of one or another position in this respect. However, from the point of view of Artificial Intelligence this question does not matter much; one must merely realize that A.I. systems must include methods for both spontaneous judgements and compilation of deliberating behavior.

1.5 The Intelligence Concept in Psychology

Let us now return to the question that was briefly addressed in Section 1.2: is it reasonable to consider intelligence as one concept or as several? For example, should Ronnie's ability to open the freezer door be considered as a manifestation of the same competence as her ability to resolve the doll problem with Valerie, or should these be considered as two different competences? As already stated, this question is of marginal importance for

[2] Malcolm Gladwell: *Blink.* Penguin Books, 2006

artificial intelligence, but it may be more important for cognitive psychology, that is, for that branch of the science of psychology where intelligence is a major topic of study.

This question is in fact deeply controversial, and the reader is encouraged to visit the Wikipedia page for "intelligence" to get a first impression of the divisions. Besides reading the article itself, it is also interesting to read the "discussion" section that is attached to that page, and the archived earlier discussions. The following is a brief excerpt from that page:

Intelligence derives from the Latin verb *intellegere;* per that rationale, "understanding" (intelligence) is different from being "smart" (capable of adapting to the environment). Scientists have proposed two major "consensus" definitions of intelligence:

(i) from *Mainstream Science on Intelligence* (1994), a report by fifty-two researchers:

A very general mental capability that, among other things, involves the ability to reason, plan, solve problems, think abstractly, comprehend complex ideas, learn quickly and learn from experience. It is not merely book learning, a narrow academic skill, or test-taking smarts. Rather, it reflects a broader and deeper capability for comprehending our surroundings catching on, making sense of things, or figuring out what to do.

(ii) from Intelligence: Knowns and Unknowns (1995), a report published by the Board of Scientific Affairs of the American Psychological Association:

Individuals differ from one another in their ability to understand complex ideas, to adapt effectively to the environment, to learn from experience, to engage in various forms of reasoning, [and] to overcome obstacles by taking thought. Although these individual differences can be substantial, they are never entirely consistent: a given person's intellectual performance will vary on different occasions, in different domains, as judged by different criteria. Concepts of "intelligence" are attempts to clarify and organize this complex set of phenomena. Although considerable clarity has been achieved in some areas, no such conceptualization has yet answered all the important questions, and none commands universal assent. Indeed, when two dozen prominent theorists were recently asked to define intelligence, they gave two dozen, somewhat different, definitions.

These definitions emphasize what almost everyone can agree on, which means that they do not address the hard questions where opinions differ. As a framework for understanding the disagreements, notice first of all that intelligence is considered as a combination of a number of more specific capabilities *each of which can be measured by suitably designed tests* whereas for the general concept of intelligence as such it is much less clear how one would design a single test. An important question is therefore whether, and for what reasons is it appropriate to bind together a number of separate, measured capabilities and apply the single term "intelligence" to the combination of them.

Several approaches to that question may be considered. If it were possible to identify a region of the human brain that is active in all these separate capabilities and inactive otherwise, then that would be a very concrete rea-

son for assuming the existence of a general intelligence facility in humans. Unfortunately, however, no such brain region has been identified and available evidence suggests that the contrary is true, and that different brain regions are active for different purposes. On the other hand, this does not prove that intelligence should not be considered as a single concept: it could be a single competence that is realized in a distributed way in the brain.

Another approach is to consider *co-occurrence* of the specific capabilities that are mentioned in the definition of intelligence. Using the statistical data from tests for specific capabilities in a large number of people, one can check to what extent the presence of one capability is related to the presence of another capability in the same person. A strong degree of co-occurrence of all the capabilities, or of a group of some of them, will be a plausible reason for assigning a name to that group and to study it as an entity.

In principle, strong co-occurrence between several capabilities may be taken as an indication that there is a common underlying mechanism that produces all of them. However there are also a number of other possible explanations, including neural-level explanations (maybe the average number of connections from each brain cell to neighboring brain cells is important for all of these capabilities, and maybe this is the underlying factor explaining an observed co-occurrence?), genome-level explanations (maybe particular aspects of the human genome influence the development of these capabilities in the fetus?), and nutritional and social explanations. It may or may not be possible to identify which of these are plausible explanations for observed facts, but even in the absence of an agreed explanation it can anyway be useful to have a name for a bundle of co-occurring capabilities. For example, given that intelligence as measured by IQ tests appears to be a usable predictor for "success in life," it may be useful to have it even if one does not understand the underlying mechanism.

Besides observed co-occurrence, there are also a number of other ways of studying the relationship between identifiable capabilities, in particular by studying individuals with unusual properties: individuals that are extremely skillfull in some particular capability but not in others, or individuals that have suffered brain damage that strongly impairs some of the capabilities. This may shed light on whether and to what extent there is a common basis for several of the capabilities.

Chapter 2

Software Architecture and Cognitive Architecture

The design of a software system of nontrivial size and complexity requires a clear definition of the system's *architecture,* that is, the major data structures, the most important procedures and algorithms that are used in the system, and the invocation paths and dataflow paths. The present chapter will address the architecture of systems that exhibit a certain degree of intelligence, with particular emphasis on systems whose behaviors are similar or analogous to the human behaviors that were described in Chapter 1.

This emphasis is made in order to distinguish our topic from the design of systems with *specialized artificial intelligence,* that is, systems that are designed for a particular task and where the intelligent behavior of the system is restricted to what is needed for that restricted task. We shall use the term *cognitive agent* for the kind of software artifact that we are addressing, the term *Cognitive Systems* for how it is used in Artificial Intelligence, and the term *cognitive artificial intelligence* (CAI) for the characteristic property of such cognitive agents. [1] The term cognitive agent will obtain a precise definition later on in this chapter.

The present chapter discusses architecture on two levels, namely, the *cognitive architecture* which is one of the main issues for the present work, and then the *software architecture* which is an underlying level that serves as a platform for the cognitive architecture. A number of cognitive architectures have been developed, first in an explorative fashion and later on for regular use, both in research and in applications. Sections 2.1 and 2.2 in this chapter will describe the principles that are usually used in these architectures: Section 2.1 introduces and defines the basic concepts and Section 2.2 describes component structures and functional processes in cognitive architectures. An overview of specific such architectures is found in the Annex of this chapter. Section 2.3 introduces the concept of *software individuals* from a technical and philosophical point of view; this is a reflection of my own interests and approach. Section 2.4, finally, begins to address the topic of the underlying software architecture, although this topic will be continued in Chapter 6. The reason for beginning on that topic here is to introduce

[1] The term *Artificial General Intelligence* is also in use, but the area of research that it designates has a broader scope.

material that is useful for understanding the next following chapters.

2.1 Agents and Agent-Related Concepts

The term *agent* is frequently used in the context of intelligent systems, usually with a qualification such as *autonomous agent, intelligent agent,* or *cognitive agent.* It is a suggestive term that captures one important intuition that we have about an intelligent system, namely, that it shall be able to decide on its own actions, maybe related to our notion of *free will.* However, the term 'agent' is used in several branches of computing and for a fairly broad variety of purposes, which means that it is defined differently by different authors. We shall therefore base our definitions on an established concept with a more precise definition.

We use the standard definition of an *automaton with outputs* which is as follows. It is a system that is characterized by a set of possible inputs (sometimes called the input alphabet), a set of possible outputs, a set of possible states, and a transition function. The automaton defines a cyclic process where in each cycle it is in a particular state and receives one of the possible inputs, and as a result it switches to another one of the states (or stays in the same state) and produces one of the possible outputs. The transition function specifies the new state and the current output as a function of the previous state and the current input. Therefore, if an initial state and a sequence of inputs is defined for an automaton, then it is well defined what is the resulting sequence of outputs. [2]

The particular case of *finite state automata* is studied in other branches of computer science, together with a number of other types of automata. However, for the purpose of Artificial Intelligence we are interested in automata with quite different properties, and in general with automata where the possible states are complex structures which may be understood as data structures or as collections of formulas, and where the transition function is defined as a program operating on those structures. Such automata will be called *software automata;* their states will be referred to as their *internal states.*

Whereas the transition function for an automaton is usually defined as a function that performs *one cycle* of the automaton's cyclic behavior, the program for a software automaton is usually written so as to include the cyclic behavior as its *main cycle,* i.e. its top-level loop.

A software automaton is said to *have autonomy* and to *be autonomous* if its behavior is organized in terms of *actions* and it is able to make an active choice of action in each cycle, based on its current internal state. For example, consider an automaton that identifies, in each cycle, a set of possible actions that it may execute, evaluates the merit of each of them using a separate action evaluation function, and then executes the action that obtained the highest merit. In this case we would say that the automaton has autonomy.

[2] The output part of the definition is omitted in many uses of automata, and automata with outputs are also called *transducers.* In the case of Artificial Intelligence the state of the automaton is at the center of attention, rather than the output, and we shall therefore prefer to use the term automaton.

There may also be other ways of organizing an automaton so that it has autonomy, but this was one possible way. Regardless of the exact approach, it is an important aspect that the mechanism for the choice of action should not be completely fixed, like in a conventional computer program. It shall be possible for the action choice criteria to change in response to events during the earlier history of the automaton.

The concept of an *environment* for an automaton is defined next. The intended notion is as follows. The automaton performs its cyclic behavior in interaction with an environment which can be in any of a number of states, called the *external states* of the automaton. There is a *perception function* that maps external states to inputs for the automaton, and an *effectuation function* that maps an external state and an automaton output to a (new) external state. The tandem consisting of the environment and the automaton proceeds cyclically, where in each cycle the automaton receives the perception of the current external state and applies its transition function, and the automaton's output is used by the effectuation function to update the external state.

The formal definition may be simplified by incorporating the perception and effectuation functions in the automaton, or in its environment. If the automaton is thought of as a physical robot then it is natural to consider the perception function as a part of the automaton, and the effectuation function as a part of the environment (or to split it so that one part is in the automaton and the other part in the environment).

It is also natural to allow that the environment can change state, in discrete steps or continuously, even between the successive cycles of the automaton, but from a technical point of view this can be represented as an aspect of the effectuation function.

We shall return to the perception and effectuation functions in the context of robots which is in a later volume of this book series. At present we simply define the environment of a software agent as the set of its possible external states. An environment where the states are complex structures will be called a *structured environment.*

An autonomous automaton is said to be *cognitive* if it has a structured environment, and if its internal state, which it uses for selecting actions, has a qualitative model [3] of the environment as a significant part. A cognitive automaton is also said to be a *cognitive agent.*

In formal terms it would be possible for the model of the external state to be equal to the external state itself, in which case the perception function must be the identity function, but in all interesting cases the model is a much smaller structure than the external state and it is used as an approximation of the latter. The model of the environment may be a hybrid model containing both qualitative and quantitative parts, in particular when control engineering is involved, but if the model is *only* quantitative, for example in the form of differential equations, then we would not call the agent in question a cognitive agent.

Notice therefore that the definition of cognitive agent as a synonym of cog-

[3] To be exact, if it has *a representation of* a qualitative model of the automaton's environment. The distinction between a representation of a model and the model itself will be introduced in Chapter 4 and can be disregarded at this point.

nitive automaton does not depend on a definition of "agents." This is intentional since the concept of agent is used so broadly in the literature. For example, the Wikipedia article for "agent, in computer science and artificial intelligence" refers to "intelligent agent" as an "autonomous entity which observes and acts upon an environment and directs its activity towards achieving goals," and the article for intelligent agent, in turn, mentions a thermostat as an example of an intelligent agent. The main article also refers to "user agent" as "the client application used with a particular network protocol." [4] These examples illustrate that the agent concept can be used for almost anything that is capable of influencing its environment, physically or computationally, so that in whatever context this term is used one has to identify what precise definition it has been given there.

The mechanism for identifying possible actions and choosing between them can be organized in a variety of ways. The following are some of the major alternatives, besides the approach that was described above:

- The internal state may contain a *current plan* and in each cycle the agent executes the next step in the current plan, but it is also able to create and adopt new plans for itself in order to respond adequately to the situations that it encounters.
- The internal state may contain a set of *current intentions,* that is, things that the system is committed to doing sooner or later, and in each cycle it identifies possible ways of realizing some of the intentions, evaluates the merit and the cost of each of these possible immediate actions, and picks the best one (according to its own criteria) for immediate execution.
- The internal state contains a modifiable set of *production rules* each of which has the form "under the following conditions ...do the following ..." In each cycle the agent checks which of the production rules are applicable given the current input and current internal state, and executes one or more of the applicable rules.

These possible designs for a cognitive agent are plausible approaches to the design of cognitive artificial intelligence. They can easily be extended in order to accomodate a variety of things that we associate with the notion of intelligence. Learning can be understood as extending or modifying the set of production rules. Goal-directed behavior can be understood as a mechanism that maintains the agent's goals in the internal state, and that identifies alternative plans or alternative actions that may contribute towards these goals, for example.

2.2 The Cognitive Architecture

In order to make the notion of cognitive agent more precise one must consider what shall be the character of the main cycle, what shall be the structure of the internal state, and what shall be in the library of capabilities that can be invoked by the main cycle. A choice of design in these respects is called a *cognitive architecture;* it specifies those aspects of a cognitive

[4] These quotations refer to the Wikipedia contents in October, 2011.

agent that are constant over time and across different application domains. These aspects typically include the following [5]

- the short-term and long-term memories that store content about the agent's beliefs, goals, and knowledge
- the representation of elements that are contained in these memories and their organization into larger-scale mental structures
- the functional processes that operate on these structures, including the performance mechanisms that utilize them and the learning mechanisms that alter them.

The representation aspect will be the main topic for the following chapters in the present volume. The present section will discuss the functional processes or *capabilities* that are required in a cognitive architecture, beginning with some concepts that are used for defining the capabilities, and proceeding with a capability catalog. The concluding subsection will briefly discuss the question of memory structure.

2.2.1 Actions and Plans

The notion of an action is fundamental to our definition of a cognitive agent, and it will be be used more precisely as follows. First of all, we distinguish between an *action* and a *process* where a process is always an instance of an action. Consider for example the action "starting the engine of a car." If a particular person performs this action on three successive mornings, then you have three processes, all of which are instances of one and the same action.

Actions may be expressed more or less specifically. The action "starting the engine of a BMW car" is more specific than, i.e. it is *subsumed by* the action mentioned above. If an action B is subsumed by a more general action A, then every instance of B is also an instance of A.

Actions are represented as composite expressions consisting of an *action verb* and *arguments* or *parameters* for the action.

A *plan* is a composite expression that specifies a number of actions, the order in which they shall be executed, and conditions that restrict their execution.

Actions may be *elementary* or *composite* according to their action verb. A composite action verb is defined by a *plan script* that specifies a number of more detailed actions that shall be performed in order to execute an action with the given verb. A plan script is like a plan except that it can refer to, and make use of the arguments and parameters of the action invoking the script. An elementary action verb must instead have an attached procedure in a conventional programming language that is executed in order to perform an action with that verb.

A plan consisting only of elementary actions is called *sequential* iff the order on the plan steps is total. An arbitrary plan is called sequential iff the order

[5] Pat Langley, John E. Laird and Seth Rogers. Cognitive Architectures: Research Issues and Challenges. *Cognitive Systems Research,* Vol. 10, No. 2. (June 2009), pp. 141-160.

on the plan steps is total and the plan scripts for all the composite actions in it are also sequential.

Some systems and approaches include additional information in plans, for example, information about the timing and duration of actions and sub-plans.

2.2.2 Knowledge Contents of the Internal State

Different designs for a cognitive architecture may differ with respect to what kinds of components are included in the internal state of an agent, but some structures are widely recognized and used. The following terms are used in particular in the context of the *BDI Architecture* (Belief - Desire - Intention) and they are appropriate for other cognitive architectures as well. [6]

Beliefs: Beliefs represent the informational state of the agent, in other words its beliefs about the world (including itself and other agents). Using the term belief rather than knowledge recognizes that what an agent believes may not necessarily be true (and in fact may change in the future).

Beliefbase: The collection of beliefs in the agent's state is called its belief-base or its beliefset.

Desires: Desires represent the motivational state of the agent. They represent objectives or situations that the agent would like to accomplish or bring about.

Goals: A goal is a desire that has been adopted for active pursuit by the agent. Usage of the term 'goals' adds the further restriction that the set of current goals must be *consistent.* For example, one should not have concurrent goals to go to a party and to stay at home - even though they could both be included among the desires.

Intentions: Intentions represent the *deliberative state* of the agent - what the agent has chosen to do. Intentions are desires to which the agent has to some extent committed. In implemented systems, this means the agent has begun executing an action or a plan in the sense defined above. It is possible but irrational to have an inconsistent set of intentions.

Desires can include both *standing desires* such as "do not allow any malware into the computer" and *occasional desires* that apply at a particular time, such as "obtain printouts of all the pictures in the favorite photos directory." The same distinction applies for goals. Intentions are always occasional, on the other hand, although they may involve repeated execution of a particular task at several points in time, for example "backup this USB every morning."

2.2.3 The Relation between Desire and Gratification

The organization of a cognitive agent in terms of desires, goals and intentions is not the only possible one, but it is relatively common. The desires are frequently expressed in terms of positive or negative *gratification,* that is,

[6] These definitions have been obtained from the Wikipedia page for the BDI Architecture in October, 2011, with moderate modifications.

quantities that are associated with particular conditions in the agent or its environment. For example, in a cognitive agent that is set up so as to perform tasks that are requested by its owner (a person) and by other agents, one may define a gratification for each completed task whose size depends on who requested the task and the difficulty of the task. Instead of defining a desire for the agent to perform all requested tasks, one may define a desire for it to maximize its gratification. This may have the advantage of adding some flexibility in its behavior, and it may be the basis for the mechanism whereby the agent assigns priorities to competing tasks. Notice however that in this case the set of desires in the agent is specified by its designer, and the gratification is an auxiliary construct.

One may also consider the possibility of reversing the relationship and to consider the gratification (of one or more types) as the primary mechanism. If the BDI concepts are applied to people then it is natural to identify some biological desires, such as eating and general wellbeing, but also some cognitive ones, for example satisfying our curiosity, pleasing our friends, or having a good conscience. Desires of the former type are obviously the result of our constitution, and a plausible explanation for why we have those cognitive desires is (at least partly) that we have been trained to have them using emotional gratification earlier in life. It is of course possible to design software agents in a similar, gratification-based manner, so that cognitive desires are due to gratification, but there does not seem to be much point in doing so. For any reasonable purpose it is natural to let the desires of a cognitive agent be something that is defined by its designer.

2.2.4 Perception and Events

Input to a cognitive agent may come from physical sensors (in particular when the cognitive agent controls a mobile robot) and from interactions with people and with other (cognitive) agents. For the purpose of a cognitive architecture it is common to assume that the inputs to the cognitive automaton is in the form of *events,* i.e. symbolic expressions that describe something that has happened momentarily. The perception function will then be a function from external states to sets of events.

Events are used as triggers that cause changes in some parts of the agent's internal state, for example, updating a belief, starting the execution of a plan, or modifying a goal. In implementation terms, each user input and each incoming message from another agent may be represented as a separate event, whereas for sensors there is usually a need for lower-level software that receives the incoming data stream from the sensor and aggregates it to the form of discrete events. For the purpose of introspection and reflexive behavior there may be internal "sensors" that observe the agent's own computational processes and report their observations as internally generated events.

Notice that events are defined to be momentary and that they are normally not preserved in the agent's memory; their only role is to initiate changes in the internal state or to invoke cognitive activity. In ordinary language the word 'event' usually means something significant or remarkable that happens over a period of time and that may contain a number of sub-events, for example, a football match, or a parliamentary debate. Such things will be called *happenings* (in the sense of things that happen) and not events

in the present text, and the term *event* will only be used in the way stated above.

2.2.5 Functional Processes in a Cognitive Architecture

Langley et al. [7] identify the following functional processes or *capabilities* that are needed in a cognitive architecture.

- *Recognition and Categorization:* Detecting that a particular phenomenon is present in the agent's environment, according to given inputs, and relating it to other known phenomena.
- *Decision Making and Choice:* Deciding which actions to perform, as discussed previously in this chapter. Decision in the broader sense of categorization may be included in the previous item, but there is no sharp borderline between different kinds of decision.
- *Perception and Situation Assessment:* The capabilities of a perception function and of 'blink' type decisions that were discussed in Chapter 1. Deliberative situation assessment, i.e. reasoning about the pros and cons and the merit value of a current or anticipated situation may be included here or under the title of Reasoning.
- *Prediction and Monitoring:* Prediction means extending the agent's model of the current external state so that it also models anticipated external states in the future. This includes both predicting the effects of the agent's actions, and predicting future states of the environment in those cases where the environment changes state even when the agent does not perform any actions. Monitoring in this context means comparing the current external state with previously made predictions for the present point in time, and adjusting current predictions accordingly.
- *Problem Solving and Planning:* Planning is the capability of constructing a sequence of actions that is intended to achieve a selected goal, in particular, that a particular condition becomes true in the agent's environment. Problem solving is a broad term that will be addressed in a later volume of this series, but in particular it includes constructing a sequence of actions in an artificial environment, for example in a puzzle.
- *Reasoning and Belief Maintenance:* Reasoning is the capability of drawing conclusions from known or believed facts, including but not restricted to logical inference. Belief maintenance is relevant if conclusions are retained in the agent's memory; it is then the capability of retaining the reasons that such conclusions were based on, so that if those reasons are later retracted then the conclusions will also be retracted.
- *Execution and Action:* Performing selected actions. This includes the use of the effectuation function visavis the agent's environment, as well as the execution of composite actions through the successive execution of their subactions. Execution of actions is closely related

[7] Article cited at beginning of Section 2.2. The last item has been added by the present author.

to monitoring that each action has the intended effects, and it requires the capability of adapting the behavior if the execution of an action does not have the intended effects, for example, by making a new plan and executing it instead ("trying again").

- *Interaction and Communication:* This includes both interaction with humans and interaction with other software agents, cognitive or not. Interaction may serve a number of purposes, such as *inquiring* in order to obtain information that is known to the other agent, *delegation* in order to arrange that the other agent performs certain actions, and *reporting* in order to provide information to the other agent.
- *Remembering, Reflection, and Learning:* Besides the obvious general capability of retaining information in the agent's internal state, this emphasizes the capability of retaining a record of past actions and earlier states of the environment, doing reasoning based on this information, and modifying the rules that govern the agent's behavior based on that reasoning.
- *Association between Semantically Related Concepts:* In agents having a very large beliefbase, this refers to the capability of identifying beliefbase items that are loosely related to items that are presently being considered, without any requirement on a strictly logical relationship. This capability is typically used as a first step that produces a number of candidates for further consideration, and where reasoning is needed for selecting between those candidates.

It is seen at once that there are many dependencies between these capabilities, which means that they can not be implemented independently of each other. One important role of a cognitive architecture is therefore to provide a framework for the effective organization and synergy between the capabilities.

2.2.6 Memory Management

The *memory* of a cognitive agent is that part of its internal state that contains records of its past actions and of past external and internal states. Memory in humans is understood to be of several kinds, in particular, short-term memory versus long-term memory, and symbolic memory versus visual and other sensor-based memory. Most cognitive architectures also partition memory in similar ways, and in particular between short-term and long-term memory. This is done not merely in order to imitate human memory, but in order to make some of the functional processes more effective and efficient. Different types of memory can be organized in different ways which are adapted to their respective usage in the functional processes, and the process of *forgetting* can be organized differently depending on the length of the desired retention of the information in memory.

2.3 Software Individuals

The concept of *software individuals* has been proposed by the present author and is advanced as an important basic aspect of the design of cognitive agents. The present section will describe this concept.

2.3.1 Definitions for Software Individuals

Consider the situation where you have installed some software in your own PC or laptop computer, using one of the ubiquitous "installation wizards." Checking the contents of the machine's persistent memory after the installation, in many cases you will find that the installation has introduced an additional directory with subdirectories that contain the program(s) and the data that are required for the service in question. Let us use the term *a software* (in the singular) for such a directory with its hierarchical contents. The software will typically contain one or more executable files that can be used to start computational *sessions* with *the software* in question.

Sometimes the working data that you introduce as a user and use with the software will also be stored inside the same directory, which then means that the sessions will both *make use of* the (contents of the) software (in the sense of the directory with its contents) and *update* the (contents of the) software.

This is all commonplace and well known, but now consider the following additional notions for a software, and more specifically for a software automaton, whereby it obtains the character of an *individual.* Suppose you make a copy of the software in another location, either on the same hard disk or, for example, on a USB memory stick. In this case we shall say that what you obtain is *another* individual. By contrast, if you move the software to another location in the persistent memory, then it is still *the same* individual. Similarly, if you change the name of the main directory of the individual, then it will also remain *the same* individual.

The assumption of using the directory structure below one particular root directory for representing a software individual should not be taken too literally since it refers to one simple and obvious possibility, but it is not the only possibility. In a more general formulation, each software individual should be represented as a distinct and coherent *data aggregate* that is separate from the data aggregates of other such individuals and that can be manipulated and referred to as an entity. We shall continue to refer to it in terms of its simple implementation as a directory with subdirectories, however, both for concreteness and since in this case there is a well-known set of operations that one can apply to it, such as copying and renaming.

A software indvidual must be understood in terms of two interacting *manifestations.* The directory with its subdirectories and files (or more generally, the data aggregate) is the *persistent manifestation* of the individual; a computational session that is invoked from the persistent manifestation is a *dynamic manifestation* of the same individual. These manifestations are interdependent: the dynamic manifestation loads information from the persistent one, but it shall also be able to update files and structures in the persistent manifestation. It is in this way that the persistent manifestation can change its contents and evolve over time.

Because of these technical arrangements, software individuals have the essential properties of *identifiability* and *longevity.* Longevity results because the persistent manifestation of the individual is preserved on its memory device between sessions, and since it can be moved to alternative devices and copied for backup. [8] A software individual can therefore meaningfully

[8] The handling of backups in terms of the same vs copy distinction for software

be said to 'exist' for periods of many years. This is important as a basis for *machine learning* in the way that is natural for a system with cognitive artificial intelligence.

Identifiability means that in a community of several software individuals, it will be possible and meaningful to assign a unique *name* to each of them, and that individuals can have models of each others' knowledge and capabilities. This is important for communication and for task delegation between individuals.

A software individual that is at the same time a cognitive automaton will be called a *cognitive software individual.* As a summary of the previously given definitions, a cognitive software individual has the following properties:

- Its persistent manifestation is a data aggregate that contains programs and data as well as facilities for starting computational sessions in the computer where it is located.
- It is identfiable and has longevity.
- Computational sessions with this individual behave as a cognitive automaton, which means in particular that the internal state during the computational session contains a qualitative model of the individual's environment.
- Useful parts of the internal states of the computational sessions are preserved during sessions, and from one session to the next, by storing them in the persistent manifestation.

A cognitive software individual is said to have *integrity* if the changes to its internal state are under its own control, i.e. if they can only be performed using actions that the individual has itself selected to perform. If it is possible for a human user or another software individual to make arbitrary changes in the files and structures of the individual's persistent manifestation, then it does not have integrity. On the other hand, if all such changes arise due to the individual's own reactions to input events, then it does have integrity.

Absolute integrity is not practical and exceptions must be made for the purpose of software development and for backup and for protecting the survival of the individual. The manager of a software individual is however well advised to restrict himself or herself to *transparent* integrity violations, so that the software individual itself can operate under the assumption that it enjoys integrity.

The capability of *reasoning* was defined above. In particular, if the individual is capable of reasoning about the contents of its own internal state, and not only about its environment, then it is said to be *cognitively reflexive.*

Notice that cognitive reflexivity does not require that the entire internal state of the individual is expressed as logical propositions; it can also be obtained if there are internal sensors that provide access to its internal structure, and if these internal sensors express their observations as propositions that can be used for reasoning.

---

individuals is obviously an issue but it will not be addressed here.

2.3.2 Software Individuals as Platforms for Cognitive Artificial Intelligence

Cognitive software individuals with a reasoning capability are a likely design for building software systems with cognitive artificial intelligence (CAI). Consider, in particular, the requirement that a system with CAI shall be able to learn from experience. This is only meaningful if the system has some degree of longevity. For very simple "learning" systems that merely learn to improve one simple behavior it may be meaningful to learn during one particular computational session, but for the more complex form of learning that is of interest for CAI systems one definitely needs a system that will stay on for quite some time. For example, one will wish the system to be able to accumulate knowledge through successive learning experiences where later steps build on what was learnt in earlier steps.

Learning in this nontrivial sense means that the agent or individual in question changes over time. It follows that if you have a collection of individuals that are initially equal, and these individuals are exposed to different environments, then after some time they can be quite different. For example, if a number of such individuals are assigned as personal software assistants to different persons, then one should expect that these software individuals will have become somewhat different after some time.

Moreover, in an environment with several different software individuals, there will be situations where they need to exchange information and to cooperate. It is known that exchange of information on the cognitive level is facilitated if each participant in the interaction has a model of the knowledge of the other parties – not a copy of that knowledge, but a model of how it is organized and what its limits are. However, having such a model requires of course that the other parties are identifiable, so that they can be assigned names that can be used both in such a model, and for addressing the agent in order to communicate with it.

Besides for exchange of information, there is also frequently a need for *service offerings* and for *delegation* whereby one agent obtains the assistance of one or more other agents. This also requires each of the participating agents to maintain a model of the capabilities of the others, and it requires the participants to appear with their names in the stated agreements that are used to regulate the service commitments.

2.3.3 Philosophical Aspects of Intelligent Software Individuals

The use of software individuals is motivated by the direct needs of systems with cognitive artificial intelligence. However, the notion of software individual also lends itself to a number of intriguing philosophical considerations which we shall briefly discuss here.

One question may be whether the notion of software individual is always necessary. In the case of physical robots for use as general household robots, for example, could one not define the physical robot as the individual that has a name and a certain longevity, and leave it at that; the question of what is "the same" individual would seem to be completely clear anyway?

The answer is that this is true if one can assume that the software contents of that household robot are never going to be separated from its physical embodiment. However, suppose persons A and B have a personal robot each, and A's robot suffers some physical damage that makes it unsuitable for use by A, but still adequate for use by B. In such a situation one might swap the software contents of the two robots, so that A and B continue to have robots that are adapted to their respective needs, but the physical handicap does not cause a problem. In such a situation one would like to consider the software structure itself as an individual, regardless of what 'body' that it is in.

Doing so raises some interesting questions however. What if the software individual that was in the physically damaged robot retains a memory of what happened when it was damaged and what were the events that led up to it? Should these memories, and whatever had been learnt from them, be brought along when this software individual moves to an undamaged robot? How shall the other software individual understand the fact that one part of its new body is damaged? In fact, should these two individuals understand the concept of being moved from one "body" to another?

The question of *responsibility* may also need to be addressed. If the notion of software individual is used, then one can also ask whether each software individual should be equipped with a sense of responsibility for its own past actions and their effects. Defining such responsibility in a legal sense would obviously be to go too far, for the foreseeable future at least, but it might be that the sense of responsibility could be appropriate and useful for the internal considerations of each software individual, especially in robot-type environments.

Consider for example a household robot that may perform a variety of actions and that is organized so as to optimize its own behavior with respect to rewards (having done actions that are requested or whose results are valued) and some sense of cost. Assume furthermore that some of these actions may fail in a way that causes damage to the environment. It may then be reasonable to include a concept of responsibility in the definition of the robot's behavior, in such a way that it considers it imperative to obtain a solution for any such damage, regardless of the cost of doing so according to its own reward and cost structure. Such a notion of responsibility is necessarily tied to a particular individual. If there are several physical household robots in the same house then it becomes a significant issue whether they are organized as one single individual, or as separate individuals.

If the responsibility concept is introduced then there follows a question about what happens if an individual is cloned, i.e. one prepares several exact copies of it, and then registers each of them as a new individual. Should all of these have a responsibility for what was done before the cloning moment? Should it be possible for an individual to reduce its own responsibility by making many clones of itself (if it has the capability of doing that)? The general observation is that the notion of individuals works well as long as it works in the same way as for human individuals, but new and nontrivial issues arise if one allows operations that do not have any counterparts for bioagents, for example the cloning of adult individuals.

2.4 Software Architecture

As described in Section 2.2, a cognitive architecture contains a considerable number of facilities. Although some cognitive agents may manage with only a subset of them, it does seem that in order to achieve the full goal of cognitive artificial intelligence one will need, at least, all what is mentioned in the subsection on capabilities. Implementing all of this is a formidable task.

Moreover, these capabilities are interconnected in a large number of ways. Perception is required for the execution of actions, planning is required before action, remembering shall apply to the outcome of actions, for the purpose of learning from experience, and so forth. Therefore there is a practical need for a *software architecture* providing basic resources and a skeletal structure whereby the facilities in the cognitive architecture can be organized.

It is possible of course to define one's own software architecture in the first stages of implementing a cognitive agent system, but there are many advantages with identifying the design of a software architecture for cognitive agents as a topic in itself, and with implementing it as a software platform that can be reused by many and that can remain relatively stable as the design of the cognitive architecture evolves. The present chapter will discuss software-level techniques that are useful in the design of cognitive agents, and that are suitable for inclusion in their software architecture. Examples will be drawn from the author's Leonardo system [9] which is a software platform for cognitive software individuals.

The use of a software architecture is important regardless of whether a CAI system is organized as a software individual or not, but the use of software individuals does have implications for the software architecture. The present section will combine those aspects of software architecture that are specific to cognitive software individuals, with those that are not.

2.4.1 Programming Language Considerations

The programming language that is used for implementing a cognitive agent will be called its *host language.* Although in principle any programming language whatsoever *can* be used as a host language, some are better adapted than others for this purpose. If one programming language contains, within the language definition, a facility that is important for cognitive agents, and another programming language doesn't so that the facility has to be realized as a separate module, then one should expect the former language to offer better performance and better integration of the facility.

Support for the manipulation of *formulas* and of *symbols* is of paramount importance. Formulas are needed for the representation language which is basic to the cognitive architecture, as described above. Symbols can be characterized as normalized strings, i.e. strings that have been implemented in such a way that there can only be one occurrence of a given string in memory. The memory address of this single occurrence can be used for speeding up access to information that is related to the symbol in question.

[9] http://www.ida.liu.se/ext/leonardo/

In Lisp and Lisp-based languages, for example, there is a sharp type distinction between strings such as `"green"` and symbols such as `green,` and symbols but not strings can have properties assigned to them. In Python, on the other hand, normalized strings is just one kind of string. The difference means that, for example, if the string concatenation operation is applied to a pair of symbols in Lisp then a type error results, but the corresponding operation on normalized strings in Python will produce the concatenated string without error.

The availability of *recursion* and the facility of *garbage collection* are necessary for reasonable implementation of operations on formulas.

A further desideratum for the host programming language is that it should provide a *syntactic style* for embedded languages. The recursively parenthesized S-expressions of the Lisp language is the original example of a syntactic style: it defines a textual representation for data structures, and it is associated with a *parser* function that converts from text to data structure, and a *serializer* function that converts from data structure to text. However, the syntactic style is not a programming language, or any other kind of language: it is merely a way of writing data textually. The SGML-type formatting conventions that are used by HTML, XML and a host of other languages is another example of a syntactic style.

The importance of a syntactic style is that it can be used as a framework for a number of different languages, including both programming languages and special-purpose languages of various kinds. Every such language can be implemented as operations on the data-structure representation of the syntactic style, so there is no need to define and implement a new syntax for each language. Another advantage is that cross-connections between the languages is facilitated.

2.4.2 Attached Functions and Procedures

One important technique for software that processes formal expressions of various kinds, including for A.I. software on the knowledge processing level, is the use of *attached functions and procedures.* The basic idea there is to let symbols have functions and procedures as property-values.

Ordinary object-oriented programming (as originated in the Simula language) provides the capability of defining *classes* and to associate procedures with each class; these procedures are available to members of the class. However, in this case the classes are defined in the program and it is not possible to add further classes during the execution of the program, and there is a sharp distinction between classes and objects, which are class members. When attached procedures are used, on the other hand, classes in the object-oriented sense can be represented as symbols, the distinction between objects and classes goes away, and any symbol can have attached procedures. Procedural properties can be added and changed while the system is running, for example as the result of self-modification by the system.

Implementations of cognitive agents typically contain a number of embedded languages, including, but not restricted to the selected knowledge representation language. The implementation of an interpreter for an embedded language requires the use of a datastructure for representing expressions in

the language, and the use of procedures defining each of the primitive operations in the language. Again, facilities for the use of formulas and symbols are a basic requirement.

2.4.3 Handling of Actions

Cognitive agents are characterized by their use of *actions,* as was discussed in Section 2.1, and a software architecture for cognitive agents should therefore include actions as an essential construct. It should provide the means whereby it is possible to construct a plan consisting of actions, to evaluate plans with respect to feasibility and cost, to execute plans by performing one action in the plan after another, or by performing them concurrently, and furthermore to administrate records of previously executed actions and plans and to analyze those records. Most of these functions belong to the cognitive aspects of the system, but the software architecture should provide the necessary infrastructure, for example, facilities for defining action verbs, facilities for logging the execution of actions, and facilities for command-line input of actions, to mention only a few.

The actions of a cognitive agent are first of all actions in its environment, i.e., its *external actions.* However, there is an important relation between them and *internal actions* as defined by procedural expressions in the host programming language. Please recall that virtually all programming languages combine the use of *functional expressions* that are evaluated without side-effects, with the use of *procedural expressions* that do have side-effects, either on variable values and datastructures within the computational session, or on files and directory structures in the persistent computer memory.

Please recall also from an earlier section that actions may be elementary or composite according to the type of their verb. Verbs for composite actions are defined by scripts consisting of other actions. Elementary actions are usually defined by procedural expressions which provide the mapping from the agent's external behavior to its internal computation.

All this suggests the possibility of *applying the agent's facilities for external actions to be used for internal actions as well,* i.e. to make it possible for the agent to plan its own internal actions, analyze its own past action sequences, and so forth. If this can be done then it will be the basis for the agent's cognitively reflexive capability (see Subsection 2.3.1).

Notice as well that the borderline between external and internal actions is not an obvious one. In a conventional programming language it is natural to include both operations on session datastructures, operations on the file structure, and operations on printers and other external devices as being defined by the same kinds of procedural expressions. However, in the case of a cognitive agent that can perform actions in the physical world, for example in a robot that moves around, one could argue that it is more reasonable to consider the printout of a photograph as an external action, not very different from a move with the robot's hand.

The *decomposition of action definitions* should be an additional consideration. In the Leonardo system, for example, one and the same notation is used for both internal and external actions by the cognitive agent. Each action consists of an action verb and some arguments or parameters for it.

[10] However, unlike in most other systems, each action verb can be defined by *three* separate procedures, namely, a checking procedure, a performing procedure, and a reporting procedure. The checking procedure checks that the given arguments and parameters are correct so that the performing procedure is executable; the performing procedure performs the verb and reports the outcome in the form of an *outcome record,* and the reporting procedure presents the outcome to the user if appropriate, for example, if the action was given to the agent as an input in the agent's command-line dialogue with its user.

This decomposition has several advantages from a software-technology point of view, and it also has implications on the level of the cognitive architecture. All three verb-defining procedures are needed when an action is input from the command line, but if an action occurs in a script then the reporting procedure is usually not used (except sometimes for debugging purposes). Moreover, the checking procedure can sometimes be used statically for checking the correctness of a script or plan, and it can then be bypassed when the script is executed repeatedly. If an agent refers an action to another agent for execution, then the performing procedure should be done in the other agent but the reporting procedure should be done in the first agent.

The checking procedure can often be wholly or partly replaced by the evaluation of *preconditions* that are expressed as logic formulas. It is usually more convenient to write a precondition than to write the corresponding checking procedure in the host language, but more importantly, if a given set of arguments and parameters do not pass the test, then the reasons for the failure can be explained to the user through a generic explanation procedure that makes direct use of the logical precondition. Directly programmed checking procedures have to include a programmed explanation facility, or one has to add a fourth procedure for the verb, namely, a precondition failure explanation procedure.

Another major advantage of expressing preconditions in logic is that they can then be used for constructing, checking and modifying plans. This is an important aspect of the cognitive architecture which will be addressed in Chapter 4.

2.4.4 Representation and Organization of Knowledge

The topics of representation and organization of knowledge were addressed in the section on the cognitive architecture, but they are also relevant in the context of the software architecture. In particular, once one has defined a representation language for use in the cognitive architecture, it is appropriate to also ask to what extent the same language can be of use for more mundane, software-technology purposes in the software architecture as well. Such "downward filtering" of software techniques from the cognitive-processing level to more elementary software levels has been used extensively in the Leonardo system (which was mentioned above).

From the point of view of the software architecture, the question of the organization of knowledge is closely connected to the question of persis-

[10] The technical distinction between arguments and parameters will be defined in the next chapter.

tent storage. One example is given by the Leonardo system, where the persistent storage for a software individual is organized as a collection of text files, called *entityfiles,* whose internal format is based on Leonardo's Knowledge Representation Expressions. In simple cases, all the information for a particular software individual is expressed as the contents of a set of entityfiles that are located in a common directory structure with a single root. Another possible solution is to consider the dynamic manifestation of a software individual (i.e., an executing computational session) as the only important one, and to obtain persistence by saving a memory dump of the session on persistent memory in such a way that it can later be resumed.

Another important issue is hos to organize very large knowledge bases, how to make them available to each of a large number of cognitive agents, and how to organize correctness control and continuous update of knowledgebase contents. This topic will be addressed in a separate chapter together with the topic of large-scale ontologies.

2.4.5 Communication between Agents

Communicating computational processes are used in most branches of Computer Science and also in Artificial Intelligence. In the context of cognitive artificial intelligence one should distinguish two main cases. One case is where one software individual is organized as several cooperating computational processes (often called "agents" in a loose sense) that are complementary in that they provide different parts of the capabilities of the entire individual. Some of these processes may for example provide sensory and motoric capabilities for a mobile robot. In this case there is no expectation that each of the participating processes, or "agents," should have autonomy or be cognitive agents; some of them may do conventional computation. The other case is where one has a community of software individuals that may be fairly similar but which have different knowledge contents, and where their tasks may require them to cooperate for the purpose of exchanging information or for delegation of tasks.

In the second case, and sometimes also in the first case, there is a need for a *message-passing infrastructure* that is adapted to the representation language that is used within the software individual(s), including a language for use in these messages. The original design in this respect is the KQML language – the "Knowledge Query and Manipulation Language," which was developed around 1990 and which was adopted in particular in the DARPA [11] A.I. community as a standard for the representation of messages between agents. More about this topic is found in Chapter 6 and in the Annex of that chapter.

2.4.6 Identity Support for Software Individuals

The basis of the Software Individuals concept is that a technical distinction is made between moving and copying a data aggregate that constitutes an individual. Existing operating systems do not lend themselves to a waterproof implementation of this notion. It is possible to implement an approximation of it, but with this implementation it is still possible to "cheat"

[11] Defense Advanced Research Projects Agency, in the United States.

and make the system believe that you have "the same" individual although in fact you have a different one. However this is a technical question that may be solved with new technology which will probably require hardware support.

2.4.7 Conclusions

In this chapter we have described the architecture of cognitive agents on two levels: a *cognitive archicture* that is expressed in cognition-related terms, and a *software architecture* that provides a platform and framework for the cognitive architecture. The software architecture addresses traditional as well as relatively novel issues in software technology, but the various aspects of its design are of course motivated by the needs of the cognitive level.

Chapter 3

Basic Notation

The present textbook uses a notation that combines the notations of first-order logic and elementary set theory, together with a few further additions. Its typographical appearance and its choice of symbols have been chosen in such a way that the notation allows a dual use, both for publication in books and articles, and as input/output notation in computer software. The present chapter introduces this notation down to the level of detail that is needed for continued use in the textbook.

3.1 Informal Introduction to the Notation

The present section is adapted for the reader that is already familiar with the formalisms of logic and of set theory, and who merely needs to know about how the details of the notation have been chosen in the present case. The more systematic specification of the notation follows in later sections of this chapter, and in its annex.

We shall be working with formulas whose elements are of the following kinds:

- Symbols, eg. `newspaper`
- Strings, eg. `"The New York Times"`
- Numbers, eg. `3.14159`
- Variables, eg. `.name`
- Tags, eg. `:name`

Composite expressions are formed using six types of brackets, namely round parentheses, square brackets, angle brackets, curly brackets, special maplet brackets, and brackets for strings with special syntax. All expressions are formed using the Latin-1 character set and are presented using fixed-width font ie. “typewriter” font.

Angle brackets are represented using the less-than and greater-than characters and are used for forming sequences with explicitly specified members. Curly brackets are used for forming expressions for sets with explicitly specified members. The following are some examples:

```
<red white blue>
{red white blue}
```

Commas may optionally be used for separating the elements of sequences and sets so they count as whitespace characters in that position.

Square brackets are used for two major purposes, the first one being for specifying literals consisting of a predicate symbol and its arguments, for example

```
[is-older peter gustaf]
```

Round parentheses are used for forming terms, recursively, for example

```
(capital-of sweden)
(population-of (capital-of sweden))
```

They are also used with logical connectives for forming composite propositions, for example

```
(and [is-older peter gustaf][is-male peter])
```

There are two kinds of term-forming functions, namely those that construct composite entities (cf. Herbrand functions) and those that are associated with a computational procedure or some other method that allows one to map a function symbol with its evaluated arguments to a new entity, for example a symbol or a number.

A mapping is considered as a set of maplets and can be written in two alternative ways, for example:

```
{johan = 23, berit = 25}
{[: johan 23][: berit 25]}
```

The advantage of the latter notation is that it allows one to identify and use individual maplets.

The second use of square brackets is for forming records which are in turn used for a variety of purposes related to actions and their effects. The following is an example of an expression for an action:

```
[move box-12 :by robot-4 :dest room-3]
```

Thus a record is an expression consisting of an operator (`move`), arguments and parameters. Each parameter is preceded by a tag.

The use of brackets for strings with special syntax is described later on in this chapter.

The definition of the notation proceeds in two steps, where we first introduce *Knowledge Representation Expressions,* or KR-expressions, and then the *Common Expression Language,* CEL. The idea is that KR-expressions is the syntax for constructing expressions using the various kinds of brackets that were described above, whereas CEL adds the further conventions that literals are to be expressed using square brackets, terms are to be expressed using round parentheses, and so forth.

The reason for making the distinction between KR-expressions and CEL is that KR-expressions provide a *syntactic style* that is being used for several other languages besides for CEL. This is analogous to how S-expressions, as introduced by the Lisp language, is used not only for writing Lisp code but

also for languages such as KIF, KQML, and several generations of PDDL. In terms of software technology, the use of a syntactic style makes it possible to have one single parser that converts KR-expressions to an internal representation in the computer, and then the various KRE-based languages can be implemented so as to use that internal representation. – The use of SGML-style bracketing in HTML, XML, and a large number of XML dialects is yet another example of a syntactic style.

After this informal introduction we proceed to a more systematic definition of the syntax. However, the reader who feels that he or she has understood the general idea can easily proceed to the next chapter at this point, and the rest of the present chapter may be used for reference when needed.

3.2 Knowledge Representation Expressions

This section defines the notation for *Knowledge Representation Expressions,* or KR expressions for short.

3.2.1 Atomic KR-Expressions

KR expressions are constructed from three kinds of elements:

- *numbers* , which are written in the usual way, for example as `128` or as `3.14`
- *strings* , which are written enclosed by double quotes, for example as `"This is a string"`.
- *tokens* , which are written as a sequence of characters that does not contain a space or a double quote, and that can not be interpreted as a number.

Special rules apply for the use of interpunction characters (i.e. those that are neither letters nor digits) in tokens, as well as for the use of "nonprinting characters" such as the tab character. These rules are specified in the Chapter Annex.

A *tag* is a token where the first character is a colon; a *variable* is a token where the first character is a full stop. All other tokens are called *symbols.*

Numbers, strings, symbols and variables are *atomic KR-expressions.* Tags are not.

3.2.2 Composite KR Expressions

A *KR-expression* can be atomic or composite. Composite KR-expressions are constructed from atomic KR-expressions using parenthesization recursively. The following are some examples of KR-expressions:

```
(+ 4 .a)
(fib 9)
<red green blue>
{1 4 9 16 25}
```

```
[move :by robot-4 :obj box-12 :dest room-3]
```

In order to define the syntax for KR-expressions we must therefore first define the syntax for an argument list.

An *argument list of arity* `n` , where `n` is a non-negative integer, is a sequence of exactly `n` KR-expressions, separated if necessary (as defined in the Annex) by at least one whitespace character (space, tab or new line) between successive elements.

An *extended argument list of arity* `n` is an argument list of arity `n` , immediately and optionally followed by one or more pairs consisting of a tag and a KR expression called the *parameter for* the tag that precedes it. The following is an example of an extended argument list:

```
red big :size 44 :caption "Dynamo"
```

A *form* is composed of a *formant* and a (non-extended) argument list, where the formant is an entity or a form. The concrete manifestation of a form encloses the formant and the argument list by round parentheses.

A *sequence* and a *set* is defined as usual in set theory, and are represented by enumerating all the elements. The concrete manifestations of sequences and sets consist of an argument list of arbitrary length, surrounded by angle brackets, $\langle\rangle$ or curly brackets, {}, respectively. [1]

A *record* is similar to a form, but it is formed using an extended argument list so tag/parameter pairs are admitted, and in its textual manifestation it is enclosed in square brackets rather than parentheses. The operator forming the record is called the *record composer.*

A *mapping* is a set of *maplets* each of which is a twotuple consisting of an entity and a corresponding value, and is written like in the following example:

```
⦅age 46⦆
```

or equivalently as

```
[: age 46]
```

The first notation is for use in publications; the second one is for software use where the special mapping bracket is not available.

The mapping can not contain two maplets ⦅e v⦆ and ⦅e w⦆ having the same first element and different second elements.

An entity e is said to be *defined in* a mapping M iff there is a maplet ⦅e v⦆ in M for some v.

Two mappings are *separate* iff no entity is defined in both of them.

A *KR expression* is either of a string, a number, a symbol, a variable, a form, a sequence, a set, a record, or a maplet.

[1] The software variant of the KRE notation uses less-than and greater-than characters as angle brackets, in order to remain with the characters on the standard computer keyboards. The publication variant of the notation, which is used here, may use the $\langle$ and $\rangle$ characters instead.

3.2.3 Convenience Extensions of KRE

Besides the basic notation that was described above, there are some extensions to the notation that may help to make it even more readable. Some of these will be introduced in later chapters, in contexts where we can see the reasons for them, but some can be introduced here.

The elements of a set or sequence may be separated by commas. This is in accordance with conventional set theory notation, but notice that here it is not necessary, just an option. The use of commas is often natural when the members of the set or sequence are elementary objects, but not when they are composite expressions.

An expression for a mapping consisting of explicitly written maplets, for example

```
{[: johan 23][: berit 25]]
```

can alternatively be written as follows

```
{johan = 23, berit = 25]
```

The latter notation is however not sufficient in situations where one wishes to refer to individual maplets. The bracketed notation is more general.

Sometimes it is convenient to be able to write the parameters at the beginning of a record expression, rather than towards the end. The notation for this is shown by the following example where

```
[move box-12 :by robot-4 :dest room-3]
```

can equivalently be written as

```
[move :by robot-4 :dest room-3 ^ box-12]
```

3.2.4 Embedding of Other Languages

Some applications require the embedding of other languages within KR Expressions. This can sometimes be done by representing expressions in the other language as strings, but doing so is impractical e.g. when the embedded expressions contain double quotes. There is a general notation whereby one writes

```
[§altlan (...  expression in the altlan notation) §]
```

This notation allows the embedding of a variety of languages each of which is characterized by its own code to be used as `altlan` , so for example

```
[§tex one may embed text in {\em LaTeX} markup §]
```

The only restriction on the expression in the embedded language is that it may not contain the textsection character § and a right square bracket in immediate succession. The explicit tagging of the choice of language is helpful for organizing the processing of such data.

The empty alternative language code is used for plain strings, so that the expresson

```
"This is a string"
```

is exactly equivalent with

```
[§ This is a string§]
```

This provides a way of writing a string that contains a double quote inside it, as in

```
[§ This:  " is the double-quote character§]
```

3.3 The Common Expression Language

The Common Expression Language (CEL) is one of several languages that use KR-expressions as their syntactic style. Therefore, every expression in CEL is a KR-expression, but the opposite is not the case. The interpretation of a KR-expression as a CEL-expression is always done relative to a *signature* that may specify, for example, restrictions on the number and the type of arguments for a particular formant. We shall refer to it as the *current signature* for the expression at hand. The exact representation for expressing signatures is described in the Annex.

The purpose of the Common Expression Language is to provide i.a. the following kinds of expressivity:

- Constructs for expressing *commands* to a software individual during a session with it.
- Constructs for *retrieving* information from the current knowledgebase in the session where it is used.
- Constructs for specifying the *composing and decomposing* KR expressions.
- Constructs for expressing statements about the world in a knowledgebase.

3.3.1 Tokens

The KRE syntax identifies two particular kinds of tokens as not being symbols, namely, tags and variables. A CEL signature may in addition specify particular roles for certain symbols according to their morphology, in particular, their use of interpunction characters, and their use of upper-case letters.

A signature may recognize a symbol as a *typed symbol* if there is an occurrence of the full stop inside the symbol (not at the beginning or at the end). This initial segment of the symbol is then interpreted as a type name. The signature will specify what are the admitted type names for use in typed symbols. The use of typed symbols is convenient in order to avoid name conflicts, but there is no requirement to use them.

A signature may also specify that symbols beginning with a particular interpunction character shall play a particular role. For example, if the first character is the dash, as in `-partof` then the symbol is interpreted as the negation of the predicate symbol without the dash, such as `partof` in the example.

Symbols of the form `nsp.Lastname.Firstname` are by convention considered as representations of the name of a person, where `nsp` serves to identify the namespace within which the name is defined. There may also be a postfix for disambiguation, for example `nsp.Lastname.Firstname.3`

When one develops a knowledge representation it is recommended to keep these options in mind even if one does not use them oneself. For example, it is recommended not to use interpunction characters as initial characters in symbols unless this is declared in the signature at hand, since otherwise there may later on arise conflicts with naming conventions that are introduced by other users.

Besides the signature-specified restrictions on tokens, there is also a recommendation that when one defines a facility that may eventually be used by others, one should not use a capital initial letter in symbols that one introduces, so that the end user can use capitalized symbols as a means of avoiding name conflicts with the implementor.

3.3.2 Actions, Terms and Evaluation

The CKL uses sets, sequences, and mappings in just the way that they are defined for Knowledge Representation Expressions. In addition it uses a number of CKL-level constructs that are represented using forms and records on the KRE level. The most important constructs are *actions, terms,* and *propositions.* One way of using these is for commands to a software individual, in which case the entire command is typically an action containing a verb and its parameters, and where the arguments or parameters for the verb are expressed as terms. Commands can also be composite actions, in which case propositions may occur for the formation of conditional expressions. Elementary propositions consist of a predicate and its arguments, where the arguments may be e.g. terms or actions.

Another way of using these is for making statements that describe phenomena in the world. Such statements are expressed formally as propositions.

Terms are represented as KRE forms, which means that they are formed using a *term composer* followed by *arguments* and are enclosed in ordinary, round parentheses. *Notice that the term composer is also located inside the parentheses,* which is different from standard mathematical notation. Actions are represented using KRE records, which means that they are formed using a *verb* optionally followed by arguments and/or by *parameters* that are marked by *tags.* Propositions will be addressed in the next subsection.

The operation of *evaluation* is defined for all CKL expressions. The evaluation operation takes a CKL expression and a mapping from variables to values as input, and produces a CKL expression as value. This operation is defined recursively as follows.

- The value of a variable is the value assigned to it by the given mapping.
- Symbols and strings evaluate to themselves.
- The value of a set expression is the set formed by the values of the elements in the given expression.

- The value of a sequence expression is the sequence formed by the values of the elements in the given expression.
- The value of an action expression is the similar action expression formed using the same verb and by evaluating the arguments and parameters.
- The value of a maplet is the maplet formed by evaluating the two elements of the given maplet.
- The value of a term is obtained using rules that are part of the signature at hand.

The evaluation of terms is defined using information that is specific for each formant. Signatures shall distinguish between two kinds of formants for terms, namely *entity composers* and *term composers.* The value of a term whose leading element is a term composer is obtained by evaluation of a *definition,* i.e. an expression that is associated with this term composer in the signature.

Entity composers have a particularly simple structure. The value of a term that is formed using an entity composer is a *composite entity* which has the following uniqueness property: the values of two such forms can only be equal if they have the same entity composer, and if their arguments are pairwise equal. Therefore they correspond to Herbrand expressions in logic, and to terms in the Prolog language.

The following composers will be used later on in the present book. The larger list of composers that are defined in the Leonardo system is available in the Chapter Annex.

```
(get .c .a)
```

Obtains the value of the ***a*** attribute for the 'carrier' entity ***c.*** Ω

```
(filemembers .ef .ty)
```

Obtains the set of those members of the entityfile ***ef*** whose type is ***ty.*** Type is only defined directly as the value of the `type` attribute and does not allow for subsumption from a supertype. Ω

```
(str.concat .s1 .s2 ...)
```

All the arguments shall evaluate to strings; this function obtains the concatenation of the values of the respective arguments. Ω

Use of Infix Notation for Operators

As an additional convenience measure, some composers are specified to be *infix composers* and these can *optionally* be placed between the first and the second argument, provided that there are exactly two arguments. It is always possible to let the composer be the first element and thereby to precede the arguments. For example, the term `(a + b)` is the same as `(+ a b)` given that `+` has been declared as infix. The infix specification is part of the signature.

Moreover, if an infix composer admits more than two arguments as well, as is the case for example for `+` then it is possible to use the natural notation

where the composer is placed between all arguments, so that e.g. `(a + b + c)` is the same as `(+ a b c)`. Notice that it is obligatory to repeat the infix operator, so `(a + b c)` is not correct.

Notice also that it is not allowed to mix several different operators, prefix or not, within the same parenthesis level. Thus it is not permitted to write `(a + b * c)` and one must write this as `(a + (b * c))`.

The following are some of the infix operators: `+ - * union`. All except `-` allow n arguments in the way just described. The `-` composer can be used with one or two arguments, but not more.

3.3.3 Propositions

Propositions are formed recursively from *literals* . They have truth-values such as `[true]` or `[false]` as their values.

Literals

Literals are expressed as records that may only have arguments, not parameters. The signature at hand specifies whether a record former is a verb, so that it is used for forming actions, or whether it is a predicate whereby it forms literals.

A *positive literal* consists of a *predicate* and its *arguments* and is enclosed in square brackets. Normally the predicate precedes the arguments, but like for terms there are some predicates of two arguments that have been declared to be infix operators. For example, one can write `[a = b]` equivalently with `[= a b]` The following predicates are defined in basic CEL.

A *negative literal* is the negation of a positive literal. This is represented by preceding the predicate with a dash character (-). For example, the negative literal `[-sub a b]` is the negation of the positive literal `[sub a b]` A negated literal can be written in infix if this is allowed for the corresponding positive literal.

The following predicates will be used later on in the present book. The larger list of predicates that are defined in the Leonardo system is available in the Chapter Annex.

`[true ]`

This predicate of no arguments always has the value true. Ω

`[false ]`

This predicate of no arguments always has the value false. Ω

`[equal` ***a b***`]`

The literal is true iff the two arguments are equal. Equality between sets is defined so that two set expressions containing the same members but in different order are still considered as equal. Ω

`[member` ***a b***`]`

Membership relation in a set. Ω

`[subseteq` ***a b***`]`

Subset relation between sets. Ω

`[singleton` ***a***`]`

The literal is true iff ***a*** is a set or sequence with exactly one member. Ω

`[hastype` ***a ty***`]`

This literal is true iff the type of the entity ***a*** i.e. the value of its `type` attribute is either equal to ***ty*** or is subsumed by ***ty*** according to the entity subsumption predicate `sub.` Ω

`[sub` ***a b***`]`

The two arguments shall be entities representing types or other things for which a subsumption relationship is defined. The literal is true iff there is a subsumption chain from ***a*** to ***b,*** including the case where the two arguments are equal. Ω

Composite Propositions

A *boolean composite proposition* is formed from positive and negative literals by combining them using the operators `not, and, or` or `imp` and surrounding the expression with round parentheses. This means that they are represented as forms on the KRE level. The operator `not` has one argument, `imp` has two arguments, and `and` and `or` can have two or more arguments. All except `not` can be written in infix mode.

The negation obtained using the operator `not` is the same as the one obtained by using a negative literal. For example, `[-sub a b]` is entirely equivalent to `(not [sub a b] )`

A *quantified composite proposition* is formed recursively from the previous types of propositions also using quantified expressions which may be formed in either of two ways, namely, untyped or typed. Untyped quantified propositions are formed as in the following examples:

```
[all {.x .y} ([.x sub .y] imp [.y -sub .x])]
[exists {.x .y} ([.x sub .y] and [.x -equal .y])]
```

In this case there are exactly two "arguments" for the quantifier, where the first argument is a variable or a set of variables. If only one variable is quantified then the enclosing curly brackets may be omitted.

Typed quantified propositions are formed as in the following examples:

```
[all :x a :y b ^ ([.x sub .y] imp [.y sub .x])]
[exists :x a :y b ^ ([.x sub .y] and [.x -equal .y])]
```

In this case there is exactly one "argument," namely, the quantified expression, and the quantified variables and their types are expressed using parameters for the record at hand. The tag in each parameter serves as a quantified variable; the corresponding parameter value shall be an entity representing a type that the variable in question will vary over. Notice that the variable symbols are represented as tags and therefore preceded by a *colon* when the variable is specified with its type in the quantifier, and by a

fullstop (i.e. point) when the variable is used in the body of the quantified expression.

The first example shall thus be read "for all x whose type is `a` and for all y whose type is `b` , `[`x `sub` y`]` implies `[`y `-sub` x`]` ".

Typed quantified propositions can be reduced to untyped ones using a simple transformation. The first of the two examples above reduces to:

```
[all {.x .y}(imp (and [hastype .x a][hastype .y b]
                      [.x sub .y] )
                 [.y -sub .x] )]
```

3.4 Notations for Special Facilities

Most systems for knowledge representation use a notation that is based on first-order logic and elementary set theory, and most of them also extend this basis in a variety of ways. Such extensions will be introduced in several chapters in the present textbook series. Here we shall only mention some extensions that are of general interest since they re-occur in a variety of ways.

3.4.1 Rules

A *one-way rule* is formed like in the following example:

```
[! [grandfather .x .z] if [father .x .y][father .y .z]]
```

In general, a one-way rule is formed as a record where the symbol `!` is the record formant, the second "argument" is the symbol `if,` and all other arguments are positive or negative literals. One-way rules are intended to be used in axioms that define predicates or functions: the rule in the example is intended to mean the same as the formula

```
(imp [exists .y person (and [father .x .y][father .y .z])]
     [grandfather .x .z] )
```

with implicit universal quantification.

The exact general definition is given in the Annex. Parameters in the one-way rule can be used for restricting the type of each variable that occurs in it.

A *two-way rule* is similar but uses the symbol `iff` instead of `if` and its meaning uses two-way implication (i.e. equivalence) instead of `imp.`

Notice that the existential operator in the definition is redundant in the case of a one-way rule, but it is essential for a two-way rule.

3.5 Relaxation of Syntax for Publication Use

The syntax for KR-expressions was defined with two major goals in mind: it should be a systematic and easily readable notation for structured information, and it should be directly usable for input and output to computers

using the standard keyboard and character set. In particular it should be possible and convenient to use this as the standard notation in textbooks and articles.

When the notation is used for publication purposes, it may actually be convenient to allow the commonly used symbols for a number of functions and predicates, although they are not available in the computer keyboard. For example, the `subseteq` predicate can be replaced by the commonly used symbol $\subseteq$, and the term composer `union` can be replaced by the symbol $\cup$ as usual.

In publication usage it is not necessary to always insist on full parenthesization of terms. A certain latitude in the use of the notation helps readability and it is insignificant as an obstacle to the easy transfer of definitions and information collections between publications and their use in a computer system. In any case it is natural to require that the computational variety of KRE and CEL shall be defined with full precision, and only to allow commonsense-based exceptions in the publication variety.

Chapter 4

Domain Modelling

One distinguishing property of cognitive agents is that they contain qualitative models of their respective environments. Such a model is called a *domain model* for the environment where the agent is situated. The present chapter addresses knowledge representation methods for use in domain models.

Since another characteristic property of a cognitive agent is that its behavior is organized in terms of actions that are subject to active choice by the agent, a domain model should not merely describe the environment in itself; it must also contain descriptions of the actions that the agent can perform, in particular, under what circumstances is it possible for the agent to perform the action, and what may be its effects.

Furthermore, in those applications that involve the use of several cognitive agents, the domain model should contain representations of the actions that those other agents can perform. Finally, with respect to actions it is natural to consider human beings and other active-choice actors on a par with software agents. For simplicity we shall use the term cognitive agent for them as well and distinguish, when necessary, between biocognitive agents (people, in particular) and technocognitive ones (those based on information technology).

If a cognitive agent controls a mobile robot, an intelligent process-control system, or some other complex mechanical system then it will usually also need to contain a quantitative model of that system and its environment. The integration between the qualitative and the quantitative model is an important and difficult issue which will be addressed in a later volume of the present series of textbooks, but not in the present volume.

Domain modelling is addressed not only in Artificial Intelligence but also in several other areas, such as the Database technology area within Computer Science, and in Computational Linguistics. Unfortunately there is no common basis for domain modelling in these disciplines; each one uses its own approaches. Making things worse, different subareas of Artificial Intelligence also do not use a common approach.

In spite of this diversity of formal or informal approaches, there are a number of important principles that recur in several of them. The present chapter identifies a generic common base within which major current approaches can be explained, and which is used as a basis for the following chapters

as well as in the following volumes in the textbook series. There is also an Annex to this chapter containing an overview of other current approaches to domain modelling, as well as a more detailed exposition of some of the chapter's topics.

4.1 Entities, Models, Time and Contexts

4.1.1 Entities and Models

One of the first steps in the design of a domain model for an application is to identify particular things in the application with which one can associate various kinds of information. These things may be for example persons, houses, roads, plants or newspapers, to take some examples. Things in the application that are identified in this way are referred to as *entities,* and entities are the elements of domain models.

Suppose in one very simple application that you are concerned with some persons, some houses where they live, a few cars that are owned by these persons, and a number of trees in their gardens. These will then be the entities in the *model* that you construct of the application. There are a few obvious relationships between the entities, and these relationships can also be included in the model.

In reality there are of course many other things in the application that might have been identified for inclusion in the model, and there may be other relations between entities that were not selected for inclusion. Often it is not obvious what things in the application merit being identified as entities. For example, if you introduce entities not only for entire houses but also for parts of houses, then where do you draw the line for what parts to identify; will you consider the set of all chairs in the dining-room furniture as one entity for example?

Therefore, a domain model is merely a *selection* of entities and relationships in the real-world application. The designer of the domain model has to make an active choice about what to include and what not to include.

In order to work with a domain model one has to introduce a *representation* for it. The most popular representations are *formal representations* that consist of a set of formulas, and *graphical representations* that consist of one or more diagrams that are typically organized using boxes and arrows (or other similar connections) between the boxes. Formal representations are favored in Artificial Intelligence and Knowledge Representation because they are better suited as a basis for automatic reasoning and inference, but graphical representations of the same domain model may sometimes be used for illustration purposes.

The present textbook uses a formal representation based on an extended first-order logic. The extensions as well as the syntactic and graphical conventions follow the definitions in Chapter 3.

Formal representations of domain models are always based on the use of symbols as *names* for some or all entities in the domain model. In the simplest case each entity in the domain model corresponds to exactly one symbol in the representation, and this symbol may be either an atomic symbol or a composite symbol which has been formed using a symbol composer.

More generally it may be that some entities in the domain model have several names, each of which is an atomic or composite symbol, and that some other entities do not have any name at all.

The representation of an application must always be the representation of a model of that application. The choice of domain model is the choice of which aspects of the application are going to be represented; the choice of representation is the choice of which symbols are going to be used as names for entities in the domain model, together with the choice of how to express the relationships between those entities.

The simplest kind of entity is a particular physical object or person. These will be referred to as *tangible entities.* By contrast there are also *intangible entities* such as contracts, or languages, and *social entities* such as families and football teams. Additional types of entities will be introduced further on.

4.1.2 The Naming Assumptions

Here is an example of a representation of a very simple domain model.

```
[has-type anders person]
[has-attribute anders sex male]
[has-type house-1 house]
[has-owner house-1 anders]
[has-type bertil person]
[has-attribute bertil sex male]
[has-type gunilla person]
[has-attribute gunilla sex female]
[has-type house-2 house]
[has-owner house-2 bertil]
[has-owner house-2 gunilla]
[are-married bertil gunilla]
```

The representation is therefore a set of propositions in the sense of the previous chapter, with `has-type, has-attribute` and so forth as predicates with obvious meanings. (The predicate `has-owner` is taken to mean that the person in the second argument is one of the owners of the thing in the first argument, but not necessarily the only one).

When one reads this representation it is natural to assume that the symbols `anders` and `bertil` refer to two different entities, i.e. to two different persons in the domain model, and similarly for `house-1` and `house-2.` In general this is called the *Unique Names Assumption* (UNA), i.e. that an entity can not have more than one name. This assumption is often used, but it is not a necessary one from the point of view of formal logic.

Consider now the following proposition:

```
[all .x (imp [has-type .x house]
             [exist .y (and [has-attribute .y male]
                            [has-owner .x .y] )])]
```

This formula says that for every house in the domain model, there is some male person who is an owner (single owner or co-owner) of that house. Is

this a true statement about the domain model, according to what is said about it in the above representation?

The answer to that question depends on whether one has adopted the *Closed World Assumption* (CWA) for the representation of this domain model. The CWA means that *every entity in the domain model is represented by some symbol that occurs in the representation.* In the present case it means that there are only two houses in the domain model, and for these two houses we have of course positive information that each of them has one male owner. On the other hand, if the Closed World Assumption is not made for this representation then one would not be able to conclude that the given formula holds, since it could be the case that there are additional houses in the domain model.

Both the Unique Names Assumption and the Closed World Assumption are *naming assumptions:* they are assumptions about how the set of names in the representation of the domain model corresponds to the set of entities in that model itself. The reason for the use of the word 'world' in the Closed World Assumption is that a domain model for an application is sometimes referred to as a "miniworld" – a very small and very simplified 'world' of a kind.

It is important to realize that a set of formulas, such as the set shown above, *is not a model.* It is a representation of a model, but it is not in itself a model. Please recall that a model consists of a *selection of objects* that occur in the environment being modelled, together with a *selection of concepts* that have no physical existence but which we have constructed to be used for characterizing the environment, for example "colors" and "families." These selected objects and concepts are what we call entities. Besides entities the domain model also contains relationships of various kinds between the entities.

Usually the representation of a domain model is accompanied by statements that clarify whether one or both of the naming assumptions are in force for the representation. There are also cases where these assumptions are applied selectively, for example, for some types but not for others. In some circumstances it is possible to express the naming assumptions in the language of logic, so that they appear in the representation itself and not as an attachment.

4.1.3 Composite Names

The definition of the Common Expression Language in the previous chapter introduced *composite symbols* alongside with atomic symbols. As an example of the use of composite symbols, consider a domain model containing a number of houses, and where in addition the roof of a house is also considered as an entity, with a `part-of` relation to the entire house. One way of introducing names for these entities is to use atomic symbols for each house, for example `house-4,` and to write expressions such as

```
(roof-of house-4)
```

as names for the roofs.

This choice of representation requires that a house can only have one roof, so for some kinds of houses one must accept that "one roof" consists of

several unconnected parts.

For a given representation of a domain model it must always be well defined what are the entity names in the representation. This is required, for example, in order to make sense of whether the naming assumptions apply or not. If all the names are atomic symbols, and composite symbols are not used, then it is straightforward to identify the names that occur in the various formulas of the representation.

For composite names (names that are composite symbols) the matter is a bit more complicated. Rather than enumerating all atomic and composite names, one would prefer to specify the atomic names and the rules whereby composite names are formed. In the example, it is sufficient to specify the atomic names and the rule that for every entity h in the domain model whose type is `house,` the symbol `(roof-of h)` is also a name in the representation. If in addition the two naming assumptions are made, then it is clear that the domain model contains one entity for each of the explicitly named houses, one entity for the roof of each of the explicitly named houses, and no other entity.

Knowing this is important in order to be clear about the meaning of propositions that use universal and existential quantifiers. In a proposition of the form "for all x ..." the variable x will range over the entities just described.

Other solutions must be found if the domain model admits one house to have several roofs, and if it admits roofless houses. Please refer to the Chapter Annex for an overview of possible approaches to this problem.

4.1.4 Time

Sometimes it is useful to have domain models that pertain to one specific point in time, so that its representation describes aspects of the application at that timepoint. In other cases one needs domain models that pertain to a segment of time from a starting time to an ending time, in which case the propositions in the representation must indicate the point in time or interval of time within the domain interval that the proposition applies to. The first type of domain model is called a *snapshot,* the other type is called an *episode.*

In the case of episodes it is natural to introduce explicit *timepoints* as a particular kind of abstract entities and to use them among the arguments in the propositions of the representation. For episodes there is also a frequently occurring need to represent *processes,* i.e. things that happen over an interval of time. The activity where one particular person makes a bus ride from one stop to another on one particular morning could be an example of a process. Processes are an additional type of abstract entities.

With respect to snapshots, on the other hand, there is no need to make use of timepoints in snapshot models since the entire model refers to one single point in time anyway. The information about what point in time the snapshot refers to may be stated outside the snapshot's representation and as a statement about it, and there may also be information within the snapshot's representation stating "what time it is," but statements within the representation do not need to refer to what holds at a variety of times, and therefore there is no need to introduce timepoints as a kind of abstract entities there.

Although additional kinds of entities will be added further on, we shall anticipate the forthcoming additions by showing the top level of the hierarchy of various kinds of entities that will be used in this textbook, and which is as follows.

```
entity
   thing
      process
      individual-entity
         tangible-entity
         intangible-entity
         social-entity
      compound
   temporal-entity
      timepoint
      time-interval
   quality
```

This diagram shall be read so that both tangible entities and intangible entities are included among individual entities, which in turn are included among things, and so forth. Most of the items in this diagram have already been mentioned; compounds are forthcoming; and qualities include colors, for example.

A particular problem arises with respect to individual entities with a limited life-length in an episode. Suppose, for example, that the domain model includes houses and applies for such a long period of time that additional houses may be built during that period, and some houses may be torn down. If the Closed World Assumption is in force and houses have atomic names then there must be separate names for all the houses, and for each house it must be specified whether and when it was built or torn down within the range of the model's duration. This affects the interpretation of propositions that contain quantifiers, since clearly for a proposition that applies to a particular timepoint one would only wish the quantification to range over those entities that exist at that timepoint. This is discussed further in the Chapter Annex. However, for the continued main text we shall disregard this problem and assume for convenience that all individual entities exist throughout the duration of an episode.

4.1.5 Contexts

One of the important aspects of intelligence in people is our ability to imagine situations that are different from the one where we are actually. A person may consider what would have happened if he had missed the bus when going to work the same morning, or what would have happened if he had won a bet. A cognitive agent should have the same capability, both because it is useful in itself, and in order to be able to communicate effectively with people that have and use this capability.

A standard method for realizing this is to introduce the notion of a *context* as a generalization of the notion of timepoint. The basic idea is to introduce contexts as a particular kind of entities, and to arrange that just like one can state that a particular condition holds at a particular point in time, one can state that it holds in a particular context.

In a snapshot-based cognitive agent there is one *known context,* namely, one that labels its model of its environment. In an episode-based agent that is equipped with memory, both the current timepoint and all preceding timepoints are considered as known ones. In a memoryless episode-based agent there is also just one known context, namely, the current timepoint. In addition there are a number of ways of constructing additional contexts from the known ones, in particular:

- *foresight-based contexts,* which are obtained by proceeding one step forward in time from a previously established context;
- *assumption-based contexts,* which are obtained by assuming that one or a few things that hold in a given context do not hold in the constructed one;
- *action-based contexts,* which are obtained from a given context by assuming that a particular action starts there, and where the situation where the action has been completed is the new context. Action-based contexts are usually called *situations;*
- *hindsight-based contexts,* which can be obtained in episodic agents where memory is lacking or incomplete, and where it may be possible to reconstruct some information about earlier timepoints using information about the current timepoint.
- *conditional contexts,* which can be obtained from a context where it is not known to the agent whether a particular condition holds or not. The conditional context makes an assumption concerning the truth of such a condition. A combination of two or more conditional contexts based on the same given context can therefore be used for reasoning about the alternative possibilities with respect to that condition.

Together these types of contexts are called *derived contexts.* These constructions can be used repeatedly, so one can for example define contexts by starting with a known context at an earlier timepoint, assuming alternatives to some facts at that timepoint, and then going forward in time from the resulting assumption-based context.

The representation of an episode consists of a set of propositions (i.e., a theory) that pertain to a particular sequence of known contexts that are related according to the successor-timepoint relation, together with the required signature for this theory. It is therefore natural to generalize the term 'episode' and to define an *imagined episode* as something whose representation contains propositions pertaining to a sequence of contexts that starts with an assumption-based one, and where the following contexts in the sequence are foresight-based or action-based one after the other. The episode that is formed by the known contexts is more precisely called the *real episode* of the cognitive agent in question.

4.2 Representation Structures

4.2.1 Levels of Representation

According to the standard definition, a *first-order theory* consists of a set of axioms using a particular first-order signature, where a *signature* is a

list of predicate symbols and function symbols together information about their arity (number of arguments). (A constant symbol is considered as a function symbol with arity zero).

The materials in Section 4.1 are therefore the beginnings of a first-order theory for domain models. We shall use the word *representation* for the combination of a theory and its signature.

Consider now a few propositions in the example in section 4.1.2, which is of course a first-order theory, albeit a very simple one:

```
[has-type anders person]
[has-attribute anders sex male]
[has-type house-1 house]
```

The predicate symbols `has-type` and `has-attribute` must be included in the theory's signature, and the same applies for the constants such as `anders, person, sex, male,` and so forth. However, these predicates and constants have quite different range, and one can roughly distinguish three *representation levels:*

- The *framework level:* The two predicates, which are intended for use in representations of any kind of domain model.
- The *ontological level:* Entities for types, such as `person,` and entities for standard values of attributes, for example `male.`
- The *observed level:* Entities for tangible objects and other things that are specific to a particular scenario, for example `anders` and `house-1.`

The borderlines between levels is a matter of definition by the designer of the representation. For example, names of countries would be on the observed level in a domain model containing descriptions of countries, but they would be on the ontological level in a domain model for an address register where the country name is merely used as a component of the address. Furthermore, in some cases one may see a need for additional levels besides these ones.

These levels apply to both signatures and theories. A theory on the framework level may only use symbols in the framework signature; a theory on the ontological level may use symbols in the framework and ontological signatures, and so forth. We shall refer to them as a *framework theory, ontological theory,* and *observed theory,* respectively.

Each context has its own signature and its own observed theory. The difference between a derived context and the one it is derived from may concern the observed-level ontology and observed-level propositions. The use of derived contexts that differ with respect to the ontological or even the framework level is a difficult problem and will not be addressed here.

Above the framework level there is the *logical level* which defines the propositional connectives such as `and` and `imp,` as well as the quantifiers. [1]

[1] In a *typed first-order theory* the signature is extended with a specification of a set of types, for each function symbol it is defined what is the type, or possible types, of the values of this function, and the types of the arguments of functions and predicates are specified or restricted. However, although we use a concept of type as already shown in some of the examples, these are not treated as types in the sense of a typed first-order theory, and our types are instead one particular

It follows that a particular domain model is represented as a theory then this theory will be on the observed level. Ideally the observed theory should characterize the domain model completely. Representations on the upper two levels are worthwhile since they can be included in the representations of a variety of domain models. In particular, a cognitive agent should contain an ontological representation which can then be extended into a specific representation for each particular environment that the agent encounters.

The set of domain models that are correctly described by a particular theory on one of the upper levels will be called the *domain space* for that theory.

Chapter 2 introduced the notion of the *beliefset* of a cognitive agent, that is, the collection of beliefs in the agent's internal state. The beliefset includes the representation of the domain model in the agent, but it may also contain other information. In particular, derived contexts and imagined episodes belong to the beliefset but are not included in the domain model. Reflexive information which is obtained when the agent observes its own cognitive behavior should also not be considered as part of the domain model.

4.3 Descriptors

Expressing domain models directly in terms of domain-model-specific functions and predicates is often quite inconvenient, and there are a number of constructs that can be defined within the framework of first-order logic and that facilitate the job of designing qualitative domain models. We shall introduce four kinds of constructs, namely *features, actions, connections,* and *quantities.* The term *descriptor* is used as a common name for all of these, as well as a few additional kinds of constructs that will be introduced later. Descriptors are not considered as entities, but descriptors and entities are together called *items.*

4.3.1 Features

A *feature* is a descriptor that may be assigned a value using a separate predicate, often called the *Holds* predicate. We shall write this predicate as `H` if it is used in the representation of an episode, so that it has a timepoint as one of its arguments, and as `Hc` if it is used for a snapshot. For example, rather than writing

```
[= (color-of house-4) white]
```

in a domain model without time, one may write

```
[Hc (color: house-4) white]
```

where `(color: house-4)` is a feature and `color:` is therefore a feature-valued function. The predicate `Hc` takes a feature as its first argument and specifies that the value of that feature is what is given as the second argument. Features are considered as composite symbols, so the feature-valued function `color:` is a symbol composer.

For tangible entities and other individual entities it is important to maintain the distinction between the entity and its name, which may be an atomic

kind of entities. Technically, therefore, we are using untyped first-order theories.

or composite symbol. For features and other descriptors this is less of an issue since on can usually assume a one-to-one correspondence between the descriptor, as an abstract concept, and the symbol that represents it. In the present work we make such an assumption, and at points we may therefore be sloppy about the distinction with respect to descriptors.

One use of features is for expressing statements like "Mary knows what is the color of `house-4`." In this case it is the feature `(color: house-4)` that is the object of Mary's knowing.

Another, equality important use of features is for representing information that changes over time. In this case one uses the predicate `H` that has three arguments. The first argument is a timepoint or (more generally) any kind of context, and the other two arguments are like for `Hc`. For example, to say that the color of the house `house-4` is white on January 1, 2012, one might write

```
[H tp.2012-01-01 (color: house-4) white]
```

The following notation is used for timepoints in the examples in the present text, although it is not suggested for general use: `tp.2012-01-01` designates the date in question viewed as a single timepoint, so that January 2, 2012 is the next timepoint. By comparison, `ti.2012-01-01` designates the same date but viewed as a time interval consisting of a sequence of timepoints.

If one were to express the change of color over time using the function `color-of` then one would need to provide it with an additional argument for expressing the timepoint, and the same would be true for every other function where the value depends on time. By using features and the `H` predicate one can concentrate the identification of time to one particular predicate.

Snapshots are useful for specific purposes even in dynamic applications. For example, the specification of the preconditions for an action usually apply only to the timepoint where the action starts, which can be considered as a snapshot for the purpose of expressing the preconditions in logic. The predicate `Hc` can therefore be used there.

4.3.2 Actions

The other important construct is the *action* which is often used as the counterpart of a simple full sentence in natural language. Consider for example the sentence

```
John will travel by bus from Stockholm to Uppsala
```

This phrase may be rendered as the following action

```
[travel :by John :using bus :from Stockholm :to Uppsala]
```

Notice therefore that an action is (represented as) a *composite expression* in our terminology; it is an expression that consists of a number of parts, just like a set or a sequence is an arrangement of a number of parts. This expression *does not in itself make any statement,* it is just a description of a phenomenon that may or may not actually occur one or more times. In order to *state* that John will travel by bus from Stockholm to Uppsala tomorrow, one can write e.g.

```
[W ti.2012-01-01 [travel :by John :using bus
                         :from Stockholm :to Uppsala ]]
```

provided that the trip will be made on January 1, 2012. The predicate `W` may be read as "within" and specifies that there is a process of the kind described by the second argument that occurs within the time interval given as the first argument.

The use of tags in the notation for actions is not logically necessary, but quite convenient. It is true that one could write the parameters as arguments for example

```
[travel John bus Stockholm Uppsala]
```

where each argument position is dedicated to a particular purpose. The use of tags offers the following advantages:

- The mnemonic advantage that the tag reminds the reader of the purpose of each of the arguments
- If some of the parameters are unknown then the formulation without tags needs to introduce existential quantifiers, for example as shown below
- If tags are selected so that they have a semantic content, for example tags for the *actor* and the *instrument* of an action, then it may be possible to write axioms that apply to entire groups of verbs and that allow inference from specific occurrences of an action.

The following example illustrates the second point:

```
[exist .c [travel John bus .c Uppsala]]
```

compared with the expression using tags where parameters may be omitted if the value is unknown:

```
[travel :by John :using bus :to Uppsala]
```

4.3.3 Connections

Connections are expressions that represent an actual or possible relation between some entities, or a relation between an entity and some kind of quantity (including a number or string). The following are some examples of connections.

```
[adjacent Germany Poland]
[between Germany Switzerland Italy]
[in-front-of Lars Anders]
```

Therefore a connection is a record (in the sense of the KRE syntax in Chapter 3) that consists of an operator and some arguments. (Parameters may also be used). The formal status of connections *differs* depending on whether one is modelling a snapshot or an episode. In snapshots it is natural to consider connections as literals, which means that the connection operator is a predicate. In episodes, on the other hand, one has to distinguish between *static* and *dynamic* connection operators, where the dynamic ones are those where the connection may change between true and false in the course of the episode. Static connections can be considered as literals, like for snapshots,

whereas dynamic connections must be considered as terms from the point of view of logic.

Notice that the classification as static or dynamic applies to the connection operator and not merely to each connection formula. If you have an operator, such as `between,` where the literals with it change over time for some arguments but not for others, then the operator as such is dynamic and all usages of it must be considered as such.

The predicate `D` can be used for expressing that a dynamic connection holds during a period of time, for example

```
[D 14.26 14.31 [in-front-of Lars Anders]]
```

4.3.4 Relationships

There are some similarities between actions and dynamic connections, which is also why it is natural to let the predicate `D` accept both types as arguments. We therefore introduce the term *relationship* that includes both actions and dynamic connections, but not static connections. This use of the term 'relationship' is consistent with its use e.g. in the so-called entity-relationship model (which is described in this chapter's Annex).

A relationship is merely a construct: it is an aggregation of a number of components, but it does not have any inherent significance besides this. Every domain model must have its own signature, i.e. its own set of operators for forming these expressions (such as `travel` and `in-front-of` in the examples above) as well as its own set of conventions for the order of arguments and the use of tags in them.

The *meanings* of these operators and tags must be defined using ontology-level axioms that include both literals (elementary propositions) and complex formulas.

Dynamic connections are similar to actions in the sense that they may apply during a period of time within an episode, but connections are different from actions since they describe a condition that is constant during the applicable interval. By contrast, actions are used for representing the change of one or more conditions during their time intervals. Therefore, with respect to the phenomena that they characterize, the connection expresses that these phenomena are equal at the beginning timepoint, the ending timepoint, and all timepoints in-between, whereas this is not the case for actions. For most actions the state of the model at the ending time is different from the state at the starting time.

Notice also that both actions and connections are generic concepts that can *apply to* particular intervals of time, but the time interval is not intrinsic to the action or connection. The *same* action can occur several times; each time it occurs is an *occurrence* of the action, and each occurrence is a *process.*

Another difference is that if a dynamic connection applies during an interval of time, then it also applies to a sub-interval of the given interval. Connections are therefore said to be *homogenous.* By contrast, an action can not occur in two different but overlapping intervals. Different instances of an action may have one timepoint in common, so that one begins exactly when the other one ends, but not more than that.

4.3.5 Quantities and Names

Quantities can be simple or composite. Strings and numbers are simple quantities. Composite quantities are used for expressing various kinds of metrics or other coordinates, or even for symbolic naming systems such as the names of persons. They will be represented as records in the present textbook series. For example, a date in the Gregorian calendar might be written as

```
[Gdate :year 2011 :month 01 :day 05]
```

Other kinds of quantities may be expressed in similar ways, for example geographical locations expressed by latitude and longitude in degrees, minutes and seconds, or street addresses, or even personal names (allowing for regional differences for how to form person's names). In this terminology the term "quantity" has therefore a quite broad scope.

4.4 Modelling Qualitative Change

4.4.1 Component Models and Process Models

Qualitative models for episodes are of two major types: *component models* and *process models.* In component models the domain is characterized as a collection of entities, a set of features for each entity, the values of the features at each point in time, and constraints that specify the interdependencies between the values of these features. For example, a simple component model for a room in a house may contain entities for a particular lamp and a particular electric switch; one feature can represent whether the lamp is on or off, and another feature can represent whether the switch is in the on or off position. There can be a constraint that specifies whether the lamp is on or off as a function of the position of the switch.

In process models, on the other hand, one uses entities, features, and constraints in the same way as in component models, but in addition the domain model allows the use of *processes* which is one additional kind of entities, besides those described above. Each process is active during a period of time and involves some of the features; it influences the values of some of them, usually depending on the values of some other features at the same time.

Notice that in simple cases the closed world assumption will apply to process entities, and that in more complex cases it may not necessarily do so.

Processes can be described on two levels of detail. The *operational level* specifies constraints between the features of the process at each timepoint in the duration of the process; the *effect level* specifies or restricts the values of the features at the end of the process' duration as a function of their values when the process started. The use of the effect level is quite important since it makes it possible to predict the effect of a sequence of actions without having to consider the details of each of them, and therefore it is the basis of action planning. However, the operational level is also important, in particular in robotic systems where it is the basis for the design of the controllers that are needed for controlling the processes.

Process models are used in several branches of Knowledge Representation, in particular for action planning, in cognitive robotics, and for reasoning about mechanical devices. The word 'process' is used by the latter area whereas in temporal reasoning and reasoning about actions one usually prefers the word 'action' or 'event'. None of these terms is ideal. The problem with the word 'event' is that it is also extensively used for a momentary phenomenon, such as the change of value of one feature, or (in the BDI terminology) for a kind of low-level message. The word 'process' is fine for mechanical systems but seems awkward if one applies it to things like John's trip to Uppsala. The word 'action' is fine for things that are performed by persons or animals, but it is instead awkward for processes that arise by causation, for example a snowstorm, a car accident that is caused by that snowstorm together with the negligence of the driver, or the resulting traffic jam.

In the present textbook series we will use both the words 'process' and 'action' and define them so that a process is an instance of an action. Thus the expression

```
[travel :by John :using bus :to Uppsala]
```

represents *one action;* each time that John makes such a trip is *one action instance,* and a process is the same thing as an action instance. Actions that do not have an animate agent, such as

```
[volcano-eruption :at mount.etna]
```

will be referred to as "actions by nature." Notice the use of the tag `at` rather than `by` here, since the tag `by` is reserved for indicating the agent.

4.4.2 Preconditions and Effects of Actions

Both component models and process models use *constraints* for characterizing the dependencies between the values of different features; process models also use them for characterizing the dependencies between the occurrence of processes and the values and change of values of features, and even for the causation from one process to another.

In order to write these constraints one needs a predicate that specifies that a particular action instance, or process, takes place during a particular period of time. The `W` predicate that was used for an example above is not the simplest possible one. The primitive predicate is `D` that is used as

```
[D .s .t .a]
```

for stating that a process that is an instance of the action `.a` begins at time `.s` and ends at time `.t` The action `.a` is a relationship expression formed in the way that has been described above. If this statement holds then the proposition `[W .i .a]` holds for every interval of time that contains or is equal to `[ivl .s .t]` i.e. the closed interval of time between `.s` and `.t.`

There exist minor variations to this notation, for example, defining `D` as a predicate with two arguments where the first one is an interval defined by a starting time and an ending time. Some authors do not use the predicate `D` and instead consider each verb as a separate predicate with timepoints or an interval among the arguments.

It is natural to use both the predicates `H` and `D` for dynamic connections, so that `[H .t .c]` expresses that the condition `.c` holds at the timepoint in question, and so that `D` states in the obvious way that the condition holds over a given interval of timepoints or situations.

In order to draw conclusions about the effects of a process one needs *action effect axioms* which are typically formed as an implication with a literal for the predicate `D` among the antecedents, as in the following example.

```
(imp [D .s .t [paint :obj .h :as .c]]
     [H .t (color: .h) .c] )
```

This axiom says that if the object `.h` is painted with the color `.c` during a time interval from `.s` to `.t` then the color of that object at time `.t` is `.c`.

Effect axioms describe the *normal outcome* of an action. In addition, almost all actions can *fail* in which case the `D` predicate does not apply and the effect axiom is not applicable. Constructs for describing action failure are introduced in the Annex of this chapter.

Moreover, action effect axioms must be complemented with *precondition axioms* that specify the conditions that must be satisfied for the action to be feasible at all. As an example, suppose we wish to specify just one condition for the `paint` action, namely, that the person that does the painting "possesses" paint with the color in question at the starting time of the action. We may now adjust the notation so that it is the paint, rather than the color of the paint that is the parameter of the verb `paint,` and we need to use the function `color-of` with the bucket of paint as its argument. The modified effect axiom is

```
(imp [D .s .t [paint :obj .h :with .p]]
     [H .t (color: .h) (color-of .p)] )
```

Preconditions are specified using the predicate `P` that takes a timepoint (or, by way of generalization, a context) and an action as arguments, and that states that it is possible to start execution of the action at the given time. The letter `P` may be read as "precondition" or as "possible." The simple precondition for the `paint` action can be written as follows:

```
(imp [H .t (possesses .a) .p]
     [P .t [paint :by .a :with .p]] )
```

This is a very simplified example, of course, and for a less imple one one would need several additional preconditions, beginning with one stating that `.p` is in fact a bucket of paint.

Continuing the same example, if one actually wishes to use the color rather than the paint as the parameter of the painting action, still using the tag `with` then one would have to write the precondition expression as

```
(imp [exists .p (and [H .t (possesses .a) .p]
                     [= (color-of .p) .c] )]
     [P .t [paint :by .a :as .c]] )
```

4.4.3 Expressing Action Effects Using Situations

One way of using action effect axioms, as exemplified in the previous subsection, is to choose the values of the `.s` and `.t` variables as timepoints,

represented for example as integers. Another possibility is to choose them as situations, i.e. as action-based contexts which were introduced above. This requires the introduction of a situation successor function `succ` where `(succ .s .a)` is the successor of the situation `.s` that is normally obtained after the action `.a` has been executed there. This function is related to the `D` predicate using the following *situation execution axiom:*

```
[D .s (succ .s .a) .a]
```

The situation execution axiom can be combined with an arbitrary action effect axiom (provided that the latter is written in a standard way) and obtain an action effect rule that is expressed directly in terms of situations. The first example above thus becomes

```
[H (succ .s [paint :obj .h :as .c]) (color: .h) .c]
```

which can be read as "in the situation that normally results from the action of painting the object *h* with the color *c,* the color of *h* will be *c.*"

Some methods for action planning and for reasoning about actions use time-points together with action effect axioms that are expressed in terms of the `D` predicate, like in the first example. Others use the formulation in terms of situations and the `succ` function. This approach is known as the *situation calculus.* These alternative approaches will be described in Volume II of the present work.

4.4.4 Static Preconditions

Although in principle it would be possible to express everything using the predicates `H` and `D,` in practice it would be quite inconvenient to do so. Some basic kinds of information are better expressed using additional predicates, which are then the realizations of static connections as discussed above. This applies for example for the predicate `has-type` which may be used like in

```
[has-type house-4 house]
```

in order to specify what is the type of the entity `house-4,` or

```
[has-type bucket-4 bucket-of-paint]
```

with obvious meaning. The type is in turn an entity that also has a type, and so forth; in this generic representation we assume that every entity has a unique type.

This representation is only adequate if we can assume that the type of something does not change over time and that it is not the subject of knowledge or belief. This means that the type concept shall only be used for basic distinctions, like distinguishing between persons, buildings, fruits, and so forth. It shall not be used for more detailed distinctions, like "house where someone is living" versus "deserted house," since this may change over time and may be subject to incorrect belief.

The decision that a particular connection operator is static is often due to a simplification that is made by the designer of the domain model. For almost every kind of fact that one can think of, it is possible to imagine situations where this fact changes over time, or that there are situations where someone has a mistaken belief about that fact. However, one important aspect of

practical domain modelling is that one has to restrict the generality of the representation to what is needed in practice.

4.4.5 Actions with Extended Duration

The effect-level description of an action can often be made by specifying its precondition and its postcondition as logic expressions. In some cases it is sufficient to combine this effect-level description with a procedural definition on the operational level, [2] but in other cases it is important to also use logic within the effect-level description. When cognitive systems are used for controlling robots or other physical systems, for example, one commonly needs actions that have extended duration in time and that can occur during overlapping time intervals, and in such cases it is important to be able to characterize intermediate points during the execution of an action in terms of their logical properties.

Actions with extended duration and effect-level description of actions has been relatively little studied in the research on Knowledge Representation, but it is quite important from a practical point of view. This topic will not be further considered in the present volume but we shall return to it in Volume II.

4.4.6 Representing the Creation and the Destruction of Objects

Returning to the entity `house-4` in previous examples, consider the action of destroying that house so that it does not exist any longer. If a domain model shall allow the representation of such events then it must make it possible to specify a *time of destruction* for each entity. Similarly, since there may be actions for building a house, or other actions that result in the creation of a new object, there must be a possibility of specifying the *time of creation* of an entity.

The binary connection operators `created-at` and `destroyed-at` are used for this purpose. It is natural to consider them as static connections, so that they are manifested as predicates, but they may also be considered as dynamic connections in case the application at hand requires the representation of alternative beliefs about the time of creation or destruction of particular objects.

Several points of view are possible with respect to the times of creation and destruction of reified relationships, and in particular for reified actions. (Reification is introduced in the next section). One possible view is that the starting-time and the ending-time of the action are the times of creation and destruction of the reification.

It is easy to see that if a cognitive agent maintains representations of both its own real episode and one or more imagined episodes (cf. Subsection 4.1.5) in its beliefbase, then there must in general be one signature that is shared by these episodes, but in addition each episode must be able to have its own addition to the shared signature. This is because some of the

[2] The distinction between operational level and effect level was introduced in Subsection 4.4.1.

imagined episodes may contain actions that have the creation of objects among their effects, and if the Closed World Assumption applies then the signature must be extended with additional constant symbols which are by definition specific to the episode in question. This explains the reasons for the chosen definition of imagined episodes above.

4.5 Reification

4.5.1 Reification of Relationship Expressions

Relationship expressions can not in themselves be included in the domain theory since they are terms, and not literals; verbs and other operators that form relationship expressions are not predicates. Therefore it is not possible to draw any conclusions from relationship expressions in themselves, but only from literals that are formed using predicates and where relationships occur as arguments.

This is quite sufficient in many cases, but there are also situations where one wishes to *reify* a relationship, and in particular an action, that is, to introduce an entity that *designates* the process that is the instance of the action at hand. In the example relationship expression

```
[travel :by John :using bus :from Stockholm :to Uppsala]
```

one can imagine situations where it is desired to specify additional information about this particular trip, for example, what were its consequences, how was it witnessed and reported, and so forth. It would not be appropriate to express that information using additional literals where the relationship expression occurs as one of the arguments since that would be ambigous: John may have made several bus trips from Stockholm to Uppsala.

In fact there are two kinds of reification for an action, namely, the generic reification and the reification of instances of the action. The generic reification is used for making statements such as "eating animals is wrong" – the generic reification introduces an entity that stands for the action in the abstract, and that can be used as the argument of a variety of predicates. Generic reifications are important for characterizing the desires of a cognitive agent.

The instance reification, on the other hand, allows the cognitive agent to introduce an additional entity into its domain model with a previously unused name, for example `travel-142` for the trip in question, and to use that entity for those additional statements. It will then be up to the system to assure that different trips are given different names, and that (to the largest extent possible) the same name is used every time one specific trip is being considered. Such an entity is called a reification of the given expression.

There must then also be a proposition that specifies the relation between the reification and the relationship being reified. This may be done using a single predicate `designates,` for example

```
[designates travel-142 [travel :by John :using bus
                            :from Stockholm :to Uppsala ]]
```

However another possibility is to convert each tag in the relationship to a binary predicate, and to write the defining information for the reification as follows

```
[hasverb travel-142 travel]
[by travel-142 john]
[using travel-142 bus]
[from travel-142 Stockholm]
[to travel-142 Uppsala]
```

The binary predicates that are introduced in this way are static predicates since they do not admit any change over time and there is no reason for them to do so. If one does not wish to introduce many separate predicates of this kind then one alternative is to introduce a single, ternary predicate for this purpose, so that the translation becomes

```
[hasverb travel-142 travel]
[param travel-142 by john]
[param travel-142 using bus]
[param travel-142 from Stockholm]
[param travel-142 to Uppsala]
```

The representation using several different, binary predicates is appropriate if one is using a computational framework that precisely supports binary predicates, such as Description Logic which is introduced in Volume II. The representation using a single, ternary predicate is instead appropriate if one's software does not have any particular preference for binary relations, and if one is able to state properties that apply for the `param` predicate in general or for specific choices of its second argument.

4.6 Representing Part-Whole Relations

The relation that holds between a physical object and each one of its parts is important in very many domain models. In principle it is very easy to represent it: we simply need a connection former `is-part-of` between the part and the whole. Like for other connection formers, the uses of `is-part-of` may be specified to be static or dynamic, and they will accordingly be represented as literals or as relationships. In either case it is natural to consider this as a transitive relation [3] since it is not always possible to specify an "immediate part of" relation in terms of layers of decomposition.

It is also natural to use a predicate `is-separate` that holds between two objects if neither of them is a part of the other and they do not have any common parts. The negation of this predicate may be used for describing connectors in the sense of parts that are partly inserted into two or more other, separate parts.

When dynamic part-whole relationships are used it becomes possible to define a sequence of actions where one part after the other is replaced in a given object that consists of many distinct parts. This introduces a classical philosophical problem, namely, at what point does the object cease to be "the same" object as one part after the other is replaced in it. Some

[3] The notion of 'transitive' needs a separate definition for dynamic connections but it is obvious and does not offer any difficulties.

answer must be given to this question if the representation described here is used, since we have been assuming that each object is represented by a particular entity. If an object is considered to have become another object by the replacement of parts then another entity must be introduced in the representation.

One natural solution to this problem is to identify some indivisible part of the object as the one providing the identity, so that other parts may be replaced while retaining the same entity as name for the object. Another approach may be to consider *every* replacement of a part as the destruction of the old object and the creation of a new one. This requires however that there are ways of keeping track of the relations between the successive "objects" that result from this convention.

4.7 The Domain Modelling Language

The Reference Annex for the present chapter contains an overview of current *representation languages* which have been developed for use in knowledge-based systems or for expressing formal ontologies. In that context it will be useful to also have a name for the representational conventions that have been used in the present chapter, consisting of predicates such as `H, D` and `designates` and functions such as `succ.` The incomplete definitions in the present chapter do not warrant being called a language, but there is in fact a precise language definition for the *Domain Modelling Language* that contains the notations used here.

Chapter 5

The Cognitive Agent Executive

In Chapter 2 we defined a cognitive agent as a system that actively chooses its actions using a model of its environment, and introduced the basic architectural concepts for such systems. Chapters 3 and 4 introduced notation and techniques for modelling the agent's environment. In this chapter we shall return to the question of the agent's architecture, and in particular to the question of how the agent may select and execute its actions.

5.1 Agent Executive and Behavior Structure

Actions may be invoked in a number of ways, some of which are quite conventional, such as according to a direct command by the user, or when an action is part of a script that defines a higher-level action. The cognitive character of the agent adds some other ways, in particular when an action is selected and invoked in order to achieve a goal that is part of the agent's goal structure. Finally, actions may be invoked in response to incoming events and according to stimulus-response rules that specify appropriate actions when a particular type of event occurs in combination with a particular internal and external state for the agent.

The top-level loop of a cognitive agent must have a structure that is suitable for satisfying these needs, and this is the reason why it differs from conventional high-level software. Please recall that many conventional computer programs perform in principle the following loop

```
repeat
   1.  Wait for the next input
   2.  Act on the received input
   3.  Communicate the results of step 2
end repeat
```

which means that if there is no input then the process is at a standstill. By comparison, the top-level loop of a cognitive agent is in principle as follows

```
repeat
   1.  Receive the current set of input events
   2.  Deliberate on the current state and current inputs
   3.  Execute action(s) that resulted from step 2
end repeat
```

The set of input events in step 1 may be empty, but the agent will then deliberate anyway and actions may result. The deliberation step may also involve the choice of spontaneous decisions, and it may result in the following:

- Decisions to perform particular action(s) immediately
- Update of the agent's environment model
- Update of the agent's behavior structure.

Incoming events may of course lead to a change in the agent's model of its own environment; this is the only way that that model can change. [1] At this point we shall however discuss the *behavior structure* in the sense of *that part of the agent's internal state that does not represent a model of the environment,* and which is used instead for defining its behavior. The *executive* of a cognitive agent will be defined as the top-level loop together with the procedures that are directly invoked by it, for example the deliberation step shown above.

5.1.1 Agenda and Task Network

Two things are particularly important in the behavior structure and are used frequently, namely, an *agenda* and a *goal structure.* The simplest form of an agenda is just a set of actions that the agent has committed to doing in the future, and usually in the near future. The deliberation step may add items to the agenda, and it may pick items from the agenda and forward them to Step 3 in the top-level loop for execution. (It may also discard agenda items without execution). This simple scheme may be used, for example, so that the deliberation step generates more actions than what the agent will be able to perform, and then it evaluates them for priority and benefit and actually performs only the best ones according to some merit function.

A more sophisticated form of agenda is the *hierarchical task network,* which is often described by the acronym HTN. In this case a "task" is an instance of an action that may or may not have been executed already. For example, if the agent has a plan to perform action A, then action B, and then action A again, then there are two A tasks and one B task in the plan.

A hierarchical task network contains a set of tasks and two kinds of relations between them. There is a *temporal precedence relation* saying that one task must be performed before another one, and a *subtask relation* saying that one task is intended as a step in the execution of a larger task. Additional relations may also be included, but these are the basic ones.

[1] To be precise, what the agent can really change in a concrete sense is its *representation of* its model but we use the shorter wording since there can not be any misunderstanding.

A cognitive agent that uses an HTN as a part of its internal state may use the deliberation step in its main cycle for a number of operations on that HTN. It may add one more top-level task to the HTN, i.e. a task that is not there as a subtask of an existing task. It may also plan the execution of an existing task by adding a set of subtasks that define the given task, together with precedence relations between those subtasks as well as with other, previously existing tasks in the HTN. In these ways the deliberation process may plan the agent's future course of action a few steps ahead.

Moreover, and more interestingly, the deliberation process may check whether the preconditions for a particular task in the HTN are satisfied, according to the information about the state of the agent's environment that is available through the agent's environment model. If they are not then the deliberative process may choose to insert additional, preparatory tasks that achieve the preconditions of the given task so as to enable it. Alternatively, the deliberation process may identify alternatives to the problematic task. For example, if that task is a subtask of a higher-level task, then maybe there is another set of subtasks that will achieve the higher-level one and where the preconditions are in fact satisfied.

The possible operations on an HTN include also the operations that were mentioned for a flat agenda, such as deletion of tasks and priority ordering of tasks before execution.

5.1.2 Desires, Goals and Intentions

Chapter 2 defined the notions of *desires, goals,* and *intensions* which can be used for directing a cognitive agent's behavior towards some long-term needs and restrictions. Please recall that a goal is defined as a desire that has been adopted for active pursuit by the agent, and it is assumed that the agent's goal adoption mechanism operates under the restriction that the set of its current goals at any one time must be consistent.

One way of using goals is as a *goal filter* on a hierarchical task network. In this case the agent's deliberation process is organized to generate proposed tasks in response to incoming events, but these proposed tasks are then checked against the agent's current set of goals, and actions that are inconsistent with the goals are not adopted in the HTN, which means that they will not get to be executed.

A goal filter will be particularly effective if it does not merely check the direct effects of an action against the goal filter, but if it also extrapolates the indirect effects of an action according to cause-and-effect rules (this is then a kind of imagination mechanism), and checks whether they may be inconsistent with the current goal set.

Notice, in this context, that there may also be introspective events, i.e. events that report on observations of the agent's internal state, including its behavioral structure. For example, there can be an event that reports that the HTN contains two tasks that can not both be executed, for example because each of them will require and consume a resource that exists in only one copy; such an event may generate a suggestion for an internal task that removes one or the other of the given tasks, and this internal task will then be checked for whether that task and its indirect consequences are goal-consistent.

Using goals for a goal filter is worthwhile if the agent's behavior is predominantly event-driven and goals are relatively easy to formulate and to apply. However, there can also be situations where goals are more complex and where events are relatively rare, which has the effect that the agent's behavior will mostly be goal-driven. This means that the agent has some standing, overriding goals, and it uses these goals for generating tasks whenever opportunity arises for a positive goal (something to be achieved) or a danger arises for a negative goal (something to be avoided).

5.1.3 Planning and Plan Revision

The capability for *making and executing plans* is often quoted as an essential aspect of intelligence. An agenda is essentially a plan that a cognitive agent updates continuously, and that specifies its own forthcoming actions. A simple variant of the executive, is therefore one where an input event may specify a goal that the agent is supposed to achieve, and where the `deliberate` step constructs a plan (i.e. a sequence of actions) for achieving that goal, then inserts the goal into the agenda, and finally extracts the action that has to be performed first from the current agenda. The purpose of using an executive with this structure is that the agent shall be able to handle several goals concurrently, so that the agenda can maintain several plans and decide how to execute them, either sequentially or in an interleaved fashion.

The notions of *desires, goals,* and *intentions* for organizing an agent's cognitive state are relatively elaborate concepts and are not needed for the simple executive that was outlined in the previous paragraph, but they are used in the *BDI executive* that will be described later in Section 5.3.

5.2 Execution of Actions in an HTN

It is convenient to separate the definition for executing an action into three parts, namely its *formal precondition,* its *performance script,* and the *presentation routine* for its outcome. The precondition specifies conditions that must be satisfied in order for the action to be performable; the performance script specifies how to execute an action when its formal precondition is satisfied.

The formal precondition need not be exhaustive or, more precisely, it is not always possible to make it exhaustive, so it may happen that the execution of the performance script *fails* even though the formal precondition was satisfied. However, formal preconditions are anyway useful for the cognitive agent's operation since they can be used for planning and for prediction. One is entitled to consider it as a default that the action succeeds if its formal precondition is satisfied.

5.2.1 Response to Precondition Failure

If an agent has committed to performing a particular action, for example due to a command by the user, or because it is part of a plan that the agent has decided to perform, and if the formal precondition of that action is not

satisfied, then there are two possible courses of action as already discussed above: make first some other actions that *achieve* the formal precondition, that is, they make it come true, before performing the intended action, *or* retract the intention to perform the action in question and consider some *substitute action.* Doing nothing i.e. dropping the intention altogether is one possible substitute action.

The mechanism for achieving a failed precondition may be organized as follows. Preconditions are expressed as propositions (i.e. as logic formulas) that are associated with each verb. They can often be expressed naturally as a conjunction of literals, and the violation of a precondition is then often due to the failure of one of those literals. In such cases one may associate one or more *achievement methods* of some kind, for example action plans, with each predicate that may occur in preconditions. Each achievement method for a given predicate should specify a sequence of actions that may be executed in order to make a literal with the given predicate become true.

5.3 The BDI Architecture

The *BDI architecture* [2] is a proposed cognitive architecture for goal-driven cognitive agents. The abbreviation BDI stands for "Belief, Desire, Intention" which were defined in Chapter 2. This architecture is a standard reference which is used both as the basis for theoretical work, and as a guideline for actual implementations. It should be realized however that different implementations of this architecture may differ considerably, so one should not think of it as a fixed specification or as a standard – it is more like a common frame of reference.

5.3.1 The BDI Top-level Loop

The standard top-level loop in the BDI architecture is as follows.

```
initialize-state
repeat
   options := option-generator(event-queue)
   selected-options := consider-and-select(options)
   update-intentions(selected-options)
   execute()
   get-new-external-events()
   drop-unsuccessful-attitudes()
   drop-impossible-attitudes()
end repeat
```

The value of the variable `event-queue` contains events that have arrived in the incoming events channel. It is maintained throughout the session with the cognitive agent and is initially an empty set or queue. The term `attitude` in this script is used as a common name for desires, goals and intentions. The `consider-and-select` operation is sometimes called `deliberate` in presentations of this script.

[2] Rao, M. P. Georgeff. (1995). "BDI-agents: From Theory to Practice". Proceedings of the First International Conference on Multiagent Systems (ICMAS'95). https://www.aaai.org/Papers/ICMAS/1995/ICMAS95-042.pdf

At the beginning of every cycle, the option generator reads the event queue and returns a list of *options,* that is, things that may become intentions. Next, `consider-and-select` selects a subset of the options to be adopted and adds these to the intention structure. If there is an intention to perform an elementary action at this point in time, then the agent executes it. Any external events that have occurred during the interpreter cycle are then added to the event queue. Internal events are added as they occur. Next, the agent modifies the intention and desire structures by dropping all successful desires and satisfied intentions, as well as impossible desires and unrealizable intentions. [3]

Operations such as `consider-and-select` and `option-generator` make use of the agent's beliefbase. The selection of intentions and plans may take standing desires and goals into account in order to increase or decrease the estimated merit of adopting that intention or plan.

5.3.2 Choice of Intention and Plan

The operation `option-generator` in the top loop script depends on the application and has to be defined in its own way for each of them. We therefore proceed to discussing the next two operations in the script, that is, `consider-and-select` and `update-intentions.`

Going back to the definitions, intentions are "desires to which the agent has to some extent committed" and a plan is "a sequence of actions." In a simple interpretation of this, an intention may be for example "go to the train station" and a plan may be "walk to the bus stop, wait for the first bus to the train station, go with that bus." The agent may contain a *plan library* consisting of fixed plans, and with information about which plans are appropriate for which intention. When performing the top loop script it needs to look up available plans for a selected intention, choose between them, and use one for the step called `execute.`

At some points the agent has to make a choice between different possible intentions, such as "go to the train station" or "cancel the train trip." This choice should be based on the desires and goals that the agent has established in its internal state. The definition of the top loop script is not clear about whether the choice of intention, and the choice of plan for a given intention shall be included in `consider-and-select,` in `update-intentions,` or even in `execute.`

The design choice in this respect depends on whether the choice of intention and the choice of plan are interdependent or not. Suppose the agent has the option of "going to Stockholm by train over the day" or "cancelling the train trip." It selects the former option, i.e. turns it into an intention, based on the fact that it is in line with some of its goals, such as seeing football (soccer) matches. Then it considers the possible plans for this. The first step of the plan will be going to the train station, and there are several subplans for this in the plan library, including walking, taking the bus, and taking a taxi. Suppose now the agent considers all the plans, decides that each of them has significant disadvantages, and then changes the intention

[3] This paragraph is a quotation from the Wikipedia article for the BDI Architecture, with minor modifications.

to the second one and cancels the trip. How is this possible in the framework of the main BDI cycle?

One possibility is to arrange that choice of intention and choice of plan are done in sequence. Then the choice of going to Stockholm is done in the `consider-and-select` step, and the `update-intentions` step consists of extending the structure of selected intentions by adding information to them, in particular, adding an appropriate plan for the selected intention. If no plan can be found then the intention *fails* in the execute step, so nothing happens in this respect, and it is removed in the operation `drop-impossible-attitudes`.

Another possible realization of the top loop script is to let the `consider-and-select` step consist of more things: consider the newly generated options, identify possible plans for each of them, and evaluate each of them with respect to both the importance of the intention (i.e. how it relates to one's desires and goals) and the quality of the considered plan with respect to those desires and goals. Then `update-intentions` consists of making a choice between the intention/plan pairs based on this evaluation.

The top loop script shown above can therefore be understood and implemented in more than one way. When one is actually designing the software for an intelligent agent one may use it as a first outline of the design in a top-down design process, but it must be complemented by a number of additional design decisions.

5.3.3 Alternative Models for Plan Execution

Although the top loop script separates choice of plans from execution of plans, it actually only provides structure for the former activity, since plan execution is represented by the simple operation `execute`. The exact character of plans, the method for executing plans, and the relation between intentions and plans must therefore be clarified, and these questions may be answered differently in different uses of the top loop script.

Consider therefore the situation where the agent has committed to one particular intention and one particular, non-atomic plan for realizing that intention. Assume for simplicity that the plan is also sequential. One way of defining `execute` is to say that it will perform the entire plan in one go, so that the `execute` operation has finished when all the steps in the plan have been performed, and then the top loop script can proceed to its next step.

This is an appropriate definition in some cases, but not always. It is problematic if the plan execution takes a certain time, as is often the case for mobile robots, for example, and for two reasons: first, it precludes any activity with the other steps in the top loop script while the robot is performing its actions, such as moving from one place to another, and secondly it does not facilitate the handling of problems that may arise during the execution of the plan, namely, if something should go wrong then.

An alternative arrangement is to let the interpreter for the top loop script maintain the execution stack for each plan being performed, so that the `execute` operation means "advance one step in the current plan, perform one more action, but no more." This has the effect that all the other steps in the top loop script are executed once again between each step in the plan and

its subplans on all levels. This solves the mentioned problem but introduces another one, namely, that it may slow down the system considerably. We shall refer to this as *BDI-aware plan execution.*

A third possibility is to use multiple *execution threads* as provided by the operating system or the programming language at hand, so that the execution of a plan can continue in a separate thread and the top loop script can proceed in the original thread. In this case the definition of `execute` in the script shall really be understood as `invoke` since the only thing that happens before the script proceeds to its next step is that the execution of a plan is *started.*

The separate-thread approach has several advantages, but it is less easy to implement and it requires certain extensions to the basic model in order to handle the situations that arise when the execution of a plan *fails* without having achieved the intention that the plan was started for. The failure of the plan should then generate an event, in the sense of that term in the BDI model, in order to communicate the failure back to the top loop. This event should be taken care of there by a reconsideration of the agent's intentions, for example, cancelling the intention in question since it could not be realized, or retaining the intention and trying again with another plan.

A similar thing should in fact be possible in the system even if a plan proceeds according to its definition but the `execute` operation recognizes that the operation takes unexpectedly much time or other resources. In such cases should the plan also generate a warning event that can be picked up in the top loop, and that may cause it to discontinue the execution of the plan in favor of some other action.

5.3.4 A Restricted BDI Model

Georgeff and Rao propose in their article about the PRS system that the top loop script and its description is an idealized definition that corresponds well to the underlying psychological theory, but they also observe it is not a practical system for rational reasoning. The authors propose the following restrictions in order to obtain reasonable performance and a manageable system:

- Consider only beliefs about the current state of the world. Each belief is a literal in the sense of logic, that is, an expression consisting of a predicate and its arguments, or the negation of such an expression.
- Represent information about the means of achieving certain future world states as a library of *plans.* Each plan has a *body,* an *invocation condition* that specifies what intentions the plan in question may achieve, and a *precondition* that specifies what must hold in order for the plan to be executable.
- The system forms an intention by adopting a plan, and in each case it creates a separate process for executing that plan, allowing for the plan to contain invocation of subplans.

The first restriction says essentially that the cognitive agent is not equipped with the capabilities of foresight and hindsight.

5.4 A Persistence-Aware Executive

The generic BDI executive of the previous section is fairly abstract, which is also why we have discussed the various possible interpretations of the steps in its executive script. The present section will describe an executive that conforms to the general ideas of the BDI executive, but which is more concrete in several ways. It describes the use of *behavior rules* that are associated with the agent's desires and which are invoked and used in several of the steps in the BDI top loop. It also allows for differences with respect to persistence, so that some desires may apply only briefly and others may apply in a lasting way. This design will be referred to as the *Persistence-Aware Executive,* PAE.

The design of the Persistence-Aware Executive will be illustrated with examples from a simulated Zoo scenario, where there are animals, wardens, meadows and cages for the animals, paths for walking between the meadows, and so forth. We make the following simplifying assumptions:

- We use only features, and not connections, for representing changeable conditions in the model
- All statements in the environment model are correct, although the model will of course always be incomplete. For example, sensor readings are assumed to be correct and up-to-date.

The reason for using this particular executive, rather than one of several other possible choices, is that it allows us to illustrate the use of several of the constructs for domain modelling that were introduced in Chapter 4.

5.4.1 Duration of Actions and Features

Long and Short Actions

The Persistence-Aware Executive allows a distinction between *long actions* and *short actions.* A long action consists of several *steps* that are normally performed in sequence, but it is possible to intersperse one or more short actions between successive steps in a long action, or even perform them concurrently with long-action steps.

An example of a long action in the Zoo scenario is the *promenade* where a warden goes from one waypoint to another, or where he makes visits to a number of animals in their different locations. Short actions during a promenade may include curing an animal, taking out an umbrella, scaring away a wasp, or answering a question by a customer.

The distinction between long and short actions is useful since it allows one to define an agent behavior where the agent acts towards an overall goal, namely, the goal of the current long action, and where anyway it is able to adapt its behavior to the situations that are encountered. Notice that this is not a universally useful ontology; it is an actively chosen modelling restriction.

Persistent and Transient Features

The PAE also identfies some features as being *persistent* in the sense that they retain their value until the value is changed for some specific reason, which is usually due to the occurrence of an action. Most actions have the effect of changing the value(s) of some feature(s). Other features are *transient* in the sense that they have one or more *stable values* that may be considered as the normal ones, and if they are changed to a non-stable value at some point in time then they return spontaneously to one of the stable values, usually already in the next timepoint. We refer to non-stable values as *transient* ones for transient features. For persistent features all values are stable.

Persistent attributes can be used for representing phenomena that we think of as actions but where there are no distinguishable "steps" like for long actions as described above. For example, the statement that the Warden carries a particular bag for a period of time may be represented by using an attributes `carries` for the entity `TheWarden` where the value is an entity such as `TheBag`, or a set of entities if the Warden can carry more than one thing. (Attributes are discussed in Chapter 6). There will then be short actions for representing that the Warden picks up the bag and puts down the bag, that is, for the beginning and ending of the natural-language action of the Warden carrying the bag.

5.4.2 Types of Desires

The concept of a *desire* is fundamental in the BDI Behavior Model. Desires can be thought of as policies: they are continously monitored while the agent operates, and instances of these desires guide the selection of intentions and goals. Desires in the PAE are defined in terms of the values of features: they specify which feature values are desirable and which are non-desirable. The persistence or non-persistence of features influences how desires relate to them. The following cases are of interest:

- *Feel good momentarily:* the agent has a desire that a particular, transient feature value or transient combination of feature values shall occur.
- *Feel bad momentarily:* the agent has a desire that a particular, transient feature value or transient combination of feature values shall not occur.
- *Feel good persistently:* the agent has a desire to establish and maintain a particular, persistent feature value, or a stable value of a transient feature.
- *Feel bad persistently:* the agent has a desire to avoid, or to minimize the extent of a particular, persistent feature value, or a stable value of a transient feature.

These are purely qualitative considerations. In addition there may be desires to maximize or to minimize the values of certain numerical-valued features.

These distinctions determine how options (in the sense of the BDI script) are generated. There are six possible cases:

- *Detect transient opportunity:* React to observations that suggest that using some short action the agent will be able to achieve an instance of *Feel good momentarily.*
- *Detect transient problem:* React to observations suggesting that without additional action by the agent there will be some forthcoming instance of *Feel bad momentarily.* (Notice that if the agent does not do any prediction of the near future so that the feel bad momentarily has already occurred, then there is nothing the agent can do about it).
- *Detect feel good persistently:* React to observations that suggest that using some short action the agent will be able to achieve an instance of *Feel good persistently.*
- *Retain feel good persistently:* React to observations that suggest that without additional action the current instance of feel good persistently is going to end. (This may require prediction).
- *Detect feel bad persistently:* React to observations that suggest that without additional action there will be an instance of feel bad persistently. (This case also requires prediction).
- *Discontinue feel bad persistently:* React to observations that suggest that using some short action the agent will be able to cause a current instance of *Feel bad persistently* to end.

In addition there are quantitative counterparts of some of these:

- *Maximize feel good persistently:* During a 'feel good' period of time, perform additional action so as to increase the value of the attributes representing the comfort level.
- *Minimize feel bad persistently:* During a 'feel bad' period, perform additional action so as to decrease the value of the attributes representing the level of discomfort, even if it is not possible to reduce it to zero.

5.4.3 Behavior Rules

These functionalities are realized in the PAE using *behavior rules* which may be of two kinds: *desirability rules* and *opportunity rules.* Each rule is associated with a *triggering condition,* an *applicability condition,* a *method,* and a *purpose condition.* However, the triggering condition and the purpose condition may sometimes be identical. Each use of any of these rules proceeds through the following steps, in principle: it is detected (by some higher-level routine) that the triggering condition is satisfied. If so, it is identified as an option, in the sense of the BDI top-level loop, and it is forwarded to the `consider-and-select` operation. There its applicability condition is evaluated, and if it is satisfied then the option may be selected, leading to the method being executed. Afterwards it is verified whether the purpose condition is satisfied; if it is then the purpose of using the rule has been achieved.

Therefore, it is the `option-generator` in the BDI top-level loop that drives this process. After it has emptied the event queue, it makes a loop over all

the behavior rules, evaluates their triggering conditions, and returns those where the triggering condition is satisfied.

Each cycle in the top-level loop performs one step of the current long action, checks the behavior rules as just described, and according to the outcome of these rules it may perform one or more short actions before the next step of the current long action. It is in this way that the cognitive agent may, for example, perform services to animals or to itself between successive steps of a promenade long action.

For example, if the agent being thirsty is a persistent "feel bad" condition, the desirable condition may be written as an expression that is true if the agent is not thirsty, and false if it is thirsty. If the desirable condition becomes false during the simulation of an episode then the agent shall seek a corrective action, such as drinking a glass of water.

There are two other negative functionalities, namely *detect transient problem* and *detect feel bad persistently.* These can also be handled using desirable conditions that are used to indicate a problem that needs to be acted on, but in these cases the desirable conditions must be applied to a *predicted future situation.* If the momentary "feel bad" is already occurring then there is no reason to do anything, since it is transient anyway, and if it is persistent and has already started then the need is to discontinue it since it is too late to prevent it from starting.

The case of *detect transient opportunity* is the simplest one of the positive functionalities. A very general way of implementing it would be to always (i.e. at each timepoint, or always between two successive steps of a long action) consider all possible short sequences of short actions, to predict their effects, and to check whether any of them lead to achieving an instance of *feel good momentarily.* This is however not a realistic method, and it is more reasonable to use *opportunity conditions* that indicate that a transient opportunity is within reach. For example, there could be an opportunity condition that evaluates to true if the agent finds itself beside an icecream stand, and that is associated with a method for obtaining an icecream, such as purchasing one.

Opportunity conditions are similar to desire conditions but there are two differences. First, desire conditions trigger an action if they evaluate to false whereas with the natural way of writing opportunity conditions they will trigger an action when they evaluate to true. However for uniformity of processing it is convenient to reverse the sign for opportunity conditions, so that they evaluate to true when there is an absence of opportunity, and to false when there is an opportunity.

Secondly, when a desire condition evaluates to false then the associated methods are methods that will make the same condition come true again, but the methods for opportunity conditions are designed so as to make some other condition come true, namely, an instance of a desire. For example, the persistent condition of being beside an icecream stand is not a goal in itself, but it may bring to mind a method that has the enjoyable although transient effect of eating an icecream bar. A general design that covers both cases is therefore one where a *behavior rule* has three components: a *rule condition,* a method or a set of methods, and a *purpose condition* that the method is supposed to achieve. The first and the third component are equal for *desirability rules* and different for *opportunity rules.* It remains to

consider the two other kinds of positive functionalities. The case of *detect feel good persistently* can be treated in the same way as *detect transient opportunity,* with or without prediction of future states. Finally, the case of *retain feel good persistently* is concerned with identifying situations where an ongoing feel-good state will be discontinued unless an action is taken. Here again there should be a behavior rule where the first and the third condition is equal: if this condition is false in a predicted future state then corrective action is needed in order to make sure that it actually retains the value true.

The quantitative-oriented behavior rules for *maximize feel good* and *minimize feel bad* can likewise be represented using these three components, and also in this case the purpose condition is different from the rule condition, since it characterizes the expected level of comfort or discomfort as a result of applying the method.

Sometimes, but not always one can use the same method for several of the cases described above. For example, becoming wet as a result of being out in the rain is a continous-valued discomfort, and opening an umbrella is useful both for avoiding becoming wet, and for not becoming more wet when one has already become somewhat wet. As a contrary example, however, the methods for avoiding being stung by poisonous jellyfish are different from the methods for reducing the pain when one has already been stung.

5.4.4 Applicability Tests and Choice of Method

Both kinds of behavior rules specify a rule condition that is evaluated regularly, and one or more methods that it may be appropriate to apply if the rule condition is violated. Usually, each method consists of both an applicability condition and a sequence of actions, and the applicability condition must be satisfied before one can consider using the method. The applicability condition is similar to the preconditions of ordinary actions. If the behavior rule contains several methods then it may be that only some of them pass their applicability tests.

There are also other ways of defining the choice of method, for example by merely specifying the purpose condition explicitly and leaving it as a planning task to find a sequence of actions that will achieve the purpose condition in a particular situation.

If several methods pass their applicability tests then the system must choose which of them to use. Simple choice criteria include making a random choice, or associating a goodness value with each method and picking the method with the best goodness value among those that pass their applicability tests.

A more refined way of making the choice is to predict the result of performing each method separately, and then evaluating the outcomes so as to pick the one with the preferred result. This can be done by assigning a numerical merit to each of them, using a combination of merit components that capture different preferences by the agent. The system's body of 'desires' shall be used as a basis for assigning the merit numbers, and for quantitatively expressed desires the contribution to the overall score may be obtained from the level of comfort or discomfort in the attributes that are used.

Since both transient and persistent feature-values must be accounted for, it is necessary to evaluate the successive states in a predicted sequence of future states, and not merely the last one of them.

Another possibility is to use a *comparison predicate* that will indicate which of two alternatives one shall prefer, using other methods than assigning a numerical score to each of them. A comparison predicate may for example be defined using a decision tree.

5.4.5 Follow-up of Behavior Rules

Once a method has been selected and applied, including the "method" of doing nothing as one of the alternatives, it is also important to perform *follow-up,* that is, to check whether the expected outcome has in fact been obtained. This follow-up operation uses the purpose component of the behavior rule. For simple desirability rules the follow-up consists of verifying that the intended feel-good effect has been achieved; for opportunity rules it consists of verifying whether the target opportunity did in fact materialize.

The outcome of the follow-up can be used for both short-term and long-term purposes by the system. The short-term use is to control whether to try again or to resign. If the desired feel-good condition is not achieved then the system may decide to try again using the same method, or using another method, but it may also decide that it is not worth trying and that it will simply accept this particular not-feel-good situation until it encounters an opportunity for discontinuing it, that is, a method for *discontinue feel bad persistently* whose applicability condition is satisfied.

The potential long-term use of follow-up is to modify the entire structure of behavior rules and method evaluation rules. Techniques for doing this belong to the area of *case-based reasoning* and are beyond the scope of the present note.

Chapter 6

Software Platforms and Architectures

This chapter continues the topic of software technology and software architecture for cognitive systems, in addition to the introduction in Chapter 2, Section 2.4.

6.1 Autonomy in a Software Platform

Conventional software technology is partitioned into several types of software artifacts: operating systems, programming languages and systems, database systems, network communication software, various kinds of Internet-level servers including mail servers and web servers, local servers such as print servers, and so forth. All of these are available resources as one begins to implement, for example, the architecture for a cognitive agent. We shall refer to them collectively as the *software environment* for the cognitive agent.

However, in the course of designing and implementing that cognitive architecture, one is likely to encounter situations where the facilities that are needed for the cognitive agent appear to be generalized versions of facilities that already exist in the software environment. The representation of the internal state, and in particular the beliefbase is a kind of database, but with additional requirements; the management of tasks in a Hierarchical Task Network is like the management of processes in the operating system, but with additional requirements, and so forth.

It is therefore natural to consider whether the architecture for the cognitive agent should always make use of the facilities in the software environment, or whether it should renounce from them, wholly or partly, and start from a lower level when building the facilities that it needs for itself. If taken to the extreme this would mean starting from the bare hardware (or even building a dedicated hardware) and doing everything from scratch. This extreme position is not very realistic today, but it is certainly possible to only keep certain lower levels of operating systems and of software for communication and interaction, and to build a new platform from there.

I shall use the term *software architecture* (for a cognitive agent) as a name for the software conventions and facilities that are designed specifically for the needs of the cognitive agent, but which belong conceptually to the topic of software technology and not to the topic of cognitive structures and processes. The term is intentionally vague: in simple cases it may refer to just a loose collection of conventions for how to use the existing software environment, and in other cases it may refer to a coherently designed and implemented software system that is intended to serve as a platform for a cognitive architecture of the kinds that were described in Chapter 4. Such a software system will be referred to as a *software platform* for cognitive systems, and its design is then exactly the software architecture, to be distinguished from the higher-level cognitive architecture that has been our topic before. The inclusion of a particular facility in a software platform, as an alternative to facilities in the software environment at hand, will be referred to as an *autonomous* implementation of that facility.

Today there does not exist any such widely used software platform, except to the extent that a programming language and its software environment can be considered as one. However, implemented cognitive systems often contain aspects of software architecture within them, and there exist a number of widely used software tools, and their associated languages, that have been designed for inclusion in cognitive systems. Knowledge of such software tools is therefore important, regardless of whether one wishes to include them in the implementation of the cognitive architecture, or if one sets out to create a more independent software platform.

The possibility of incorporating major aspects of the conventional software environment into a dedicated platform where they can be integrated and generalized, suggests an intriguing question with respect to the future development of that software environment: *is really the contemporary partitioning of the environment into operating system, database system, and all the other parts the best one?* For example, why should the shell command language be unrelated to the programming language, and why should the operating system's directory structure for files be separate from the database, and why should the embedded programming language in web pages (such as Javascript) be different from the server-side programming language (such as Python), and why should either of those be separate from the database query language? This list of questions can easily be continued.

After all, the basic divisions in the contemporary software environment structure is fifty years old, and the initial structure has been amended many times since then. This alone would be a reason to consider re-engineering the entire structure. In such a case, it could be that a systematically designed platform for cognitive agents might be a good design for a general-purpose software environment as well.

6.2 Programming Language

The choice of programming language is important in the consideration of software tools and of the possible design of a software platform. Cognitive systems impose a number of requirements and desiderata on the programming language being used; the most important of these were already discussed in Chapter 2, in particular:

- Support for the construct of symbols, at least in the sense of normalized strings.
- Availability of a syntactic style and software support for it, for example S-expressions in Lisp.
- Attached functions and procedures, i.e. the possibility of assigning functions and procedures as attribute values of symbols, and of looking them up dynamically for the purpose of executing them.

The second item is important in particular because it provides the basis for using the host language as a *high-level implementation language* for a number of specialized languages that are needed in the overall design of the cognitive architecture.

The preferred choice for many researchers in Artificial Intelligence is the Lisp programming language. This language was first defined by John McCarthy in 1958 and was specifically intended as a programming language for Artificial Intelligence research. [1] During the first two decades of its existence it was widely accepted as the foremost software tool in the field and an important aspect of the field's research. Consequently there were frequent research projects and research articles that addressed the further development of this language. However, since around 1980 the programming language issue has largely been kept in the background: other and more widely used languages are also being used, and the choice of programming language is often considered as a non-issue.

One result of this is that the programming technology that is actually being used in the development of cognitive A.I. systems is not very well documented, neither in research articles nor in textbooks. There does exist a software development style that centers around the use of Lisp and languages with similar properties that have been developed more recently, such as Python, and this software style also includes many other aspects (and more significant aspects) besides the immediate language questions. The present chapter is an attempt to document relevant aspects of this fairly tacit know-how.

Note for the review version of the present book: One consequence of the scarcity of written sources for this topic is that I must partly rely on my own understanding of, and my habits in A.I.-related programming, and these do not necessarily agree with what others would say. Feedback on the present, tentative contents of this chapter is therefore eagerly sought.

In addition to Lisp, descendants of Lisp, and Python, there is also another widely used programming language that is specific to Artificial Intelligence and related topics, namely Prolog. For Prolog the situation is different from the one for Lisp in the sense that Prolog does have (or had for a longer time, at least) research projects and research publications that address the development of the language as such. Prolog has some capabilities that Lisp in itself does not have, in particular the inference capability, but it lacks the availability of a syntactic style and its use for writing language interpreters.

[1] However, the needs for Artificial Intelligence were not the only consideration for John McCarthy at the time. He was a member of the design committee for algol 60 but, according to his own account, he was disappointed because the committee did not wish to accept a number of proposals that he made and that he considered important for programming in general. Consequently he designed Lisp as a first step towards realizing those proposals.

6.3 Execution of Actions

The autonomous execution of actions is the characteristic property of a cognitive agent. It is also an aspect of the overall system for the cognitive agent where high-level, cognitive capabilities can be "projected" down, i.e. they can be made use of in what is usually thought of as entirely mundane aspects of a software system.

One particular case, among several others, is when the human user of the agent requests a particular action to be performed. This defines a meeting-point between the cognitive architecture and the software architecture. For the software architecture point of view, it is natural to provide the agent with a *command loop* whereby the user can input a command consisting of a command verb together with its arguments, and where the system executes the command and is then ready for the next user command.

The question is now how user commands in this sense shall be related to actions in the general sense of the cognitive architecture. It may be natural to consider the user input command as an action among all other actions in the agent, and in particular to consider each user command, possibly with an enclosing wrapper, as an event in the sense of the cognitive architecture. If this solution is adopted then it means that commands from the user command loop can be provided with the same facilities as other actions, for example, precondition checking and appropriate measures if the preconditions for the action are not satisfied.

Thus far it is easy. The next step is what happens if what is originally a user command is forwarded to another agent, in a network of cooperating cognitive agents, and if problems occur during the execution in that other agent.

Notice, in this context, that the commands that are input from the user command loop need not always be commands that request a change in the internal state of the cognitive agent. In a robotic situation they may be commands that request a movement or other physical action, and requests such as obtaining printouts of a file or set of files is an intermediate type of task.

It is easy to see how these possibilities can be extended, for example in the direction of getting the system to learn from experience and to handle given tasks in a goal-directed manner and using the pattern from previously performed actions. Diagnostic reasoning may come into play. In summary, high-level cognitive capabilities can be applied to apparently low-level services in the software.

6.4 Organization of Knowledge

Chapter 4 described representation of domain models using logic formulas. The most obvious way of expressing such domain models in text files, which will be necessary for working with domain models consisting of large numbers of formulas, is to list them one after the other in some suitable order.

Another possible way of organizing similar information is in terms of *entities*

and their *attributes.* In this case, an *entity description* consists of a symbol (representing the entity itself) and a set of attribute-value pairs, each of which consists of an attribute which is likewise a symbol, and an arbitrary KR-expression that is the value for the given attribute of the given entity. The entire collection of information in a file will then be a set of entity descriptions, called an *entityfile.*

The representations as a propositional file and as an entityfile are of course equivalent and one can easily be expressed in terms of the other. In the very common case where the propositions are ground literals, i.e. each proposition is a predicate (or the negation of a predicate) with its arguments, and the arguments do not contain any variables, it is natural to collect all literals with the same first argument and to combine them into one entity description. One attribute may contain all the literals with the given first argument, or one may choose to have one attribute for each of the predicate symbols being used, for example.

The choice of alternative in these respects may have an impact in two ways: with respect to appearance and ease of reading in the file in question, and with respect to what kinds of computations and algorithms are being encouraged.

The issue of ease of reading is of course only relevant if the files are going to be read by people in the first place, but this is often the case. As the following small examples show, it is boring and inconvenient to read a number of literals that repeat the same argument. Compare

```
[population sweden 9000000]
[capital sweden stockholm]
[flag-color sweden blue]
[flag-color sweden yellow]
[official-language sweden swedish]
```

and so forth, with

```
sweden
  [: population 9000000]
  [: capital stockholm]
  [: flag-colors {blue yellow}]
  [: official-language swedish]
```

In small examples one can live with either representation, but if one is going to spend some time working with factfiles of this kind one will certainly prefer a presentation of the latter kind.

The second issue concerns what type of computation is being encouraged. The propositional representation suggests the use of logical deduction and other logic-based methods for using the information in question, whereas the representation as entity descriptions suggests the use of functional style, beginning with the use of a function `get` where `(get sweden capital)` would evaluate to `stockholm` in the example.

The choice between these alternatives, or other ones in the same space, is therefore not as innocuous as it may seem, since it tends to define the initial direction for work that may end up being fairly large. At the same time one does not necessarily have to exclude one alternative while choosing the other, and a balanced use of both these perspectives as well as more of them may often be the best approach.

Will add a few lines about OWL here, the main issue being the kind of reification that you are forced into if attribute values are not allowed to be composite expressions.

6.5 Mobility and Persistence

The topic of *persistence* for cognitive agents was addressed in Chapter 2 in the context of software individuals. Persistence is important for two reasons, namely, learning and cooperation. Learning in a cognitive agent is only meaningful if the agent has a duration of life that can be counted in months, and even in years, which means that it must be designed in such a way that its internal state, including what it has learnt, is preserved even if the computer where it is running is stopped temporarily, and if it malfunctions or is destroyed.

Persistence is therefore related to mobility, in the sense of the ability for a cognitive agent to be moved to another host, since this is necessary if the host at hand ceases to function. However, there may also be other, practical reasons for agent mobility. Some of the software agents that I myself use regularly are located on USB sticks so that they can be used both in a laptop while travelling, and in a better equipped desktop when the opportunity arises.

The persistent representation of a cognitive agent can be obtained in a variety of ways. One approach was described in the context of software individuals in Chapter 2, where the agent has two manifestations. The *dynamic manifestation* is a computational session with the agent; it is in this manifestation that it can interact with its environment and with other agents. The *persistent manifestation* is as a set of files in the computer's permanent memory (hard disk, for example, or USB stick), containing a textual representation of all the data in the dynamic manifestation that need to be preserved. Computational sessions start by loading the information from the persistent manifestation, and are arranged so that the session updates the appropriate files following any essebtial changes in its internal state.

An entirely different approach is to use *memory dumps,* where the entire memory state of the executing session with the agent is written to permanent memory at regular intervals. This may have the advantage of starting the agent more rapidly; it also allows greater flexibility in the choice of data-structures within the agent. For example, cyclic structures are notoriously difficult to convert to textual form in a reasonable and readable way.

On balance, the present author strongly favors the first approach, where the persistent manifestation is organized as a collection of text files containing expressions in the agent's representation language. Startup time has never been an issue anyway in the cases I have worked with; data structures that express logical formulas like in KR expressions are inherently tree-formed anyway, and auxiliary structures that are needed for performance reasons can easily be reconstructed when a new session is started. The big advantage with organizing the persistent manifestation as a set of textual, human readable files is for development and maintenance: the source information that the developer provides to the agent is in the same format as the information whereby the agent preserves itself.

The use of a collection of text files, and even the representation of the entire cognitive agent as a directory with subdirectories in the host computer's file system, may of course also be questioned. Would it not be better to store the persistent representation of the agent in a database system of some kind, which could then be used during the dynamic sessions as well as between them? This is a valid question. One argument in favor of the directory-and-file approach is that it can be put to work with minimal effort, besides what is needed anyway for parsing and serializing the chosen representation language, whereas the creation of a database that satisfies the needed requirements (as well as the identification of those requirements) would be an effort in itself. In a concrete development situation is could therefore be an undesirable delay for the entire project. On the other hand, if an adequate database system is already available then this is a serious alternative.

6.6 Knowledge Acquisition

This section needs to be written. It should address the downloading of information from the web, including parsing the XML and, if necessary, the HTML code, as well as extracting information from these sources. It should also address the use of existing large knowledgebases, ontologies and factbases, including Wikipedia, DBpedia, Cyc, Freebase, SUMO, others.

6.7 Lower Level Computation

The cognitive architectures in Chapter 4 assume that inputs arrive in the form of discrete messages, called events, and they allow that a set of events – none, one, or more – may be received in each main cycle. There must therefore always be some underlying computational processes that produce those events. This includes *sensoric software* in robotic systems that receive and process incoming sensor data, *communication software* that receives information from other cognitive agents or other computational processes in general, and *human dialog software* of various kinds – for graphic input, speech input, and so forth. We shall address these in turn.

The description of the cognitive architectures did not say very much about the question of output from the cognitive agent; it was tacitly assumed that this can be an issue for the implementation of each particular elementary action. In the present section we shall discuss the input and the output issues together for each of the three cases.

6.7.1 Sensorimotoric Software

Robotic systems that operate in the real world make use of a considerable number of sensoric and motoric devices, in most cases. The implementation of their sensorimotoric information processing is usually based on a number of communicating and specialized processes, often using a network of specialized processors.

Among the available approaches for organizing this software, two should be mentioned in particular. One is the use of CORBA [2] (Common Object Request Broker Architecture) which provides a reliable and high-performance communication method and which is an industry standard.

Another approach is represented by the Robotic Operating System (ROS) [3] which is not an operating system in the standard sense, but Linux-based software that "provides libraries and tools to help software developers create robot applications. It provides hardware abstraction, device drivers, libraries, visualizers, message-passing, package management, and more" (quoted from the ROS website).

Although actual robot systems make use of a distributed architecture of this kind for sensorimotoric purposes, there is a frequently used abstraction called the *three-level architecture* which stipulates that the entire software for the robot may be defined on three levels that operate with different frequency i.e. duration of main cycle. The cognitive system that is the topic of the present book is seen as the uppermost level and it communicates with the middle level in terms of discrete messages, or what we here call events. The lowest level contains control algorithms, so many of its inputs and outputs are sampled approximations of continuous signals. The middle level contains an intrinsically discrete state and is defined in terms of state transitions that are triggered by incoming messages from both the lowest and the upper level of the structure. The essential difference between the middle and the upper level, in this design, is that the upper level contains a notion of history and a notion of goals, whereas the middle level only contains a current state and it has no representation of its past or its future.

Literal implementations of the three-level architecture are not realistic for real-world robots, but this design may anyway be useful as an abstraction of the real system structure, for example, to the extent that the cognitive level has a need to model the lower software levels of the robot. It may also be that a direct implementation of the three-level design can be of use for simulation purposes and in computer games.

6.7.2 Communication Software

Lower levels of communication software are provided by standard operating systems. From the point of view of cognitive agents it is however important to have a well defined software level for agent-to-agent communication which is compatible with other parts of the agent's architecture. In particular, it is required that the *message content* shall use the same notation as is used for the representation of domain models within the agent. In many cases such a communication mechanism can also be used for communication between a cognitive agent and a computational process that is not a cognitive agent, namely, if the latter is programmed so that it accepts the agent-to-agent protocol.

As already mentioned in Chapter 2, the earliest widely adopted definition for agent-to-agent communication was KQML which stands for "Knowledge Query and Manipulation Language." In this case a 'query' is to be interpreted as an information request from one agent to another, and not as e.g.

[2] `http://www.omg.org/`
[3] `http://www.ros.org/wiki/`

a database query. KQML uses the syntactic style of S-expressions and is organized as a set of *communicative acts;* this term is intended as a generalization of the term "speech acts" since software agents do not use speech for communication between them. These communicative acts are formally defined using their satisfaction conditions, along the lines of Searle's speech act theory. [4]

The KQML language was the basis for a standardization effort in the FIPA (The Foundation for Intelligent Physical Agents, which existed from 1996 to 2005) which was a body for developing and proposing computer software standards for heterogeneous and interacting agents and agent-based systems. The FIPA "standard" called the Agent Communication Language (FIPA-ACL) is the most widely used one of FIPA's proposed standards. After the dissolution of FIPA its work continues in an IEEE standards committee.

The specification of ACL consists of a set of communicative acts and the description of their *pragmatics,* that is, the intended effects on the internal state of the sender and receiver agents. As an adaptation to the technical context, communicative acts are also called *message types.* The pragmatics of a message type is similar to the satisfaction condition but it is more limited since it does not take changes in the physical world into account. For example, the pragmatic effect of a command to play a particular tune would be that the receiving agent has as a goal to play that tune, not that the tune is actually played.

The pragmatics of a communicative act is described using both a narrative form and a formal semantics in a specification language that is based on modal logic.

The FIPA-ACL standard has been implemented i.a. as the Java Agent DEvelopment Framework (JADE). The *Interagent Communication Language* (ICL) is the language used by the *Open Agent Architecture* (OAA) which is based on the Prolog language; it is one of the alternatives to KQML and ACL.

Additional information about these communication methods can be found in the Annex of this chapter.

Besides the software for reliable exchange of messages, there is also a need for a *registration facility* for cognitive agents, especially if cooperating agents can be located at different locations within the Internet, and if they can move or be moved from one location to another. This is related to the notion of software individuals that was introduced in Chapter 2: if the overall system is organized in such a way that cognitive agents have an identity that remains fixed even if the agent is moved, and if each agent has a name and there is clear notion that the *same agent* can not exist in several copies, only then is there a firm basis for message-passing and cooperation between agents, that is, between intelligent software individuals.

6.7.3 Human Dialog Software

Communication between a human user and a cognitive agent can take many forms. The most obvious ones are using a graphical interface, us-

[4] Speech act theory will be described in a later chapter.

ing speech, and using written natural language, besides of course using a simple command-line interface. Additonal methods include the use of pointing and other gestures, which require that the computer system can observe the behavior of the user, for example using a video camera or using sensors on the user's body. Gestures by the computer system require the use of a robotic device of some kind.

The use of natural-language communication in the form of speech or as written natural language is a very large topic and I hope to be able to include an account of it in a later volume of this work. Interaction using gestures is not on the horizon for the present book series.

6.8 Software Maintenance and Documentation

The use of version management systems such as CVS (the Concurrent Versions System) [5] is a standard technique in large and distributed software development projects, and it would be plausible to use such a system in any implementation of a cognitive architecture as well. But on the other hand, the argument about the advantages of autonomous facilities may be made here also: Provided that we have a software and cognitive architecture for a cognitive system whereby agents can communicate with each other on the cognitive level, would it not be better to use this facility for version management and for the distribution of software updates as well, instead of relying on an external facility?

Like in other cases where an autonomous solution can be considered, there would be a good case for doing so if the facility could be implemented with moderate effort using other resources that exist already in the platform, and if it could find other uses besides for the management of software updates, and if it could be made more powerful due to the use of, and synergies with other facilities in the agent.

A similar issue arises with respect to software documentation in the sense of technical reports describing various aspects of the software. The software environment provides text editors and formatting tools, and one may easily do the documentation work using these. However, it is also possible to consider incorporating some of these into the software platform.

The design in the Leonardo system may serve as an example here. Leonardo uses a text markup language that uses its syntactic style (KR-expressions of the kind defined in Chapter 3), together with translators from this markup language (TSL) to HTML, Latex and plain text. This is used in tools for the preparation of pdf documents and of webpages, but it is also sometimes used within the system for generating output lines in the user's command-line dialog. The implementation of TSL is small and simple, and its integration with CEL provides macros, loops and access to the Leonardo knowledgebase.

In one use of this facility, the textual descriptions of predicates and command verbs are stored as properties of their respective symbols, along with the type and attribute information and the procedural definitions for these symbols, and both text and structural information can then be inserted into documentation while it is being formatted. For example, the definitions of predicates in Section 3.2 of this document has been obtained in this way.

[5] http://cvs.nongnu.org/

This design makes it possible to keep documentation fragments together with the program parts that they document, which is in the interests of keeping the documentation consistent with the actual software during its evolution.

Chapter 7

Multiple Models and Representations

7.1 The Need for Multiple Representations

In Chapter 4 we introduced the notions of models and their representations as follows: a model of an application consists of a selection of entities in that application, together with a selection of relationships between those entities, or relationships between entities and scalars such as numbers or strings. Therefore, the model is considered to be *part of* the world in the application, which is also called the *domain* in question. Once a model has been identified, one can construct a *representation* for the model, usually as a collection of formulas of some kinds.

Both the *modelling step* and the *representation step* may be simple and straightforward in some cases. For example, in a family relationship domain model one may select which persons are going to be included, one may acquire the information about their family relationships, at this point one has already identified the model, and then one may represent the model using literals such as

```
[married-to bertil tilda]
[child-of gunilla bertil]
[is-female gunilla]
```

However, in many other situations things are not so simple. Suppose, for example, that instead of the binary family relationships one was interested in using relations such as "is a friend of," "is an enemy of," or "trusts." Or, suppose the entities in the model are cities rather than persons and one is interested in relations such as "is near" and "has good connections with." In such cases it is unlikely that one can find an objective, ie. observer-independent, way of acquiring the information for the model. We shall refer to this kind of relationships as *imprecise relationships.*

One may argue of course that people already make use of imprecise relationships, and different people may have different opinions about the qualities of friendships and of travel connections, and that therefore it is not a problem if the same thing occurs in software individuals. However, the problem is that although we as people make some use of imprecise relationships, we do

not use them blindly, and each time we deliberate about something where considerations such as these come into play, we tend to bring in additional information in order to obtain the best possible basis for our decisions or conclusions.

Notice however that the issue here is not whether one should assign a numerical value to the relationships, for example "Gustaf is 76 percent good friends with Anders." The issue is instead that imprecise relationships are umbrella terms that include a number of different aspects, and in many cases of deliberation it is important to go beyond the umbrella term and deliberate on those aspects of it that are relevant for the topic at hand.

This means in no way that the imprecise relationship ie. the umbrella concept is useless and should be ignored. It may be used in a first stage of deliberation, in order to formulate an issue and ask oneself what questions need to be asked. It may also be used for summarizing the conclusion of the deliberation, both for one's own sake and for communication with others.

Our conclusion must be instead that there is a need for *multiple representations* of the same domain – some that are more general and sweeping, and some that are more detailed and specific. A cognitive system should be able to use multiple representations, and it should be able to switch between using one and using the other in the course of deliberation.

In fact, in a longer perspective one should also require that a cognitive system should be able to *design its own representations* in ways that are tailored to the needs of various classes of problems that it encounters.

The reason why we specify this as a need for multiple representations, and not as a need for multiple models, is that deliberations and other computational processes in the computer are performed on representations and not on models, so the need for multiple representations is the primary one. Moreover, it is possible to take the position that there shall only be one model, but several representations of this model, so the use of multiple models (for example, one for each representation) is a possibility but not a logical necessity in view of what has been said.

The need for multiple representations has also been observed in other branches of Artificial Intelligence, besides in the study of modelling principles. In particular, in the study of problem-solving methods it has been observed that one major problem-solving method in humans is to convert a stated problem to another representation, to solve the problem in the new representation, and to covert the solution back.

At the same time it must be admitted that this insight has not led to as many concrete results as the topic would merit. Basic Artificial Intelligence techniques, such as those that are described in Volume II of the present textbook series, are consistently based on the use of a single representation, and the same applies for major cognitive architectures. In view of the topic's importance we shall discuss it in the present chapter, but only in order to comment on how it relates to the system architecture issues.

7.2 Direct and Secondary Representations

Before one begins using multiple representations in a cognitive system, it is worthwhile to have a framework for understanding how different representations of the same reality may be related to each other. The following conceptual framework is good as a first step: we allow for having one model of the application, one *direct representation* and one *secondary representation.* The model at hand will be considered as an *overdetailed model* relative to the secondary representation.

For example, consider a model for cities in a country that is organized as follows: it identifies which of the actual cities are to be considered as entities for the purpose of the model [1] and for each city-entity it specifies the geographical location of the center point of the city in terms of longitude, latitude, and altitude above sea level. A direct representation of this model may consist of propositions such as

```
[geoposition stockholm +16.2 +59.3 +10]
```

where the last three arguments indicate longitude, latitude, and altitude in that order, and in degrees east, degrees north, and meters, respectively. The secondary representation might instead consist of propositions such as

```
[near stockholm soedertaelje]
[-near stockholm malmoe]
[north-of luleaa stockholm]
```

The first representation has its name since it corresponds in a very direct way to the model itself. The second representation is a reduced one in the sense that some information in the model has been lost there. In order to evaluate whether the literal

```
[near stockholm soedertaelje]
```

is true in the model or not, one has to use the coordinates of the two cities in order to calculate their distance, and then make a judgement whether the calculated distance is so short that the conditions for the predicate `near` shall be considered to be satisfied. Since many different assignments of geographical coordinates may produce the same truth-values in the secondary representation, there has been a loss of information.

However, although some information has been lost, one may argue that some other information has been added. If a cognitive agent were asked to use the direct representation for determining whether Stockholm is near Södertälje, then it should ask with a counterquestion of "How do you define near?" unless it had alredy been told about this.

Moreover, *near* is an imprecise relationship in the sense defined above, and whether two cities are considered to be near one another may depend on the context, for example which means of transportation is being contemplated, and it may also depend on the habits and capacities of the agent using the term.

[1] Notice that in reality there may sometimes be different definitions about what is a city and what is not, and whether a particular urban area shall be considered as one city or as several.

Consider therefore the following two perspectives on the secondary representation in this case.

(1) The representation consists only of literals such as those listed above, but it does not contain a definition of its imprecise relationships in terms of underlying information.

(2) Besides literals such as those listed above, the representation also contains definitions of the imprecise relationships in terms of a combination of city coordinate information and other information that is also available to the cognitive agent.

We shall refer to these as the *implicit* and the *explicit* approach to handling imprecise relationships. This distinction applies of course not merely to the case of handling geographical locations and their distances, but in fact to any kind of situation where imprecise relationships are being used.

The following is how one may imagine a robotic cognitive agent that uses the implicit approach. Its perception system obtains a direct representation of the agent's model of its environment, for example, estimates of the position, orientation and size of objects in the environment. Thereafter, there is an interpretation process that obtains the secondary representation, which in turn is used for the purpose of deliberation.

The most straightforward way of obtaining the secondary representation is to attach an *interpretation function* [2] to each predicate symbol. For each ground literal [3] using this predicate, the interpretation function may be invoked in order to calculate whether the literal is true or false in the model at hand, by referring to the direct representation of the model.

The explicit approach requires that the relationship between the model and the secondary representation is represented explicitly. The most straightforward way of doing this is to use *interpretation axioms* that are formed as implications where predicates in the direct representation occur in the antecedent, and predicates in the secondary representation in the consequent. However, other methods may also be considered, for example the use of a decision tree for defining a predicate.

The important thing in this distinction is that the explicit approach requires that the predicates in the secondary representation shall have their definitions represented explicitly, so that they are available for inspection, for other kinds of computation and not merely for interpretation, and in particular so that these definitions can be changed by learning. In the implicit approach, the rules for obtaining the truth-values of literals in the secondary representation are represented as program stubs that are in not generally accessible to computation and learning.

This means also that the direct representation of the model in the implicit approach may use arbitrary datastructures as permitted by the host programming language. In the explicit approach the direct representation must be expressed in terms of logic formulas or other expressions that can be manipulated on the knowledge-representation level.

Let us also relate the explicit approach to the notion of three *levels of*

[2] Compare what was said in Section 6.2 about the importance of using a host programming language that permits attached functions.

[3] Ie. a literal not containing any variables.

representation, that was introduced in Chapter 4, Section 4.2, namely the framework level, the ontological level, and the observed level. If the representation is chosen as a set of propositions in logic, ie. as a formal theory, then there is correspondingly a framework theory, an ontological theory, and a theory consisting of observations. Normally the latter consists mostly of ground literals, i.e. simple propositions consisting of a predicate and its arguments, or a negated predicate and its arguments. Usually also the ontological theory consists mostly of composite propositions that use quantifiers and logical connectives.

In these terms, the definitions of the predicates in the secondary representation belong to the ontological theory, and as such they can be used for inferring observations in the secondary representation from observations in the direct representation.

7.3 Alternative Representations in Action Monitoring

We shall now discuss one important use of multiple representations, namely, for action monitoring in robotic agents, that is, checking whether the actual effects of an action is consistent with the agent's expectations for the action, and taking appropriate corrective measures if this is not the case. The discussion will be based on an idealized architectural top level for a system that is capable of monitoring the execution of actions,

Effect-level descriptions of actions (cf. Subsection 4.4.1 in Chapter 4) characterize actions in terms of the external state when the action starts and when it ends. Operational-level descriptions also describe intermediate states during the execution of an action that proceeds over several steps of time. Correspondingly, effect-level monitoring of actions checks that the preconditions of an action are satisfied before it starts executing, and checks that the postconditions of the action are satisfied when it has ended, but it does not monitor intermediate states during the action's execution.

The two types of monitoring are not mutually exclusive since one may sometimes consider operational-level monitoring as an add-on that applies in addition to the effect-level monitoring. Operational-level monitoring is essential in many cases, in particular when there are concurrent actions of extended duration that may have interactions affecting their respective outcomes. However, the present chapter will only consider effect-level monitoring since it is the natural first step to address.

The idealized architecture that is used here does not take the possibility of interacting concurrent actions into account. However the approach to validation that uses the architecture does not depend on that restriction.

7.3.1 Elementary Episode Management

For the present purpose we define and use a simple, HTN-style executive for effect-level monitoring. The definition is based on the cognitive agent using a few concurrent *episode representations* each of which consists of a sequence of contexts. Each context describes the state of the environment, or a

possible state of the environment, at a particular timepoint. In particular the agent shall maintain its *observed episode* with a representation of the state of the environment at successive timepoints up to, but not beyond the current time.

In addition the agent shall maintain an *agenda* and a *monitoring episode* for this agenda. The agenda can be simply a sequence of tasks [4] that the agent intends to perform in the next following timesteps.

In each main cycle the agent performs optionally an action (i.e. it performs one action or no action), observes the outcome of the action in its environment which also has the effect of incrementing the current time by one, and adds one more context to its observed episode with the results of the observations in it, so that the latest context in the observed episode is always a representation of the current state of the environment.

There is an obvious question what will happen if the environment is not observable in its entirety; a natural way of handling this is for the agent to extrapolate the contents of the previously current timepoint with the modifications that the agent can infer. This is not a relevant question for the present purpose however.

In addition to maintaining the observed episode, the agent also maintains the monitoring episode in the following way. The monitoring episode shall always start with the observed-episode context for the current time; contexts for earlier timepoints are not retained in the monitoring episode. Instead, the monitoring episode also contains contexts for the *following* timepoints which will occur if and when the current plan is executed. The contents of these contexts are derived using information about the normal effects of the intended actions, i.e. using the action effects laws for these actions.

Moreover, when an action has been performed and a new context is added to the observed episode, the agent also compares the contents of the context for the new timepoint in the observed and the monitoring episode. If these are equal then the action has had the intended effects, and the agent can proceed. If they are not equal then something went wrong with the action, and it is not clear whether it is appropriate to continue with the present plan. In this case the agent shall deliberate and this may result in a change of plan.

The questions of planning and plan revision are left outside the present discussion, but one can merely observe that a plausible way of planning may be to generate a number of alternative plans and to verify them using separate contexts, in a manner similar to how the monitoring context is used. Predictions of the effects of each candidate plan can be used as a basis for choosing which plan to use.

[4] Besides the previously defined distinction between an action and a process, where a process is an instance of an action, there is also a certain need for a distinction between an action and a task. A plan or an agenda consists of a number of tasks, each of which represents an action that the agent will perform if it executes the plan. Tasks are not processes since they need not have been executed, and since the same plan may be executed several times, but it is possible to have a plan that contains several tasks that represent the same action, namely, if the plan involves performing the same action several times. The distinction between action and task is therefore well-defined, but on the language level it is easy to confuse them, and in the present chapter we shall sometimes write 'action' when 'task' would have been more appropriate.

7.3.2 The Control Episode

The basic design in the previous subsection took for granted that the actions are executed using physical effectors that are at the disposal of the cognitive agent, and that the feedback concerning the actual outcome of the action is obtained from physical sensors. It is in the nature of things that the data that are produced by sensors, and that are required by actuators are much more detailed than what is used on the deliberative level. However, in line with the points made in Section 1 of this chapter, there may also be situations where the deliberative process needs to have access to a more direct representation of lower-level data streams.

A principled way of arranging this is to maintain a *control episode* concurrently with the observed episode and the monitoring episode. It is distinguished by using a more direct and more detailed representation than the first two episodes, whereas those two should use the same signature and therefore the same level of detail.

Furthermore, the control episode will only contain two contexts, namely, a context for the 'previous' timepoint and the 'new' one. During each cycle, sensor data are added to the new context, and the old context is used for obtaining default values for these data in case the real data from the sensor are not available. (There are also other uses of the data in the 'old' context but this is outside the present topic).

The same design may be used if the cognitive agent is to operate in simulation mode, where the control episode is used as the object structure for the simulator, so that it should actually be called the *simulation episode.* The simulator shall be able to receive the expression for an action, as decided by the cognitive agent, and to construct the contents of the new context in the simulation episode from the contents of the old one, using a program or script for the effects of the action. When the construction of the new context has been completed, the old context can be discarded, and the cycle is repeated.

7.3.3 Episode-Oriented Operations for Simulation

The implementation of the designs in the previous subsections requires a number of *operations* which are like action verbs except that they are below the cognitive level – they are not part of the action-selection process that is characteristic of a cognitive agent. Our terminology will be that an *instruction* is an expression consisting of an operation and its arguments. Therefore, operations are analogous to action verbs, and instructions are analogous to actions.

Operations and instructions are required both when a cognitive robot operates in its real-world environment, and when it operates in a simulated environment, but of course the choice of operations is likely to differ.

In the case of simulated, sequential execution of actions, it is possible to manage with a very small set of primitive operations, if used together with the general facilities of a language such as CEL. The following assumptions are made for the purpose of the discussion.

- The cognitive agent manages all three episodes itself, including the update of the simulation episode.
- The domain model is such that all actions are performed during a single timestep, and the only changes in the external state are due to the immediate effects of an action. In other words, if the cognitive agent does not perform any action, then there is no change in the environment. Under these conditions it is sufficient to construct a new context in the observed episode when an action is performed, but not otherwise.
- The cognitive agent and the simulation have already been initialized, so that the three episodes are already in place and no further episodes need to be created. The full system will need operations for these purposes as well but they need not be defined here.
- Those entities that occur in the observed and monitoring contexts are also available and can be used in the simulation context. The simulation episode may contain additional entities but then the observed episode will not be able to observe them.
- The actual model and, therefore, its direct representation, consists of a number of entities and a number of attributes and corresponding values for each entity.

Effectuation Mechanism

The foremost primitives for updating the simulation context, i.e. constructing the new context from the current one, are the function `p.get` and the operation `n.put` where `p.get` takes two arguments – an entity and an attribute – and returns the current value of the attribute for the entity in the *previous* simulation context. Similarly, `n.put` takes three argument – an entity, an attribute and an arbitrary KR-expression – and assigns the third argument as the entity/attribute value in the *next* simulation context at hand.

The entire update for the simulation context is organized so that it first sets the new context equal to the previous one, and then invokes a script that is associated with the action verb for the action that is to be performed at this point. This script is defined using the `n.put` operation, and its execution results in the update of the simulation context. It is then up to the designer of the simulation to 'program' the action verbs in accordance with his or her intentions.

Since attribute values can be arbitrary KR-expressions, it becomes possible, if an implicit approach is used for the compact representation of the model, to use a large variety of structures in the simulation episode. There must therefore be a reportoire of functions for composing attribute values and for extracting their components, for example, functions that operate on numbers and on strings. When attribute values are sequences there is a need for functions that obtain elements of a sequence and that put together sequences, and so forth. A reportoire of such functions is defined in the Annex of this chapter, and it will be used in examples here as well as in Volume II.

The use of an explicit approach, on the other hand, leads to much stronger restrictions for the structure of the model and its direct representation.

Perception Mechanism

From a quite abstract point of view we have already defined the essence of the perception mechanism: it shall consist of the computational devices for obtaining the truth-values of literals in the compact representation from the current data in the control episode ie. the simulation episode. However from a computational point of view there are some other aspects to this. In particular, it does not make much sense to recompute all items in the concise representation from scratch in each cycle of the system, in particular if the concise representation has been chosen in such a way that many parts in it are persistent or have stable values (cf. Section 5.4.1). One is therefore interested in techniques whereby one can efficiently identify those changes in the control episode (or the simulation episode) that lead to a *change* in the compact representation, and to pay particular attention to them.

This concern with *identifying change* in the perception mechanism is further motivated by that fact that many of the methods for deliberation about actions and their execution are organized in terms of the changes that arise due to the execution of actions, as well as those changes that arise due to "actions of Nature."

7.4 Conclusions

The topic of multiple models and representations is a complex and difficult one, and in the present chapter we could only scratch the surface of it. One obvious further question concerns the use of several layers of compact representations, or representations that are compact in different ways. Another question concerns the use of quantitative information in the control episode, for example, for the purpose of control algorithms.

In this chapter we have tacitly assumed that the use of imprecise relations is a characteristic aspect of compact representations. This is not a necessary assumption, however, since one may easily find examples of the use of a direct and a compact representation where anyway the predicates in the latter are precise, and are strictly defined in terms of predicates in the direct representation.

We have also suggested that if the explicit approach is used to the relation between the representations in question, then the relation between them can be expressed in terms of axioms in logic. In practice in should expect to find that the logic in question is going to require some nonstandard features, for example the use of nonmonotonic logic in order to specify defaults.

There are also a number of philosophical issues in this context. For simplicity we have defined the model as being *part of* the aspect of the real world that is being modelled, and that it is complemented by one or more *representations of* the model. However, if the model is then defined so as to use metric values for, for example, the size and weight of a physical object, or the geographical coordinates of a city, then one may question whether these attribute-value assignments are really a part of the reality – are they not aspects of our representation of the reality?

This is a reasonable argument, but the answer to it is that it does not matter, and all that is needed is that we recognize that what we have called the

"model" should really be called a "primary representation," but admitting that this primary representation does not have to be expressed using formulas, or diagrams; it may be considered as an abstract structure consisting of "entities" (whose exact character is left undefined) and relationships or other structures over entities.

With all due respect for philosophical issues such as these, there is also a danger of being too much preoccupied with them since they may be issues of form rather than substance, and since they may take attention away from more significant issues in the design of cognitive agents.

Chapter 8

Overview of Artificial Intelligence Research

Different people have different opinions about what are the subareas of Artificial Intelligence, and about which of these subareas are the most important ones or the most basic ones. The following description is believed to be representative of the views in the Cognitive Systems area.

8.1 A list of subareas of A.I.

The following is the list of research areas in the call for papers of the 2011 AAAI conference (Association for the Advancement of Artificial Intelligence).

```
A  Agent-based and multiagent systems
C  Cognitive modeling and human interaction
A  Commonsense reasoning
C  Computer vision
B  Constraint satisfaction, search, and optimization
C  Evolutionary computation
C  Game playing and interactive entertainment
C  Information retrieval, integration, and extraction
B  Knowledge acquisition and ontologies
A  Knowledge representation and reasoning
C  Machine learning and data mining
A  Model-based systems
C  Natural Language Processing
A  Planning and Scheduling
B  Probabilistic Reasoning
C  Robotics
C  Web and information systems
```

The list of topics for the International Joint Conference on Artificial Intelligence (IJCAI) is similar. - In this list each area has been marked with one of the letters A, B, and C, with the following meanings. The A and B groups include work with a similar methodology, based on core computer science (algorithms, etc), the use of formal logic, and applying these tools

to the qualitative modelling of phenomena in the real world and reasoning based on the information in such models, as well as to the design of cognitive agents that can make use of those models and that can perform the said type of reasoning.

The difference between the A and B groups is that the A-listed topics tend to be more basic from a cognitive systems point of view, so that one should study them first before proceeding to the B-listed topic. [1] This borderline is not strict but it can serve as a guide.

The C-listed topics are by and large interdisciplinary, in the sense that each of them has strong connections to some other scientific discipline besides artificial intelligence, and in most cases outside computer science. The field of robotics contains considerable work in mechanical engineering and in control engineering, for example, and the field of natural language processing as seen from A.I. integrates seamlessly with work in computational linguistics and with the cognitive sciences in general.

The cases of evolutionary computation and machine learning should also be mentioned separately. The actual literature that goes by the name of machine learning has a large component that makes heavy use of probability theory. This work is only remotely connected to the work in the A and B categories, or to the design of cognitive systems. There are also some niche areas, such as case-based learning and inductive logic programming, that are well related to the A and B categories, but these areas seem to be marginal in the contemporary research literature on machine learning.

With respect to evolutionary computation, there is a range of work that is represented at one end by the approach of 'artificial life', where one tries to develop computational life-forms using a simulated evolution process, and at the other end by work on generate-and-test techniques (i.e. systematic search techniques) that are applied to moderately complex objects that have a dynamic interpretation, for example simple scripts in a specialized language, or decision trees. In the former case the methodology is remote from what we have in the A and B listed areas; in the latter case there is a quite close connection.

According to this analysis, the A listed areas should be considered as the basic ones, the B listed areas are also included but are best studied in a second step, and the C listed areas integrate work that is close to the A and B areas, with work that has an entirely different character.

[1] One may ask why 'search' and 'optimization' are not listed in the most basic category, since these are mathematical techniques that are widely used in A.I. The answer is that from a systems-design point of view it is natural to first study the A listed topics in order to understand what the field is about, and then to proceed to the relatively more technical details of the algorithms being used. From a mathematical mindset one may prefer the reverse ordering.

8.2 Interdisciplinary Areas

The interdisciplinary topics in the C category form three natural groups according to the choice of interdisciplinary counterpart, as follows.

Interdisciplinary with Robotics

This includes the following ones of the topics listed above:

```
A  Agent-based and multiagent systems
C  Computer vision
A  Knowledge representation and reasoning
C  Robotics
```

Interdisciplinary with Human-Machine Interaction

This includes:

```
A  Agent-based and multiagent systems
C  Cognitive modeling and human interaction
A  Commonsense reasoning
C  Game playing and interactive entertainment
A  Knowledge representation and reasoning
C  Natural Language Processing
```

Interdisciplinary with Large Knowledgebases and their Use

This includes:

```
C  Information retrieval, integration, and extraction
B  Knowledge acquisition and ontologies
A  Knowledge representation and reasoning
C  Web and information systems
```

8.3 Alternative Views of Artificial Intelligence

Our view of the character and goals of Artificial Intelligence research was described in Chapter 1, which began with concrete examples of the use of intelligence in small children and in adults. This results in a view of A.I. where *cognitive agents* are emphasized, that is, one is concerned with the design of software systems that are able to *act intelligently* in a way that resembles human intelligence, and in a variety of naturally occurring environments. This is the classical view of Artificial Intelligence, and it was expressed, for example, in the proposal for the design of an 'advice taker' by one of the field's pioneers, John McCarthy at Stanford University.

8.3.1 Systems with Specialized Intelligence

This classical view may be contrasted with approaches that work towards the design of *specialized varieties of intelligence.* One example of this is the work on programs that play specific games, for example chess-playing programs. These are sometimes seen as examples of artificial intelligence, in the sense that intelligence is considered as essential for being able to play chess well by humans. Today there exist advanced programs that play chess extremely well and that are able to match even the world's best

human chess players. These programs are quite sophisticated and shall by no means be thought of as mere brute-force search programs. However, there are few connections between the techniques that are used in these programs and techniques that are needed for cognitive agents, except for a few basic principles such as the importance of combinatorial search.

Another example of specialized intelligence is for automatic translation of natural language. It was previously thought that general-purpose artificial intelligence would be required for good-quality language translation, since in order for a person to translate a text he or she must understand its contents. However, at the present stage of development it turns out that better translation results are obtained using specialized, 'stupid' translation programs that use statistical methods and large databases of known translation patterns, and knowledge-based language translation systems are not able to compete with them at present. This is another example of a specialized task that evidently makes use of general-purpose intelligence in humans, and where a specialized approach has been more successful so far in computers.

Given these observations one may wonder whether the most promising long-term strategy for intelligence in computers will be to build a number of specialized 'intelligences', like the ones just described, and to integrate them gradually. Some researchers have argued this position. Time will tell, but it is fair to assume that a cognitive system of the kind that is addressed in the present book will *at least* be one of the participating 'intelligences'. Whether it will just be one out of several, or whether it will also be the cohesive force that makes it possible to integrate the others, or whether it will eventually be the universal principle that includes the others, this remains to be seen.

8.3.2 Alternative Computational Infrastructures

The topics in the A and B categories of the topic list above are based on conventional computational technology: algorithms, datastructures, programming languages, software architectures, and the use of formal logic. This technology has been developed for, and is adapted to the use of the von Neumann computer design with a small number of processors that work with almost entirely passive memories.

This computational infrastructure is very different from the structure of the human brain with its extremely large number of concurrently active brain cells. Some branches of research propose that this is a serious handicap, and that a better strategy for the development of machine intelligence is to build computational systems that are more similar to the neural or neural-based system in people and in animals. This is the approach of *neural computation.*

A somewhat related view questions the precedence of the knowledge-based techniques in the A category above, and propose that it is better to first implement the counterparts of more elementary human behavior, including sensorimotoric behavior and other stimulus-response behavior. The proposal is that intelligence of the kinds discussed above – agent intelligence, chess-playing and language-translation capabilities, and so on – are based on these simpler behaviors and that they *emerge* from them, i.e. they are

evolved from them. These simpler behaviors can then conceivably be implemented either using neural networks or other alternative hardware architectures, or using conventional programming-language technology.

Approaches using alternative computational infrastructures have produced good results concerning the implementation of lower-level behavior, below the level that is normally called 'intelligence'. However, their usefulness for the construction of cognitive agents or systems with specialized intelligence of the kinds discussed above has not been demonstrated and there is no concrete indication that it will be a viable approach.

Index