UNIVERSITY OF OSLO
Department of Informatics

Introducing Actions in CommonSens: A Hybrid Agent-Based Approach

Master thesis

Kristin Simonsen

May 1, 2013

Abstract

Systems that provide ambient assisted living (AAL) are currently emerging at a rapid pace, and the amount of functionalities these systems provide is constantly growing. AAL provides assistance at home for sick, disabled or elderly people, by placing sensors in their homes. These sensors are used for monitoring and assistive purposes.

Activities of daily living (ADLs) are all the possible situations that may occur in the home. Situations we wish to monitor are called events. A complex event processing (CEP) system analyzes the data received from the sensors and detects events. In order for, e.g. the monitored person not waking up in the morning, to be detected by health personnel, CEP systems can send an alarm as a response to each event. A challenge is how to supplement such a system with the ability to respond to events in a more flexible way by acting in the environment, e.g. by turning on the lights and checking if the person wakes up. With actions, the CEP system can act upon the environment, using actuators placed in the home.

Events and actions are dual terms, since both describe a set of states in the environment. Events are predefined changes that may be detected, and actions are planned changes that may be executed upon the environment. Actions can be implemented in CEP systems using intelligent agents. An intelligent agent can be regarded as an entity which observes and acts upon an environment through sensors and actuators.

We have considered how the CEP system CommonSens can be extended to include the functionality of intelligent agents. For describing agents we use the planning domain definition language (PDDL). We propose an architecture that reflects the duality between events and actions. Additionally, we design an agent architecture for CommonSens, and propose the overall design. Properties in the system are chosen with two important ideas in mind: (1) The resulting design will represent the environment and current functionalities in CommonSens as accurately as possible. (2) The level of complexity in the intelligent agent is kept at a minimum in order to gain knowledge and experience with the behavior of agents.

We use a hybrid agent architecture which can handle alarms in a reactive manner, as well as having the functionalities needed for goal-directed behavior. The latter gives an opportunity for the system to support both atomic and complex actions. The hybrid agent architecture consists of three layers. The reactive layer is responsible for sending alarms to health personnel when a critical event has occurred. The action selection layer handles atomic actions. The planning layer constructs plans, which constitute complex actions. We also discuss how the life cycle of intelligent agents in CommonSens will look like.

Acknowledgements

First of all, I wish to thank my supervisor Ellen Munthe-Kaas for her guidance and support during the course of this thesis. I have an immense appreciation for our discussions of the possibilities in this important topic. I also want to thank my dear fiancé, Remy Kelley, for his patience, support and encouragement. Finally, I thank my friends and family for providing me with moral support.

Kristin Simonsen
University Of Oslo
May 1, 2013

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Ambient assisted living (AAL), also known as automated home care, is the placement of sensors in a private home to provide monitoring and assistance for the person living there. This type of system can give sick, handicapped or elderly people a great advantage as they could live safely at home for a longer period of time instead of being placed in a government-provided institution. AAL could for instance help sick people through recovery, by doing medical examinations in their homes each day and returning the results to the doctor. Another possibility is to provide help for people with disabilities by lifting them into the bathtub, opening doors for them and doing similar assisting tasks. Elderly people could benefit from reminders of taking their medication and turning off the stove. The patients this affects will avoid being isolated from their normal social lives and daily routines as well as maintaining their privacy and sense of independence. The society will also benefit financially. AAL is currently an active research field. The importance of this topic can be highlighted by mentioning the EU-funded AAL Joint Programme in Europe, which forms an international unity for research, development and innovation within this area[1].

Within the topic of AAL, one of the challenges researchers face is the detection of activities of daily living (ADLs). An ADL is anything that may occur in the home. It could for instance be a person walking from the living room to the kitchen. The system will use sensors for extracting this information. The sensors are chosen and placed as needed. Typical sensors are for instance cameras, temperature monitors and device controllers found in electronic kitchen devices. Not all ADLs are of interest to the system, since we wish to monitor the home for specific situations. The situations which are of interest are called events.

Events are predefined situations in the home which need attention from the system. It could be the detection of a decreasing temperature, a device left on for too long, or an irregular movement from the monitored person. The system must therefore be able to differentiate between the ADLs, which are expected in the home, and the events, which are unexpected and possibly dangerous. Events can be of different levels of complexity, because events can describe both simple and complex situations. Therefore we have two different types of events; atomic events and complex events. Atomic events are ground events which can not be split into two or more other events. An example of an atomic event is the detection of a light switch being switched on. Complex events include two or more atomic or complex events, and define complex situations in the home. A complex event could for instance be the detection of a person falling to the ground. The complex event must in this case define the timeframe from when the person is standing to the moment the person is detected lying on the ground.

Complex event processing (CEP) is a set of techniques used for analysis of complex events received from a stream of sensor data[2]. Sensors pull low-level data from the home, this data is represented as tuples. For complex events multiple tuples are needed from two or more different sensors. In order to get a translation of this data into readable information we need a CEP system. The CEP system will in this way provide a higher-level interface for declaring and detecting events. ADLs will continuously occur in the home, but the CEP system can choose to pull only those which are declared as events. Complex events may occur consecutively or concurrently, and a CEP system should be able to detect events in both situations. Not only can a CEP system detect events, it should also have the ability to respond to them. CEP functionality is currently used in e.g. enterprise applications with requirements of real-time reaction upon business critical events[3].

A CEP system which does not have an ability to counteract or actively respond to events that occur, is very good for monitoring purposes or for collecting statistics. These systems may have some event triggered functionality, such as notifying someone or setting off an alarm. In regard to AAL, a notification such as this would not be very helpful for instance for elderly people that often forget their medicine. We would rather give them a warning somehow, so they could solve the situation themselves. It could be useful to see if other kinds of responses could improve the functionality of the system. The techniques we need for implementing responses which react to events in a CEP system are actuators and actions.

Actuators are hardware entities that perform basically the opposite job of a sensor. While sensors have a sensing unit which detects certain attributes of its environment, actuators have an internal motor which moves or changes something in its environment. A good example is an actuator for moving the wheels of a robot. Actions describe the functions an application can use to act upon its environment. Actuators and actions are interconnected as sensors and events are. The hardware entity, the actuator or sensor, will be performing a task. This task is specified by the software entity, the action or event. Actions are executed by actuators and events are detected by sensors.

As for system responses, one can implement a link between events and actions in which a specific action is chosen as a direct response to an event. If this same robot had a camera as a sensor, and it detected as an event that an object is blocking the way, the application should somehow decide upon which action to perform next. A system with behavior like this is regarded a reactive CEP system. A reactive CEP system has a way of reacting to the events that occur, using actuators, in addition to detecting them. With regard to AAL, events can be viewed as a sign that "something is not right". Involving actions gives the system a way to respond to a patient's events.

Whenever possible, we should use existing techniques and systems to present ideas. At the University of Oslo, we have available an experimental CEP system, CommonSens[4]. It was developed within the Distributed MultiMedia Systems research group. CommonSens is a CEP system designed for the purpose of performing AAL in the homes of elderly people. The motivation for introducing such a system was the continuous growth in the percentage of elderly people in the world, and possibly a future problem of housing them in nursing homes. The system is set up by placing sensors in their homes, which can detect a fixed, predefined set of atomic and complex events.

Introducing actuators and actions also requires certain techniques. Existing functionality can be found in artificial intelligence (AI). Intelligent agents are entities in a system which receive percepts, AI's equivalent of events, and perform actions. We can regard an intelligent agent as an entity which observes and acts upon an environment through sensors and actuators. Percepts are extracted from and actions are executed upon an agent's environment. Each agent has an internal agent program which manages all reasoning within the agent. Reasoning in AI is the agent's ability to make the correct choices when taking actions. Intelligent agents can also make plans, think ahead and reason in uncertain environments. Some agents can even learn from their own behavior and past experience.

Since we wish to unite the CEP system CommonSens with actuators and actions using intelligent agents, we need to first explain how events are currently handled by CommonSens. Events are defined by queries. These queries are provided to the system at configuration time, and will be compared against the sensor data. If the sensor data match the conditions in one of the queries, the sensor data is identified as an event, and CommonSens will alarm health personnel so that the monitored person can be assisted in his or her own home. The events vary from the person not taking their medicine at the correct time to serious accidents as for instance the person falling to the ground. As a response to all of these events, the system will alarm health personnel. Clearly, when the amount of events is large, this response type is not efficient nor will it help solve the issue of reducing the load of the health personnel at nursing homes. When every event results in an alarm, their workload is not reduced optimally. If all events which are not severe could somehow be prevented or corrected, this would help keep the number of house visits of health personnel to a minimum.

The system as it is now might pose difficulties if introduced to several homes today. If there is a large number of queries, the number of events and therefore alarms would be very high. It is understandable that we wish to have many queries, since there are a number of different situations that need to be monitored. For instance, we might wish to check that the monitored person is taking his or her medicine and that the person wakes up at a certain time each day. With the current implementation, CommonSens would send an alarm at deviations from both of these situations. We propose using actuators to perform actions upon the home, which again should help in preventing an extensive number of events leading to an alarm. Intelligent agents is a technique that can be utilized for action description and action handling. Also, an intelligent agent has an internal set of rules or goals which it can use to search for a solution to the problem. The solution will consist of an atomic action or a complex action, also known as a plan. The examples given earlier could be solved by using intelligent agents in the following way. When the person does not take the medicine at the appropriate time, the tv monitor could display a warning to the monitored person. For the case of the person not waking up at a certain time, the system could turn on the lights in the bedroom or turn on an alarm clock and check if the person wakes up as a result of this.

In this thesis we consider how CommonSens could be extended to include the functionality of intelligent agents. Our goal is that the resulting architecture will reflect a duality between events and actions.

There would be many advantages of using intelligent agents in CommonSens. Plans are, in fact, complex actions. This enhances the duality between events and actions. Additionally, intelligent agents can perform goal-directed behavior. Nevertheless, there are many choices that must be made before implementation. These choices will determine how the resulting structure and behavior of the system will be. We need to decide upon agent programs, which define how the agent will perform reasoning during action selection. An agent architecture must be chosen, this will determine the overall structure and information flow in the system. Planning can be done in numerous ways, so we need to choose which method works best for our domain.

In order to demonstrate intelligent agents and planning we use an illustrative example, Shakey the robot. The example was a mandatory exercise in the course INF5390 in the subject artificial intelligence at the University of Oslo[1]. Shakey originates from the work of Rosen and Nilsson[5]. We will use the example from the mandatory exercise because it limits the scope of Shakey's actions. The robot Shakey maneuvers through an environment using its sensors and actuators. It can move, push boxes, stand on boxes and turn on and off light switches. Shakey is a very simple example of how intelligent agents work in practice, this will help in our understanding of how CommonSens can make use of the functionalities of intelligent agents. It will also be useful in explaining planning, and how a planning algorithm may work.

In Chapter 2 we present relevant background material for the rest of the thesis. We present details about CommonSens with all its components and models, along with intelligent agents and classical planning. In Chapter 3 we review agent architectures and discuss how we can find the best architecture for our system. Chapter 4 analyzes the design of intelligent agents and planning using the Shakey example. In Chapter 5 we propose the design of our architecture, and show how it can be included in CommonSens. Chapter 6 will provide our conclusion and propose possible future work.

[1]Exercise can be found at this web page: `http://www.uio.no/studier/emner/matnat/ifi/INF5390/v12/undervisningsmateriale/ovinger/INF5390-2012%20Exercise%202.pdf`

Chapter 2

Background and Related Work

Over the past few years, the concept of smart homes has become a popular research topic within the health care sector world wide. Ambient assisted living (AAL) as an application domain aspire to significantly improve the well being of people in the comfort of their own homes. CommonSens is specifically designed for taking care of the elderly. As the average life expectancy continues to grow, there will be a higher percentage of elderlies in the not so distant future [6].

In this chapter, we will present the background material of this thesis. Section 2.1 will briefly explain CommonSens, and Section 2.2 will go deeper into the details of its event model. Both sections will present the work of J. Søberg[4], the developer of CommonSens. Section 2.3 gives us the details of intelligent agents, a technique that emerged from Artificial Intelligence. This section illustrates the work of S. J. Russell and P. Norvig[7].

2.1 CommonSens

”CommonSens is a multimodal complex event processing (CEP) system for detecting activities of daily living (ADLs) from sensors in the home”[4]. The system should provide the necessary tools for observing the monitored person and his or her environment, and also alarm health personnel in emergency situations. When implemented in a large scale, CommonSens should minimize Health Care costs, reduce the quantity and workload of health personnel and hopefully improve the life satisfaction of the elderly person.

In the process of introducing a CEP system to an environment, one needs to define specific models for describing the sensors and its data tuples, the environmental objects and how events are handled. Accordingly, CommonSens has three models. The sensor model describes the properties of each sensor and the relationships between physical and logical values. The properties of a sensor are its capabilities and coverage area. The environment model is needed for describing the three dimensional geography of the monitored environment. Properties as placement of objects and walls, shapes and sizes of the objects and signal behavior are all handled by this model. Signal behavior can differ depending on the object, thus each object has its own permeability value defining if the signal goes through, stops or bounces off the object. Finally, the event model detects states and state transitions that occur in the environment and compares these to some predefined queries. In a home environment, the majority of the sensor data is needless, so this model has the task of detecting the data of interest. The concepts and properties of the event model are defined in Section 2.2.

The system is configured to match each subject's personal needs. The monitored person has his or her own health issues, living arrangements and daily routines. A health services employee has knowledge of what is required for the monitored person to be able to live in a healthy and safe state. The application programmer will write the queries in cooperation with the health services employee.

2.2 Event Model

The event model describes the concept of events and defines all of its semantics. All possible things that can happen in the monitored environment are generalized as states and state transitions. A state consists of a set of variables including data values. The data values give information about certain measurable properties which describe what the environment look like. We will introduce these properties shortly. A state transition, on the other hand, gives information about a change in one or more of the state's data values. We can visualize this with a door that opens, its data value then changes from state closed to state open, which in itself is a state transition.

The conceptual world will have a vast number of possible states and state transitions, but only a small subset of these will be of interest, i.e., events. This is modeled in Figure 2.1. The system uses queries for describing events. States are detected by the sensors, and if the state values match the query description

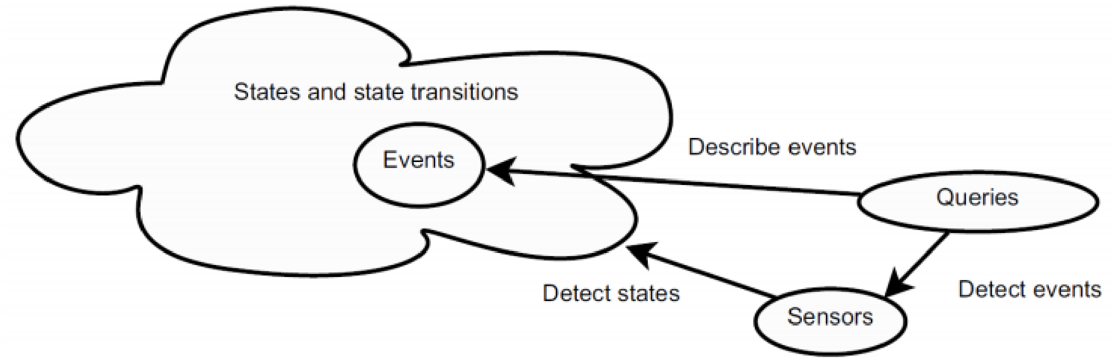

Figure 2.1: The relation of events and states in the environment [4]

the state will be identified as an event. A state transition is an event if all states involved match a set of queries.

2.2.1 Temporal and Spatial Properties

The temporal and spatial properties is an important part of the event model, from which the system can acquire information regarding when and where an event occurred. Firstly, we need to address the temporal properties of an event.

"A timestamp t is an element in the continuous time domain T: t ∈ T."

Each data tuple has two timestamps, a start timestamp t_s and an end timestamp t_e. The range of these two timestamps is defined as the time interval $\tau \subset$ T. The duration of τ can be found as $\delta = t_e - t_s$. This gives us a chance to query the duration of an event, for instance how long time a person uses to take his or her medicine. Secondly, we have spatial properties of an event. Each event contains a specific Location of Interest (LoI).

"A location of interest (LoI) is a set of coordinates describing the boundaries of an interesting location in the environment."

The system needs LoIs for each state, because it gives a specific location for where the event happened. A home has several similar objects. The queries specify LoIs as part of their conditions, and therefore need these LoIs to be covered by corresponding sensors.

Overall, the temporal and spatial properties help pinpoint events out of all other events and state transitions. They are also the key components of both events and queries, which will be introduced in Section 2.2.2 and Section 2.2.3.

2.2.2 Atomic and Complex Events

The term event has many different interpretations, also within the field of computer science. In [4], the author has defined the term as follows.

"An event e is a state or state transition in which someone has declared interest."

There are two different types of events, atomic and complex events. Later on in the thesis, the term event will be used for both. An event is considered an atomic event if it is impossible to divide it into two or more events. It contains one set of properties for one specific duration and LoI.

"An atomic event e$_A$ consists of four attributes: e$_A$ = (e,loi,t$_s$,t$_e$). e is the event loi is the LoI where $(|loi| = 1 \vee loi = \emptyset)$ and $(t_s, t_e \in T)$."

For two atomic events A and B with timestamps $\{a_s, a_e\}$ and $\{b_s, b_e\}$ we have six classes of interval relations inspired by J. F. Allen[8]. The first class defines consecutive events and the five others are concurrent events.

A before B : $a_e \leq b_s$
A equals B : $(a_s = b_s) \wedge (a_e = b_e)$
A overlaps B : $(a_s < b_s) \wedge (a_e > b_s) \wedge (a_e < b_e)$
A during B : $((a_s > b_s) \wedge (a_e \leq b_e)) \vee ((a_s \leq b_s) \wedge (a_e < b_e))$
A starts B : $(a_s = b_s) \wedge (a_e < b_e)$
A finishes B : $(a_s > b_s) \wedge (a_e = b_e)$

"Two atomic events e_{Ai} and e_{Aj} are consecutive iff $e_{Ai}.t_e < e_{Aj}.t_b$."

The events e_{Ai} and e_{Aj} are consecutive if the end timestamp of one of them is less than the start timestamp of the other. This proves that they do not overlap in time.

"Two atomic events e_{Ai} and e_{Aj} are concurrent iff $\exists t_u$,

$(t_u \geq e_{Ai}.t_b) \wedge (t_u \geq e_{Aj}.t_b) \wedge (t_u \leq e_{Ai}.t_b) \wedge (t_u \leq e_{Aj}.t_b)$"

Explained, for e_{Ai} and e_{Aj} to be concurrent there needs to exist a t_u that can be found in both intervals. This proves the events do overlap.

When two or more atomic events are related to each other according to one of the interval relation classes, the set of these events is called a complex event.

″A complex event e_C is a set of N atomic events: e_C = {e_0, ..., e_{n-1}}.″

2.2.3 Atomic and Complex Queries

In order to detect an atomic event, we need a predefined query which searches for this event in the environment. LoIs and timestamps are of the same syntax as in the events. Additionally, the queries use capabilities adapted from the sensor model.

″A capability c is the type of state variables a sensor can observe. This is given by a textual description c = (description)″

Each sensor has its own set of capabilities, which maps queries and sensors. This way, the application programmer will address the capabilities, instead of querying a sensor directly. This choice of architecture is based on the fact that the sensor readings have different complexity and data types.

″An atomic query q_A is described by a tuple: q_A = (cond, loi, t_s, t_e, preg). cond is a triplet (C, op, val), where c is the capability, op $\in \{=, \neq, <, \leq, > \geq\}$ is the operator, and val is the expected value of the capability. If set, loi, t_s, t_e and preg specify the spatial and temporal properties.″

The most relevant part of the tuple is the condition, which describe a state through the state variable called capability. If we want to detect the state of a door, which is a capability, the condition can possibly be OpenDoor = True. We can also detect a specific door, by for instance setting loi = Bedroom. There could also be some temporal properties which could tell the system when the door is expected to open. All these properties need to be set a priori by the application programmer.

The system needs to run a number of concurrent queries in order to discover a complex event. The application programmer also has to write queries which describe the fact that the atomic events are concurrent. For this purpose, CommonSens also has the ability to run complex queries.

″A complex query q_C is a list of atomic queries, and logical operators and relations ρ_i between them: q_C = $\{q_{A0}\rho_0 ... \rho_{N-2}q_{AN-1}\}$. If the complex query only consists of one atomic query, $\rho_0 = \emptyset$".

q_{Ai} are atomic queries and $p_i \in \{\&\&, ||, !, \rightarrow\}$ are logical operators describing relations between the atomic queries. When examining the data tuples received from a query, the following attributes are revealed.

(timestamp, sensor, capability, value, t_s, t_e)

In addition to the properties of an event, the data tuples also have information about which sensor the data is coming from. An example of a tuple will be (151000, $Oven_0 1$, GetTemperature, 22, 141000, 141000). The first value is the timestamp when the tuple arrived, the second is the sensor from which the state came from, the third is the capability, the fourth the value of the state variable, and the fifth and sixth are the timestamp of the state. If a query should detect this exact instance, the condition should be GetTemperature = 22.

2.3 Intelligent Agents

"An agent is anything that can be viewed as perceiving its environment through sensors and act upon that environment through actuators"

Intelligent agents will choose the best possible rational action in a given situation. An agent will for each incoming set of percepts, data received from its sensors, select an action based on these and its internal knowledge that will maximize its performance measure.

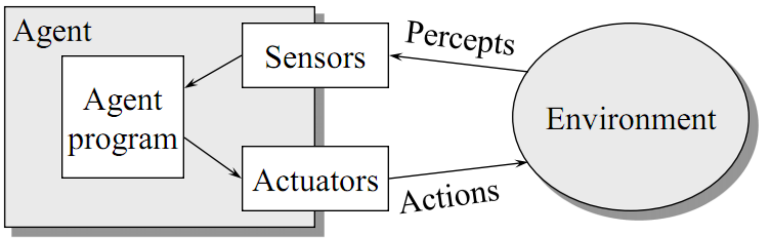

Figure 2.2: An intelligent agent [7]

As noted in Figure 2.2, an intelligent agent can be considered as an entity that interacts with its environment. Internally, it has an agent program which is used to map percepts and actions. How the agent program is structured will determine the functionality and performance of the agent.

2.3.1 Environment Properties

Before starting with the task of building an agent, the task environment needs to be established. The task environment can be explained as the problems to which intelligent agents are the solution. As the authors[7] propose, we use the PEAS (Performance, Environment, Actuators, Sensors) description.

Table 2.1 shows an example from the book[7], where the intelligent agent is a taxi driver robot. When designing an intelligent agent, one must firstly consider what the performance measure should be. A performance measure captures the desirability of a sequence of states the environment goes through. This sequence of states occurs because the agent performs a sequence of actions according to the percepts it receives from the environment. There will very often be conflicting goals, so the performance measures will require a certain amount of trade-off.

Agent Type	Performance Measure	Environment	Actuators	Sensors
Taxi driver	Safe, fast, legal, comfortable trip, maximize profits	Roads, other traffic, pedestrians, customers	Steering, accelerator, brake, signal, horn display	Cameras, sonar, speedometer, GPS, odometer, accelerometer, engine sensors, keyboard

Table 2.1: PEAS description of the task environment for an automated taxi[7]

"As a general rule, it is better to design performance measures according to what one actually wants in the environment, rather than according to how one thinks the agent should behave."

Thereafter, the environment must also be considered. The more restricted the environment is, the easier the problem becomes. Actuators and sensors must also be established. In order to choose the appropriate agent design, there are some techniques for categorizing some properties of the task environment. These properties will now be thoroughly explained.

PROPERTIES OF TASK ENVIRONMENTS

Fully observable or **partially observable**: The environment is fully observable when the agent's sensors at each point in time give access to the complete state of the environment. The effect is that the sensors in the system can detect all aspects that can be relevant to the selection of the next action. Fully observable environments are easier to implement because the agent does not have to keep track of how the system evolves. In the book, this feature is known as the percept history, and if needed, the environment is partially observable.

Single agent or **multiagent**: For a more complex agent system, we need to divide the problem into two or more agent components. These components will cooperate towards common goals or compete against each others goals. This describes a multiagent environment. If an agent's behavior is maximizing a performance measure, whose value is depending on another objects behavior, we need to make this object an agent as well. If this is not the case for any of the objects, a single agent environment will suffice.

Deterministic or **stochastic**: A deterministic environment is defined by the fact that the next state of the environment can be completely determined by the current state and the next action executed. Environments where this statement is false, are stochastic. For a multiagent environment, it can be deterministic even though an agent may not predict the next action of other agents.

Episodic or **sequential**: An episodic environment implies that the agent's experience is divided into atomic episodes where each episode consists of the agent perceiving one event and performing one action. Crucially, the agent's decision making does not depend on the behavior in previous episodes. Otherwise, the environment is sequential.

Static or **dynamic**: An environment is dynamic if the environment can change its state during the time when the agent deliberates. In such an environment, the agent needs to keep sensing changes while it is deciding on the next action. If the environment does not change during deliberation, it is static.

Discrete or **continuous**: When the environment is discrete, each agent has a finite number of states, events and actions. Furthermore, each state has a finite range of values. If the states, events and actions can take an infinite number of values, the environment is continuous.

Known or **unknown**: In a known environment all the outcomes of all possible actions can be predicted. The application programmer should know how the technology works, and therefore how the actions will physically change the state of the environment. If this is not feasible, the environment is unknown.

2.3.2 Agent Programs

Intelligent agents consist of both architecture and program. The architecture of an agent, described in Section 2.3.1, is the physical computing device the agent is running on, with its physical sensors and actuators. We will now proceed to the agent program. An agent program as in Figure 2.2 takes the current state as an input and returns a single action. It can be visualized in pseudo code or as a schematic diagram. We will briefly explain all kinds of agent programs.

TABLE-DRIVEN AGENT

The table-driven agent has the simplest implementation of all agent programs. It stores a table of all percepts and actions. When a new percept is received, it performs a lookup function in the table and returns the appropriate action. The pseudo code for a table driven agent is given in Figure 2.3.

```
function TABLE-DRIVEN-AGENT(percept) returns an action persistent:
percepts, a sequence, initially empty
table, a table of actions, indexed by percept sequences, initially fully
specified

append percept to the end of percepts
action ← LOOKUP(percepts, table)
return action
```

Figure 2.3: Pseudo code for a table driven agent [7]

This agent program is not scalable. The table size grows exponentially as the complexity of the problem grows. The lookup table will contain $\sum_{t=1}^{T} |P|^t$ entries, where P is the set of possible percepts and T is the total number of percepts the agent will receive. A simple game of chess would have a table with at least 10^{150} entries.

Simple reflex agent

The simple reflex agent assumes a fully observable environment and it is based on the condition-action rule. The percept history is ignored, and the action selection is only based the current percept. As shown in Figure 2.4, each percept is checked against a set of conditions and the resulting action will be executed.

```
function SIMPLE-REFLEX-AGENT(percept) returns an action persistent:
rules, a set of condition-action rules
state ← INTERPRET-INPUT(percept)
rule ← RULE-MATCH(state, rules)
action ← rule.ACTION
return action
```

Figure 2.4: Pseudo code for a simple reflex agent [7]

As we see in Figure 2.5, the agent will for each new percept check which action the rule chooses and returns this action.

Model-based reflex agent

The model-based reflex agent introduces an internal state as a way of handling the part of the environment it can not observe at each point in time.

As seen in Figure 2.6, the agent will for each new percept check the model and find a new state based on the previous state, the previous action and the new percept. It then derives an action using the condition-action rule as the simple reflex agent does.

As we see in Figure 2.7, the current percept is combined with the internal state and compared to the agent's own model of the environment. The agent's own model keeps track of "how the world evolves" disregarding the agent's own actions. In addition to this, it checks "what my actions do", i.e. how its

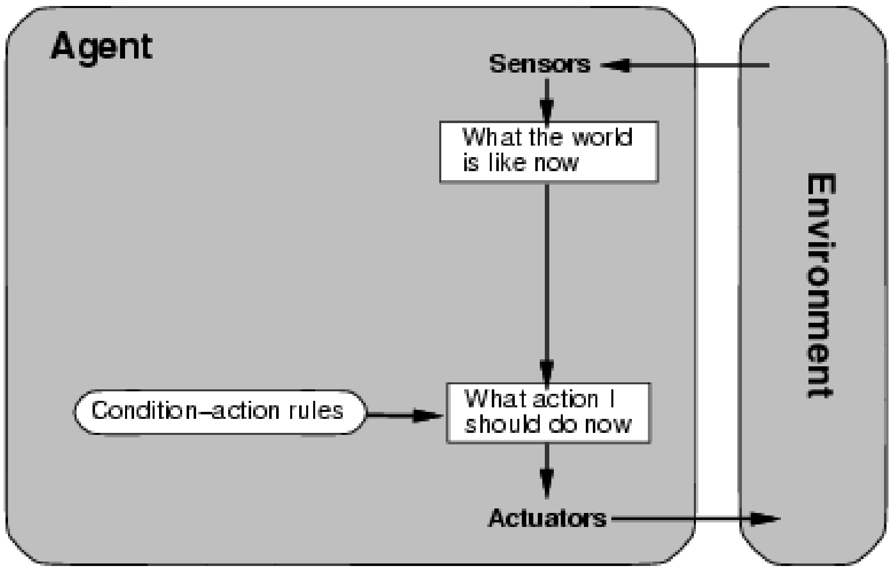

Figure 2.5: Schematic diagram of a simple reflex agent [7] [9]

```
function MODEL-BASED-REFLEX-AGENT(percept) returns an action
    persistent: state, the agent's current conception of the world state
        model, a description of how the next state depends on the current state
            and action
        rules, a set of condition-action rules
        action, the most recent action, initially none

    state ← UPDATE-STATE(state, action, percept, model)
    rule ← RULE-MATCH(state, rules)
    action ← rule.ACTION
    return action
```

Figure 2.6: Pseudo code for a model-based reflex agent [7]

actions affect the environment.

GOAL-BASED AGENT

When an agent has a superior goal state, we need each action to be chosen so that the agent will be closer to that goal for each action executed. This describes the goal-based agent, where the action selection is part of a search or a planning algorithm. The goals are functionally different from the condition-action rules,

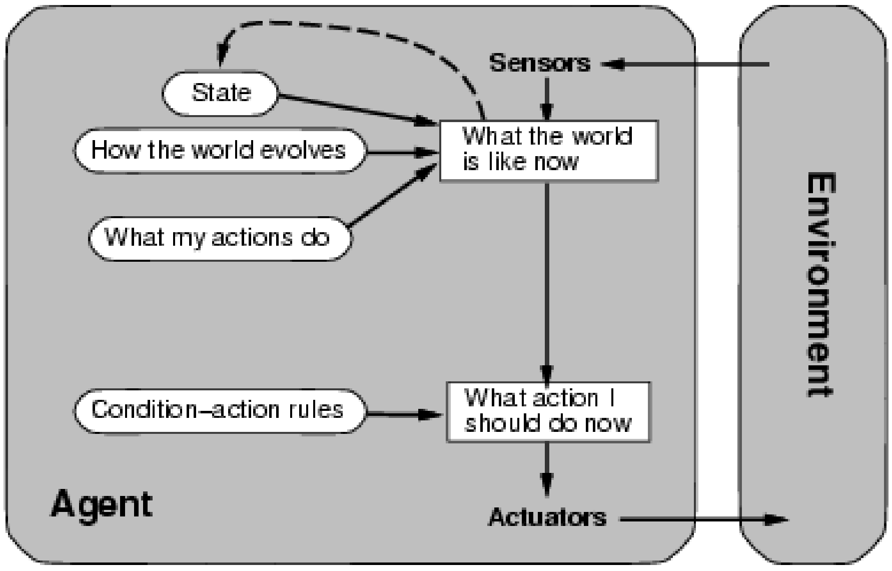

Figure 2.7: Schematic diagram of a model-based reflex agent [7] [9]

because they have a consideration for the future state of the environment and how best to get to a state closer to the goal state.

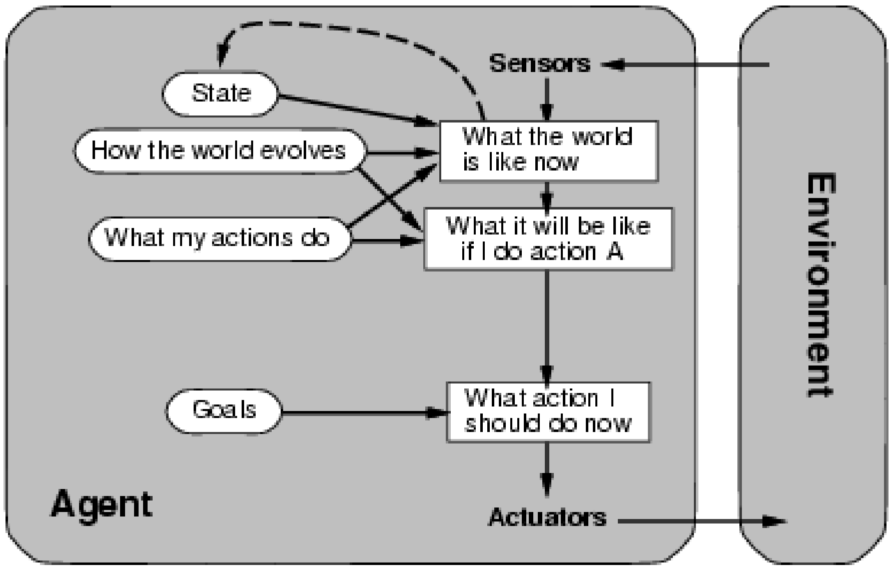
Sensors
State
How the world evolves
What the world is like now
What my actions do
What it will be like if I do action A
Environment
Goals
What action I should do now
Agent
Actuators

Figure 2.8: Schematic diagram of a goal based agent [7] [9]

In Figure 2.8, we see a new element "What it will be like if I do action A", which implies an algorithm for finding the best action A for the overall goal state G. This step will be part of a search algorithm for the goal state.

UTILITY-BASED AGENT

In the real world, goals are not sufficient for an agent program in order to get to the goal in a proper manner. There might be many action sequences which can get to the goal, but they will not all be the best solution. Properties as time, safety and cost can all be factors which make one action sequence better than another. We call these properties utilities. Thus, we sometimes need a utility-based agent. This agent has a utility function, which maximizes an internal performance measure when selecting actions. When the utilities have conflicting goals, for example time and cost, the utility function should specify the adequate trade-off.

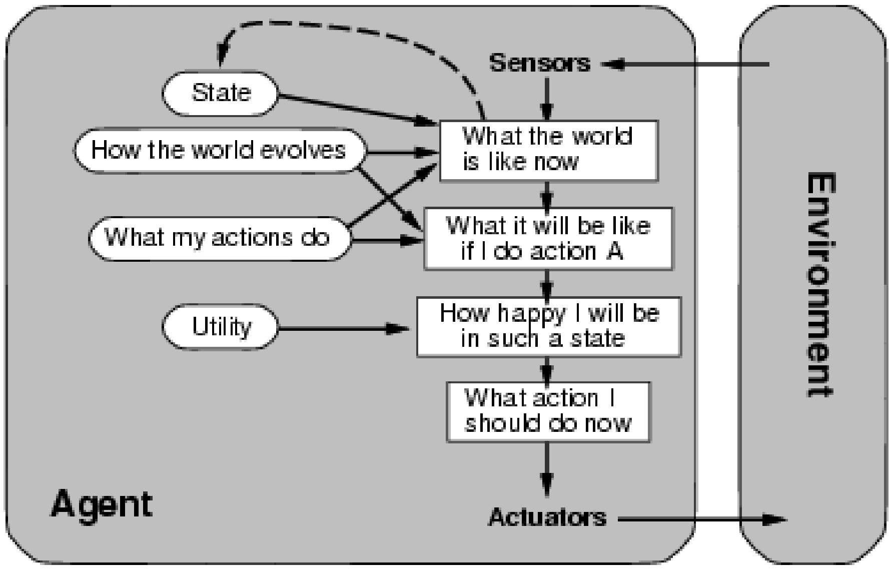
Sensors
State
How the world evolves
What my actions do
Utility
What the world is like now
What it will be like if I do action A
How happy I will be in such a state
What action I should do now
Agent
Actuators
Environment

Figure 2.9: Schematic diagram of a utility based agent [7] [9]

As we can see in Figure 2.9, the utility function will choose the next action in "How happy will I be in such a state". The action with the best utility will be chosen accordingly.

LEARNING AGENT

Building an agent program is hard and time consuming work. Alan Turing made an estimate of the work of building these machines by hand and concluded that ”Some more expeditious method seems desirable”[7]. The concept of teaching the machine to learn is preferred in many AI projects for building state-of-the-art systems. A learning agent is an agent that monitors and deliberates on its own success and overall performance.

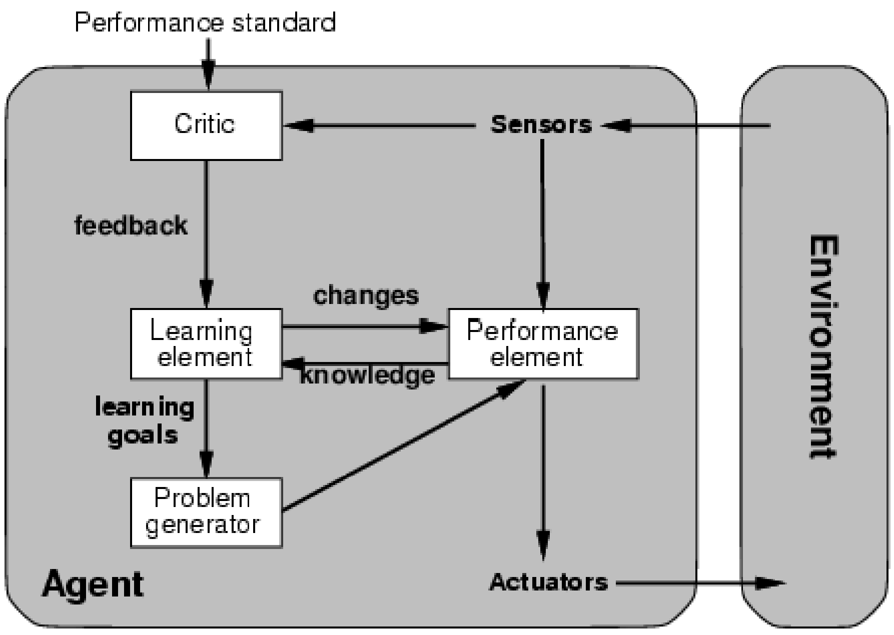

Figure 2.10: Schematic diagram of a learning agent [7] [9]

In Figure 2.10, the performance element can be viewed as the whole agent in Figure 2.9. The learning elements will make improvements on the performance element based on feedback from the critic, which measures how well the agent is doing based on a predefined performance standard. The problem generator will suggest new random actions in order to challenge the agent, which will result in a broader learning experience.

2.3.3 Classical Planning

When the agent has a consideration for the future state of the environment and possibly a goal state, a planning algorithm will choose the best action for reaching a certain goal. The state of the environment is defined in a factored representation; a set of variables. In AI, there are multiple languages that can be used to define states and actions. These languages use logic sentences, which consist of certain variables or functions bound together by connectives. More information regarding this can be found in Appendix A.

Propositional Logic

Propositional logic decides upon the truth of a sentence. A complex sentence combines atomic or complex sentences using logical connectives. A truth table can then be utilized to solve inference problems. We give an example, where the proposition symbol L_x is true if the given light is on. The sentences R_x are rules or proven truths that will be used derive the truth of a proposition symbol.

- L_1 and L_2 are on. This is a given truth. The connective is AND in the truth table.

$R_1 : L_1 \wedge L_2$

- L_2 and L_3 are bi-conditional. This means L_2 is true if and only if L_3 is true. This connective can be solved in a truth table as XOR.

$R_2 : L_2 \Leftrightarrow L_3$

We will show the truth of L_3 through a truth table. The truth table represents all possible models, which refer to all the agent's "possible worlds". The knowledge base KB is true where R_1 through R_x are true. We can see in Figure 2.2 that where KB is true, L_3 is true. Thus, L_3 is entailed by the KB.

L_1	L_2	R_1		R_2	KB
true	true	true	true	true	true
true	true	false	true	false	false
true	false	true	false	false	false
true	false	false	false	true	false
false	true	true	false	true	false
false	true	false	false	false	false
false	false	true	false	false	false
false	false	false	false	true	false

Table 2.2: The truth table of KB in propositional logic

The entailment KB $\models L_3$, is proved because L_3 is true in every model in which KB is true. In other words, for all "possible worlds" the agent will encounter, given the "rules" or sentences R_1 and R_2, L_3 is true. This way, we have proved the truth of L$_3$.

KB $\models$ *L$_3$ proves that the light L$_3$ is on.*

FIRST-ORDER LOGIC

First-order logic creates relations and functions between objects. It has a more human-friendly syntax than propositional loc, especially in very complex sentences.

The problem above would be represented in FOL with the objects L$_1$, L$_2$ and L$_3$ which are defined by constant symbols. There could also be relations, which are defined by predicate symbols. The most definite relation in this example is if a value describes a light switch, Switch, all the objects L$_i$ in Switch(L$_i$) are true. In addition, we can represent "has the state" and the symbol could be State(x, y). A relation could also be a function, where each input has only one value. Here, "located at" could be an example, and LocatedAt(x, y) the function symbol. We will only use the given constant and predicate symbols in the following example.

Quantifiers are often used in FOL, they express properties representing sets of objects. The universal quantification symbol $\forall x$ states a fact for all variables. It can be pronounced "for all x". The existential quantification symbol $\exists$ states a fact for at least one existing variable. This symbol can be pronounced "there exists an x".

Let x be the state of a light switch, ON or OFF. Our sentences are given below.

- For all x, the state of L$_2$ is x if and only if the state of L$_3$ is x.

$\forall x$ *State(L$_2$, x)* $\Leftrightarrow$ *State(L$_3$, x)*

- The state of L$_2$ is ON.

State(L$_2$, ON)

We can derive the state of L$_3$ by using the inference rule Generalized Modus Ponens. For all i in p$_i$,

$$\frac{p'_1, p'_2, ..., p'_n, (p_1 \wedge p_2 \wedge ... \wedge p_n \Rightarrow q)}{SUBST(\theta, q)}$$

The atomic sentences are defined by p_i, p'_i and q, and there is a substitution θ where SUBST(θ, p'_i) = SUBST(θ, p_i). We derive premises from p_i, our sentence with two unknown variables, p'_i, with one unknown variable, q, our query, and θ, our substitution value. The premises are:

p_1 is State(L_2, x)
p_2 is State(L_3, x)
p'_1 is State(L_2, ON)
p'_2 is State(L_3, y)
q is State(L_3, x)
θ is {x/ON, y/ON}
SUBST(θ, q) is State(L_3, ON)

As we can see above, FOL uses Modus Ponens and substitution to solve L_3. Using the premises and substitution rule, we can derive the value x in our query q; State(L_3, x). The inference rule produced the same solution as in propositional logic, but is easier to follow for humans and much more scalable.

Planning Domain Definition Language

In classical planning we use a language called PDDL (Planning Domain Definition Language) to construct plans. A complete planning problem needs to be defined through the initial state, all actions available and the goal state. PDDL has a lifted representation, thus it lifts the reasoning level from complex propositional logic to a subset of first-order logic. The only logical connectives used are $\land$ and $\neg$. Figure 2.11 shows an action schema representing the action of turning on a light switch.

```
Action(LightOn(l),
PRECOND: Off(l) ∧Switch(l)
EFFECT:On(l))
```

Figure 2.11: Action schema for turning on a light switch

The components of the action schema have important purposes. PRECOND states the preconditions of the action. Thus, the sentence following it must be true in order for the action to be executed. EFFECT shows how the action affect the state of the environment. The action schema can be thought of as being universally quantified, and all kinds of values can instantiate the variables. However, there is one rule that decides if the values can be used.

"Action a is applicable in state s if the preconditions are satisfied by s."

"$a \in ACTIONS(s) \Leftrightarrow s \models PRECOND(a)$"

An action a available for state s can be executed iff every positive literal in PRECOND(a) is in s and every negative literal is not. We use the example from Figure 2.11 to show the rule.

$\forall kitchenLight(LightOn(kitchenLight) \in ACTIONS(s)) \Leftrightarrow$
$s \models (Off(kitchenLight) \wedge Switch(kitchenLight))$

Basically, if the light is not off, or not a value of predicate Switch, the action is not applicable.

We will proceed with an example of a complete planning problem. Figure 2.12 shows a morning routine problem. Firstly, the initial state tells us how the state of the environment is before any action has occurred. We use constant symbols for the objects. L_1, L_2 and L_3 are true in predicate Switch, C is in predicate CoffeeMachine and A is in predicate AlarmClock. The predicates Off and On give the state of the objects. Secondly, the goal is to start the alarm clock, turn on the lights in the bedroom, bathroom and kitchen, and start the coffee machine. Finally, we have all the action schemas. These actions should suffice to complete the goal state.

Before proceeding to the planning, we must state some important principles in PDDL.

1. All the predicate states are ground, non-dividable and functionless.
2. *The closed-world assumption* regards all states that are not mentioned as false.
3. *The unique names assumption* means that all constants are distinct.

We will now construct a plan using the three actions StartAlarmClock, LightOn and StartCoffeeMachine. As we see in the action schemas, their effects can alter the initial state towards the goal state. The plan for achieveing

```
Init( Off(L_1) ∧Off(L_2) ∧Off(L_3) ∧Off(C) ∧ Off(A) ∧ Switch(L_1)
∧Switch(L_2) ∧Switch(L_3) ∧CoffeeMachine(C) ∧ AlarmClock(A))

Goal( On(A) ∧On(L_1) ∧On(L_2) ∧On(L_3) ∧On(C))

Action(StartCoffeeMachine(c),
PRECOND: Off(c) ∧CoffeeMachine(c)
EFFECT:On(c))

Action(LightOn(l),
PRECOND: Off(l) ∧Switch(l)
EFFECT:On(l))

Action(StartAlarmClock(a),
PRECOND: Off(a) ∧AlarmClock(a)
EFFECT:On(a))
```

Figure 2.12: The PDDL description of the morning routine problem

the goal is stated below.

[StartAlarmClock(A), LightOn(L_1), LightOn(L_2), LightOn(L_3), StartCoffeeMachine(C)]

When these actions are executed, they change the initial state of the environment into a new state of the environment, which completes the goal. The goal state then becomes a subset of the new state of the environment.

```
Action(Increase(t, o),
PRECOND: TooLow(t) ∧On(o) ∧ Temperature(t) ∧ Oven(o)
EFFECT:Increase(o) ∧ Normal(t))
```

Figure 2.13: Action schema for increasing the temperature

A real problem is when an action should have conditions. For example, the temperature example needs to monitor the effects of the previous action properly and react accordingly. Figure 2.13 shows a regular action schema for increasing the temperature.

```
Action(ConditionalIncrease(t, o),
EFFECT: when StillIncreasing(o): noop
∧ when ¬ StillIncreasing(o): Increase(o) ∧Normal(t))
```

Figure 2.14: Conditional action schema for increasing the temperature

In Figure 2.14, we see how a conditional action schema checks if the electric oven is still working on increasing the temperature. This will result in a no-op, which means no operation is needed.

Chapter 3

Architecture and Methodology

We will in this chapter go into details regarding current agent architectures available and why this is important to define at an early stage of intelligent systems design. Our overall goal throughout this chapter is to investigate which agent architecture is appropriate for our domain. There will be made some assumptions about the system, we will aim for a simple structure as a basis for a possible future extension.

Since we wish to bring together CommonSens and intelligent agents, we need to firstly assess the properties of intelligent agents and how these fit into CommonSens. We will address the three well-known agent architectures in Section 3.1. We will proceed with a classification of our system-to-be in Section 3.2. Then we will state the methodology of three existing architectures in Section 3.3. In Section 3.4 we discuss these methods with regard to CommonSens, and its current architecture.

3.1 Agent Architecture

Müller[10] proposes that every application developer should be certain that an object-based approach is insufficient before deciding upon using an agent-based paradigm. If the agent-based properties are needed in the system-to-be, we will need to use agents. These properties are stated below.

- *Dynamic*: The system must adapt and respond to a dynamic environment

- *Failures*: The failures that occur must be dealt with through re-scheduling, re-planning or re-allocation
- *Behaviors*: The behaviors can be long-term, goal-directed or short-term and reactive
- *Guaranteed time*: Both critical and complex reaction and response times must be guaranteed
- *Node properties*: Distributed, autonomous or heterogeneous nodes
- *System properties*: Reliability, robustness and maintainability
- *Information*: Complex, decentralized resource allocation problems when encountering incomplete information
- *User*: Need of a flexible interaction intended for human users

We will now review all these properties with regard to CommonSens. The system needs to adapt and respond to the dynamic environment. Environment properties are discussed in Section 4.1, there we elaborate on why our environment is dynamic. Failures could be a real threat to the monitored person, so the system should be able to handle this. The monitored person could benefit from all behaviors stated, i.e. long-term, goal-directed, short-term and reactive. Examples could be long-term movement monitoring, short-term health crisis reaction, and a goal-directed daily physical activity training program. Response times in CommonSens are probably of human scale. Alarms must be sent at a reasonable time, but a millisecond difference would not be considered a serious problem. However, delays in terms of several minutes are probably unacceptable, and delays in terms of hours are definitely unacceptable.

The nodes should be distributed, autonomous and heterogeneous. The sensors in CommonSens have these properties, so actuators should of course also be implemented with this in mind. As for the incomplete information issue, the work of Søberg[4] does not explicitly state that this leads to a resource allocation problem in CommonSens. Finally, CommonSens does focus on a human user. As for a flexible interaction, this may be a good idea for the future. A way for the user to interact with the system should be achieved through an interactive tablet of some sort. An example of a situation could be that the system asks the user if the stove should be on, and the user replies yes because he or she is making a time-consuming meal. As we see, most of these properties are needed in the system-to-be. We therefore conclude that agents are, in fact, needed in CommonSens.

In earlier work by Wooldridge and Jennings[11], the importance of a defined agent architecture was discussed as a prominent factor to the success of the implementation of an agent-based system. The agent architecture should be modeled in a way so that the design of the system will fulfill all the properties of the agents. There are different approaches to the architecture, with vastly different resulting implementations. Ferguson[12] differentiates the deliberative, reactive and hybrid architectures in the following way. If the agent deliberates upon the options that are present when choosing an action, the agent is considered deliberative. Alternatively, if the agent's choice of action is preprogrammed or "hardwired" to execute in response to the environmental conditions, the agent is considered reactive. The architectures which realise the choice of action by using a combination of deliberative and reactive techniques, are considered hybrid. In the remaining part of this section, we will use related work from Müller[10], Wooldridge and Jennings[11] and Wooldridge[13] to describe the possible agent architectures.

3.1.1 Deliberative Architecture

The classical approach to building agent systems emerged from how symbolic AI views agents; a type of knowledge-based system. The basis of the symbolic AI paradigm is Simon and Newell's physical symbol system hypothesis. They assume an internal representation of the world maintained by the agents, and that symbolic reasoning helps modify a mental state. From this hypothesis, deliberate agents emerged. Ideally, deliberate agents can make decisions based on logical reasoning, using an internal symbolic representation of the world. Developers might be fooled into believing it is sufficient to present an agent with a logical representation, and as a result the agent will do some theory proving for them.

There are two major problems with this architecture. *The transduction problem*; translating the world into an acceptable description in time for it to be useful. *The representation/reasoning problem*; representing real-time entities and processes that agents can reason with in time for the results to be useful. As Wooldridge and Jennings states, "... most researchers would accept that neither is anywhere near solved."[11]. This has caused researchers to look for different solutions, which led to the emergence of reactive architectures.

As a positive, this architecture is the only one which has the ability to act pro-actively in a goal-directed manner. It may not be reactive, as it does not guarantee any specific response times in real-time environments, but it can be utilized to devise plans that eventually will lead the agent to a goal state. In systems requiring planning, hybrid architectures have proved a better choice.

3.1.2 Reactive Architecture

The problems encountered by utilizing deliberative agents have caused questions regarding the viability of the symbolic AI paradigm, and led to the development of reactive architectures. Reactive agents do not have a symbolic model of the world. Decisions are made at run-time based on a limited amount of information and set condition-action rules. Decision-making is solely based on the percepts received from sensors. The idea is that robust behavior should be more important that optimal behavior. One of the largest critics of symbolic AI was Rodney Brooks. In his research he presented two ideas:

1. Situatedness and embodiment: Theorem provers and expert systems do not provide real intelligence, this is situated in the world.
2. Intelligence and emergence: We get intelligent behavior as a result of the interactions between an agent and its environment. Intelligence is subjective; it can not be defined as an innate, isolated property.

Reactive agents can provide a response time guarantee, and are highly reactive as real-time systems have a need for. There have been many successful implementations of reactive agents, but these architectures are restrictive as they can not do complex tasks which require means-end reasoning or cooperation. Also, there is a lack of a clear methodology in how to build such systems.

3.1.3 Hybrid Architecture

The deliberative and the reactive architectures have many shortcomings, some researchers have suggested that neither of them are applicable. Completely deliberate agents have intractable general-purpose reasoning mechanisms, which are not reactive and therefore not suitable for systems with real-time requirements. Completely reactive systems, on the other hand, have a limited scope because they can not implement complex goal-directed behavior.

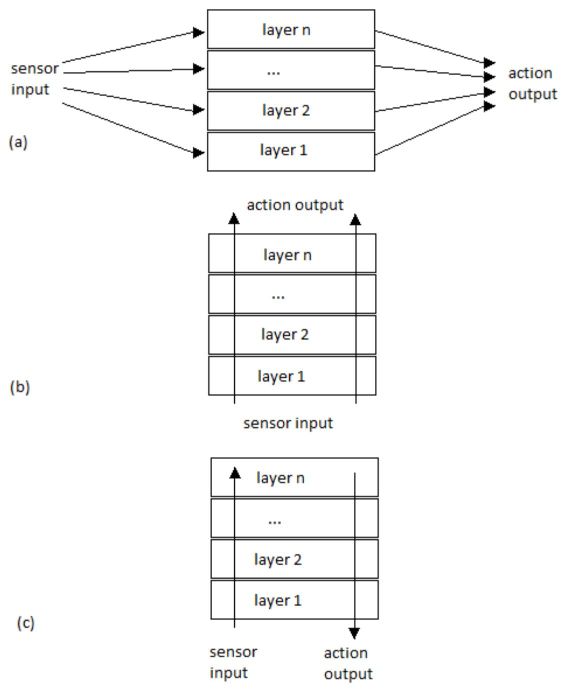

Figure 3.1: Information and control flow in three different types of a layered agent architecture (inspired by [13])

A solution to this is a hybrid architecture also known as a layered architecture, which combines features from deliberative and reactive architectures. The agent's functionalities may be structured into multiple layers in a hierarchy. These layers can interact with each other for behavioral purposes. Layers with deliberative features will contain symbolic representations of information, thus they will make decisions and perform planning. Other layers may have reactive requirements, they skip the complex reasoning and react rapidly to critical events. The reactive part of the system often takes precedence over the deliberative part.

This layered architecture supports reactivity, deliberation, cooperation and adaptability. Different layers may run in parallell. This increases computational capability, but more importantly a reactive layer can monitor for events while a deliberative layer is planning.

With regard to the layering, information and control flows can either be horizontally or vertically layered. *Horizontal layering*, as in Figure 3.1(a), is structured in a way so that each layer is connected to both input and output. The layer itself acts as an agent, deliberating upon which action to choose. However, to ensure that the overall behavior is coherent there is a need of a mediator function. If there are n layers, each with m possible actions, this gives m^n layer interactions that must be considered by the mediator. This will be a bottleneck in decision-making and will certainly result in a poor performance as well. In *vertical layering* the input and output are dealt with by at most one layer at a time. In one-pass architectures, as showed in Figure 3.1(b), the control flow goes upwards through the layers, the final layer generating an output. In two-pass architectures, as in Figure 3.1(c), the information flows up to the highest layer, and the resulting commands flow down again. This approach is not fault-tolerant, failures in a layer has serious consequences for the performance of the whole agent.

3.2 Classification

As Müller[10] suggests, we should elaborate on which agent architecture is appropriate for our domain and present software application. We will use this work as a tool for discussing the paradigms. For a classification to be acceptable, it is suggested to find answers to the following questions.

1. Of which material state is the agent?
   (a) *Hardware agent(s)*: interact with a physical environment, including some internal software components.
   (b) *Software agent(s)*: programs that interact with a software environment.
2. Establish the mode of interaction between an agent and its environment.
   (a) *Autonomous agent(s)*: the system only considers the agent and its environment.
   (b) *Multiagent(s)*: agents use knowledge in order to coordinate actions and collaborate towards a goal.
   (c) *Assistant agent(s)*: these agents mostly interact with and on behalf of humans.

These two questions must be answered, as this will give us an agent taxonomy. We need to precisely define which properties our agent is supposed to have. This can help us in deciding which architecture to choose.

Question 1 is quite simple; does there exist a physical environment with which our agent interacts? CommonSens has hardware sensors, and should implement actuators when including actions in the future, so the answer is that our agent is a hardware agent. Software agents interact within a software environment, thus they never detect physical events nor do they interact with the real world.

In question 2 we need to determine how the agent should interact with its environment. This property will depend heavily on how complex we want the system-to-be to become. Multiagents and assistant agents as described in 2(a) and (b) will use knowledge, and as multiagents both will need communication skills. As we will thoroughly explain in Chapter 4, CommonSens is best served with the property of a single agent environment, the system as a whole will be viewed upon as an agent in itself. Autonomous agent is the best mode of interaction, since our primary focus will be on the system and its environment. This may need to be changed over time, if the desired properties of the agent(s) change.

We have now concluded that our architecture will be based on an autonomous hardware agent. Most autonomous control systems are placed in this category. Autonomous control systems should be self-managing in all possible situations, which is a requirement they share with CommonSens. Often, autonomous hardware agents are best suited with a hybrid architecture. When reviewing Section 3.1.3, we decided to choose this approach for CommonSens. The reactive layer should deal with critical events, for instance the monitored person falling, which require a real-time reaction. Atomic actions and planning (complex actions) should be represented as deliberate layers, including a symbolic representation of the environment, plans, goals and action schemas.

Before going into more details of our hybrid architecture, we need to state the importance of a clear methodology in the architecture design.

3.3 An Agent-Oriented Methodology

It is very important to look at the methodological differences between traditional development and agent-oriented development. Agent-based development of a system brings forward an approach that is very different than the conventional developing practices. The methodology should therefore try to incorporate these different techniques, while maintaining some of the traditional developing features.

The methodologies analyze the methods and phases of agent architectures, and we will in this section focus on methodologies for hybrid architectures. Three examples from Wooldridge[13] will be enhanced here. Keep in mind that the agent is an autonomous hardware agent. The three hybrid architectures are Innes Ferguson's TouringMachines, Jörg Müller's InterRap and the three-tier architecture known as 3T[1].

TOURINGMACHINES

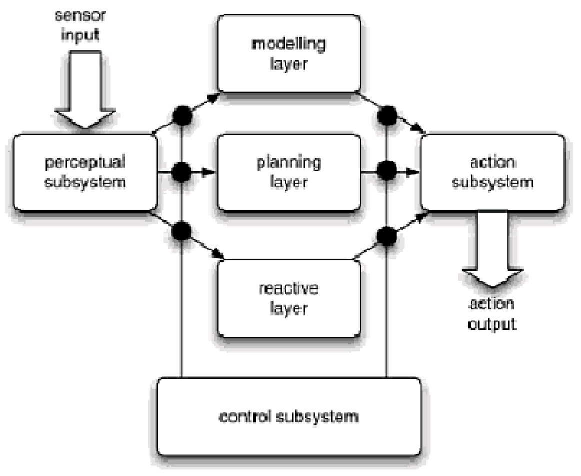

Figure 3.2: The TouringMachines architecture [13]

TouringMachines has a horizontally layered architecture, consisting of three activity-producing layers. These are shown in Figure 3.2. Activity-producing

[1]Since the author did not explicitly reference the name of the developer(s), I assume that 3T is a term for a known type of three-tier architecture. After some researching, the developers of 3T might be the authors of Bonasso et al.[14] Again, this is not certain to my understanding.

signifies that each layer produces suggestions for which action that should be performed by the agent.

The *reactive* layer provides an immediate response to changes in the environment, using situation-action rules. These rules will directly map sensor input to actuator output. The *planning* layer employs a library of plans called schemas, and elaborates these schemas at run-time until it finds one which matches the goal. Each schema contains subgoals, which the planning layer must match with other schemas in the library. Finally, the *modelling* layer has information about all agents in the environment, and can predict conflicts and generate new goals which can resolve these conflicts. These goals will be forwarded to the planning layer.

The decision of which layer should have control over the agent is made by the *control subsystem*. It is implemented as a set of control rules.

InteRRaP

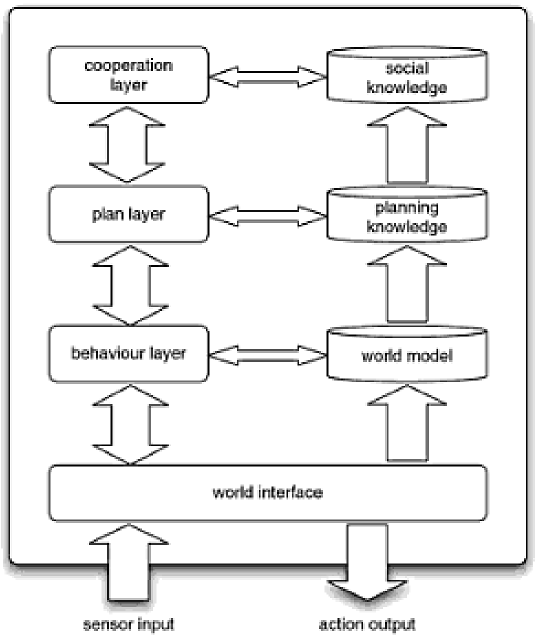

Figure 3.3: The InteRRaP architecture [13]

InteRRaP is a vertically layered two-pass agent architecture. As we see in Figure 3.3, it consists of three layers. The lowest, *behavior-based*, layer has a reactive behavior. Moreover, the middle, *local planning*, layer performs goal-driven planning for the agent. Finally, the uppermost, *cooperative planning*, layer deals with interactions between agents. Each layer has a dedicated *knowledge base*, a model of the world which is appropriate for the layer. This knowledge represents the agent at different layers of abstraction; high-level for plans and action of other agents, middle-level for plans and actions of the agent itself, and low-level for detailed information about the environment.

Interactions between layers are either *bottom-up activation* or *top-down execution*. When a layer is incompetent in dealing with a situation, it passes control over to a higher layer with bottom-up activation. If a layer can deal with the situation itself, there is no need to do this. Also, when a layer wishes to make use of functionalities in a lower layer in order to achieve a goal, top down execution is used.

3T

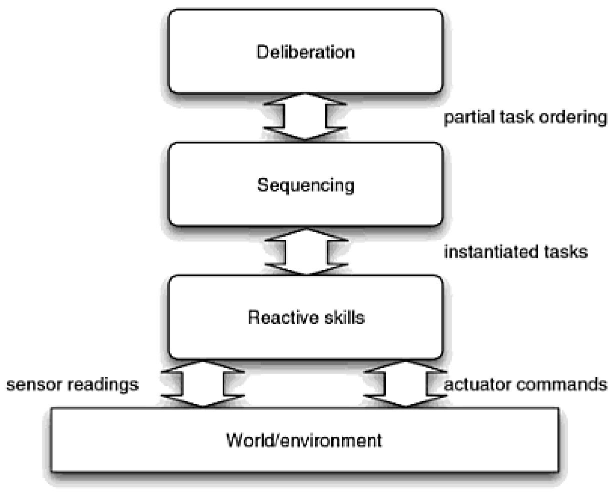

Figure 3.4: The 3T architecture [13]

3T is also a three-level agent architecture, as shown in Figure 3.4. One of the main functionalities of 3T is the usage of skills. A skill is much like we envision actions, it is a primitive behavior that defines a certain ability.

The *reactive skills* layer contains a set of skills, of which one will eventually be executed. Each skill is selected and instantiated by the *sequencing* layer. Deliberation and planning is performed in the *planning* layer.

3.4 CommonSens Architecture

In Section 2.2 we illustrated the relation of events and queries in Figure 2.1. These conceptual ideas will be included in our new architecture, which has relations between events and queries as well as actions and action schemas. This is given in Figure 3.5. We propose such an architecture to ensure that we maintain our philosophy regarding the fact that events and actions are considered dual terms.

Events are predefined situations in the environment which need attention from the system. These are described using queries and detected by sensors. The sensors have a sensing unit which can detect the states in the environment. Actions will also be predefined, and represent something that will change the environment. Action schemas describe the actions, and they are executed by actuators. These actuators have an internal motor which can change the states in the environment. So, events are used for detecting states and actions are used for changing states.

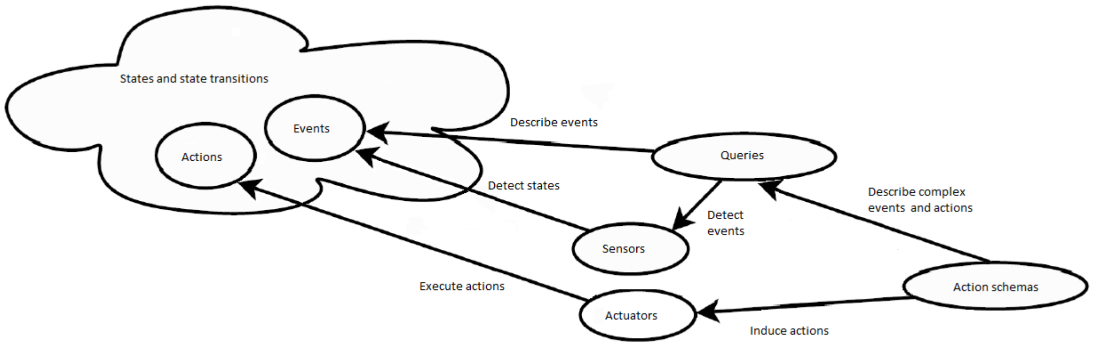

Figure 3.5: The relation of events, queries, actions and action schemas in CommonSens [4]

Note that we will not describe the agent architecture at this point, but the relations between the environment states, hardware and knowledge in CommonSens. We begin with a description of the states of the environment. An event is a set of states or state transitions which are detected in the environment. An action results in a change in a set of states. Events are thus passive, and actions are active entities. Also, the hardware sensors and actuators are also interconnected, as they perform consequently the passive and active behavior. The sensors detect the events in the environment, whilst the actuators execute the actions upon the environment. Finally, knowledge is needed in order for the hardware entities to perform their behavior. The queries describe which sets of states and state transitions should be identified as events. On the other hand, the action schemas describe the sets of actions, with resulting changes in the states. This knowledge should be predefined, and added to the system in the a priori phase. This will be discussed in Section 5.7.

The duality of actions and events is not an easy statement to prove. As we see in Figure 3.5, both describe something in the environment. They are only similar in context after the information, from the queries or action schemas, are translated to low-level data. This low-level data is the data tuples received from the sensors or sent to the actuators for execution. The data tuples for actions should have a similar structure as we saw in Chapter 2 for events.

(timestamp, actuator, capability, value, t_s, t_e)

The temperature example could give the following tuple (173000, $\text{Oven}_0 1$, RaiseTemperature, 22, 181000, 184000). The first attribute is the timestamp of when the action is generated. The system should be able to derive the specific actuator needed for the task. Then follows the capability and resulting value of the action, and finally t_s and t_e, which represent when the action should be executed.

We conclude this section with the fact that events, sensors and queries have common behaviors with actions, actuators and action schemas. The purpose of events and actions are different, events are used for monitoring reasons and actions for supportive reasons. Nevertheless, the structure is quite similar, and this indicates that events and actions may be implemented in a somewhat similar way. As long as the action schemas are translated into readable commands for the actuators, the structure of their low-level data could be similar to the sensor data.

Chapter 4

Analysis

This chapter will provide an analysis of the intelligent agent paradigm with regard to CommonSens. We need to make some choices regarding environmental properties, language and complexity of the system-to-be. In Chapter 3 we discussed the different agent architectures in a high-level manner. Before proposing our own design, we need to specify how intelligent agents and planning can be used in practice. We will now go into the details of intelligent agents, using the earlier research of agents from Chapter 2.

"An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators." [7]

If CommonSens should be extended to include actuators in its implementation, the system should benefit from using intelligent agents as a concept. This will provide many possibilities for expanding and improving the system in the future. As an example of a possible future extension, learning agents can collect and store their inputs as knowledge and learn which new behavior is appropriate using suitable performance measures. An agent's behavior is described by an agent function which maps any given inputs to an action. This abstract agent function is implemented by an agent program.

"In which we see how an agent can take advantage of the structure of a problem to construct complex plans of action." [7]

Planning in AI is defined as devising a plan, which is a list of actions, that can achieve the goals of an agent. Thus, planning will deliberate and produce complex actions. A plan is a solution to a search problem, solving the problem of reaching the goal state of the agent using the available action schemas. Also, there may be several plans which lead to the same goals, performance metrics could describe how to choose between them. Alternatively, these plans could be inserted in the system a priori as queries are already.

In Section 4.1 we will choose the appropriate environment properties for the system. We proceed in Section 4.2 with a discussion regarding the language of the agents. Section 4.3 will explain how planning can be used as a tool for goal-directed behavior. Throughout this chapter we will use a simple example of how such a system could work. Shakey is a simple robot which can execute actions leading up to a goal. We will use Shakey as a very simple example to show how actions and planning can be done in practice.

4.1 Environment properties

Choosing the correct environment properties is essential in creating an agent architecture. The system's complexity increases with the complexity and size of the environment. We therefore need to choose these properties wisely, so that our environment is comprehensible during execution. These properties will also affect how the system performs planning. We will go through the seven main properties of the environment in CommonSens, and explain why we choose the way we do.

It is important to first discuss if we require a multiagent or a single agent environment. A single agent approach at first seemed as an unappropriate aim for our environment, because we have multiple different actuators which should perform different actions of varying complexity. Nevertheless, we discovered in Section 3.2 that our agent should be autonomous, thus it will act as one agent. This means that we will concentrate on only the agent and its environment. We therefore choose a **single-agent** environment. If we added agents for every actuator, there would be need of an immense amount of communication, and we have concluded that this could be a huge bottleneck for performance. The functionality of multiagents is simply not worth that unless there is a way of optimizing the communication between the agents. There exist research on both multiagent[15] and single agent[16] environments in systems similar to ours. Our goal is to have all agent functionality in one place, so that the communication will not intrude on the normal behavior of the system.

Our environment is assumed to be **fully observable**. We assume this because the application programmer should place appropriate sensors for all aspects of the environment that he or she considers important to monitor. Thus, when needed, we can at all times retrieve information from all these aspects of the environment. Realistically, the cameras must be excluded from this assumption because cameras always have blind spots. If the monitored person falls in such a spot, this should be detected in other ways. There could perhaps be a query noticing the persons inactivity when in a blind spot.

The environment is assumed **deterministic**. This is true for a state which is on or off, because if the current state is off and we perform an action, TurnOn, we will be able to determine that the state of this object will be on after the action has been executed. For temperatures, the effects of the action needs some time. If we turn the temperature up 5 degrees, it will not register in the room a few seconds later. Therefore, the temperature monitors are actually stochastic. Nevertheless, we will assume a deterministic environment as a basis to build upon.

It is also assumed to be **episodic**. As an example, if the temperature monitor detects a low temperature, the agent should not consider what the temperature was several hours ago. This might not be the case with the cameras monitoring a person. If the monitored person went to bed it should not set off an alarm because there is no movement in the environment. The agent program needs to keep track of the monitoring history, and thence conclude that the person is sleeping. This is also a special case, and we will again assume the whole environment as episodic.

Furthermore, this is a **dynamic** environment, since the environment can change its state during the time when the agent deliberates. This must be taken into account when actions are executed. A simple example here is the electric oven, which can only be on or off. If it has been on for too long, and the agent concludes that an action should be invoked, the monitored person might turn it off during the deliberation. In the worst case scenario, the action will reverse the monitored person's behavior. We will not assume a static environment, since the fact that this environment is dynamic is undeniable. Classical planning, which we will use later in this chapter, requires a static environment.

The environment is **discrete**, each state has a finite range of values. This applies to the temperature monitor which has values as in a thermometer, and actions as a positive or negative addition to the current value. Cameras have discrete input, but are often viewed as representing continuously varying values.

Finally, it is assumed to be a **known** environment. This is because we assume the agent programmer has a good enough state of knowledge of the laws of physics of the technicalities of the environment. In the temperature monitor example, this will be an assumption that we know whether an action of +2 will result in the current temperature plus two degrees instead of only 2 degrees.

Intelligent agents may be difficult to comprehend when lacking a descriptive example. We need to explain how to construct plans given a specific goal. In order to do that we introduce Shakey, a robot which maneuvers through an environment consisting of objects. Figure 4.1 illustrates Shakey's environment. This example is inspired by a mandatory exercise given in a course in artificial intelligence given at the University of Oslo[1]. The name Shakey most likely originates from the work of Rosen and Nilsson[5], which introduced "Shakey the robot". The exercise presented at the University of Oslo had less complex actions, and a very simple environment. This is why we did not choose to present an example from the origin of Shakey.

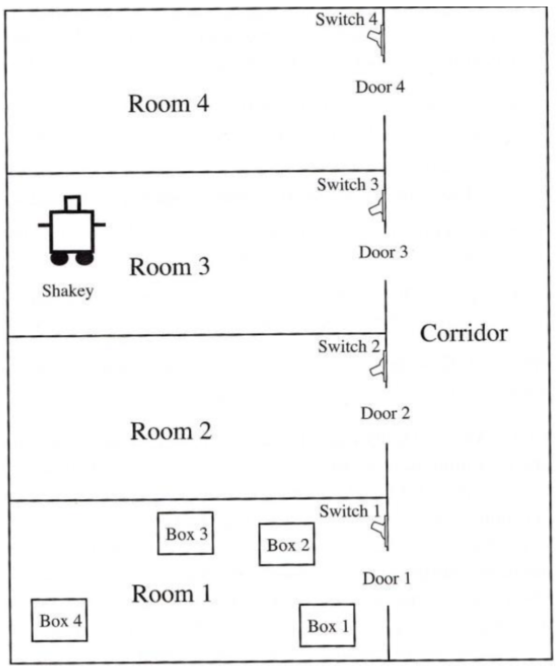

Figure 4.1: Illustration of Shakey and its environment

Shakey has sensors and actuators and can execute actions following a plan.

[1]The exercise is given at `http://www.uio.no/studier/emner/matnat/ifi/INF5390/v12/undervisningsmateriale/ovinger/INF5390-2012%20Exercise%202.pdf`.

Shakey has the following available actions.

GO
Shakey moves from one location to another within a room.
PUSH
Shakey pushes a box from one location to another within a room.
CLIMBUP
Shakey climbs up a box.
CLIMBDOWN
Shakey climbs down from a box.
TURNON
Shakey turns on a light switch.
TURNOFF
Shakey turns off a light switch.

These actions are used by Shakey for it moving around, pushing boxes around, climbing up and down boxes and turning on or off light switches. As we see in Figure 4.1, Shakey has a simple environment consisting of four rooms and a corridor. Each room has a door connecting it to the corridor. There are also objects in the environment, i.e., four boxes and four light switches.

Shakey uses classical planning for creating plans, as we will revisit later in this chapter. Classical planning assumes an environment which is fully observable, deterministic and static. As we have seen, CommonSens deals with a dynamic environment. We will assume a static environment for the purpose of creating a basis for planning, which can be extended to support dynamic environments in the future.

Shakey needs a fully observable environment, meaning everything necessary to detect can be detected through its sensors. We assume Shakey has a camera for observing the environment, wheels for moving around and arms for climbing and pushing. The environment is deterministic because the agent is the only element in it which can alter the states in the environment. This means that all changes in states follow from an action executed by Shakey. No states change during Shakey's deliberation, thus the environment is also static.

4.2 Language

There are many different languages for describing logical sentences. We have first-order logic (FOL), propositional logic and Planning Domain Definition Language (PDDL). When CommonSens was developed, it was implemented with a new language. This language is perhaps not optimal for describing actions and plans in agent programs. The action handling is most easily defined through a common, known language. It may not be optimal to introduce another language to CommonSens, in addition to the existing language it uses. When implementing intelligent agents in CommonSens, one should consider if some aspects in the representation of events should be altered to provide consistency in the system.

As S. J. Russell and P. Norvig[7] suggest, we will use PDDL because Shakey needs classical planning. This because Shakey's environment is fully observable, deterministic and static.

Actions are described by a set of action schemas in PDDL. The functionality these schemas provide can give us a definition of all actions ACTIONS(s) and the result for each action RESULT(s, a) for a problem-solving search with initial state s and action a. In classical planning specifically, the actions are generalized as leaving most of the states unchanged. The actions change the state of the environment using an add list and a delete list.

$RESULT(s, a) = (s - DEL(a)) \cup ADD(a)$

The result of executing action a is discovered starting with the initial state s, deleting all negative fluents DEL(a) and adding all positive fluents ADD(a) in the actions effects.

In Figure 4.2 we see a PDDL representation of Shakey's initial state, as showed in Figure 4.1. Shakey's action schemas are also represented in PDDL in Figure 4.3. The constants used in the initial state and the predicates used in the action schemas are listed in Table 4.1. All actions will now be explained according to the definition of action schemas in PDDL.

The first action GO(x,y,r) takes as input two variables, x and y, true for predicate Location and one, r, true for predicate Room. As a precondition, Shakey must initially be in Room r at Location x. Also, x and y must be in the same Room r. This action will move Shakey from x to y. This must be done by changing a constant in the system representing the location of Shakey.

`Init`($At(LocShakey) \wedge PlacedOn(Ground) \wedge In(LocShakey, Room3) \wedge$
$In(Box1, Room1) \wedge In(Box2, Room1) \wedge In(Box3, Room1) \wedge$
$In(Box4, Room1) \wedge In(Switch1, Room1) \wedge In(Switch2, Room2) \wedge$
$In(Switch3, Room3) \wedge In(Switch4, Room4) \wedge Off(Switch1) \wedge$
$Off(Switch2) \wedge Off(Switch3) \wedge Off(Switch4)$)

Figure 4.2: Initial state in Shakey example.

`Action`($GO(x, y, r)$,
`PRECOND`: $PlacedOn(Ground)$
$\wedge At(x) \wedge In(x,r) \wedge In(y,r) \wedge Location(x) \wedge Location(y) \wedge Room(r)$
`EFFECT`:$\neg At(x) \wedge At(y)$)

`Action`($PUSH(b, x, y, r)$,
`PRECOND`: $PlacedOn(Ground) \wedge At(x) \wedge At(b,x) \wedge In(x,r) \wedge In(y,r) \wedge$
$Location(x) \wedge Location(y) \wedge Room(r)$
`EFFECT`:$\neg At(x) \wedge At(y) \wedge \neg At(b,x) \wedge At(b,y)$)

`Action`($CLIMBUP(b)$,
`PRECOND`: $At(b) \wedge PlacedOn(Ground) \wedge Location(b) \wedge Box(b)$
`EFFECT`:$\neg PlacedOn(Ground) \wedge PlacedOn(b)$)

`Action`($CLIMBDOWN(b)$,
`PRECOND`: $PlacedOn(b) \wedge Box(b)$
`EFFECT`:$\neg PlacedOn(b) \wedge PlacedOn(Ground)$)

`Action`($TURNON(s)$,
`PRECOND`: $At(b, s)$
$\wedge PlacedOn(b) \wedge Off(s) \wedge Box(b) \wedge Switch(s) \wedge Location(b) \wedge Location(s)$
`EFFECT`:$\neg Off(s) \wedge On(s)$)

`Action`($TURNOFF(s)$,
`PRECOND`: $At(b, s)$
$\wedge PlacedOn(b) \wedge On(s) \wedge Box(b) \wedge Switch(s) \wedge Location(b) \wedge Location(s)$
`EFFECT`:$\neg On(s) \wedge Off(s)$)

Figure 4.3: Action schemas in Shakey example.

Furthermore, PUSH(b,x,y,r) has as input one variable, b, true for predicate Box, two, x and y, true for predicate Location and one, r, true for predicate Room. This action shares preconditions with GO(x,y,r), but in addition Location x must be equal to the location of b. This means that Shakey's location and the location of the box must be equal in order for Shakey to push the box. The issue regarding locations will be discussed shortly. The action will move both Shakey and the Box b from x to y.

The third action CLIMBUP(b) has as input one variable, b, true for predicate Box. It has as a precondition that the location of Shakey must be equal to the location of Box b. Also, Shakey must be on the ground. This action moves Shakey from the ground to the top of Box b. Due to this, the two previous actions, GO and PUSH, must have an additional precondition; that Shakey is on the ground.

Similarly, CLIMBDOWN(b) has as input one variable, b, true for predicate Box. As a precondition, Shakey must be on Box b. Common logic shows that Shakey's location must be equal to b, because otherwise Shakey could not be on Box b. In other words, the precondition we have gives us this information. This action moves Shakey from Box b to the ground. The Locations before and after the action is executed, x and y, are assumed to remain equal even though this may not be true in a real environment.

The next action, TURNON(s) has as input one variable, l, true for predicate Switch. It has multiple preconditions. Shakey must be on Box b. The location of Box b, let us call it x, must be equal to the location of Switch s, which we will call y. These variables, x and y, should be accessible through constants along with the question of Shakey's placement on the ground or a Box b. Additionally, Switch s must be off. The action turns on Switch s.

Finally, TURNOFF (s) has as input one variable, s, true for predicate Switch. Its preconditions is similar to TURNON (s), but Switch s must be on. The action turns off Switch s.

We have now analyzed the preconditions of the actions, but there are also some environmental conditions for Shakey. To turn on of off a light switch, Shakey must be in the same room and also stand on a box. Shakey must move a box to the same location as the light switch and then climb it. When Shakey has climbed a box, it cannot move before it has climbed down from the box. Furthermore, in order for Shakey to switch rooms it must first go to the location of a door, which is assumed to be located in both rooms with which it is connected to. These facts are based on reasoning, and we need this information to be implemented correctly in order to get the correct results.

Constants	Predicates
Switch1, Switch2, Switch3, Switch4	Switch, Location, In, At, On and Off
Room1, Room2, Room3, Room4, Corridor	Room and In
Door1, Door2, Door3, Door4	Door, Location and In
Box1, Box2, Box3, Box4	Box, Location, In, At and PlacedOn
LocShakey	Location, In and At
Ground	PlacedOn

Table 4.1: Constants and predicates in Shakey's environment

All the predicate states are ground, non-dividable and functionless. The two assumptions from database semantics should be mentioned.

- The closed-world assumption regards all states that are not mentioned as false.

- The unique names assumption means that all constants, as for instance Room1 and Room2, are distinct.

We need constants to represent a number of variables true for predicate Location. For action GO, we need Shakey's current location. For action PUSH, we need Shakey's location and the location of a box. We also need these locations for the action CLIMBUP. Also, for the actions TURNON and TURNOFF, we need the location of a box and a light switch. The connection between these constants and the predicate Location is found in Table 4.1. For instance, action CLIMBUP in Figure 4.3 treats b as both a Box(b) and Location(b). In addition to this, we also need constants for the different rooms, boxes, light switches and Shakey's placement, respectively predicates Room, Box, Switch and Placement. The predicates On and Off are needed for the light switches, In for a location in a room and At as a position of an object.

4.3 Introducing Planning

In CommonSens, the queries are instantiated using state machine techniques. If queries and planning should be dual terms, as events and actions, we would need them to behave in a similar way. The fundamental question here is if they really can be treated equally. We could either adapt the state machine principle into our planning component, or reversely change the query component language into PDDL or another subset of FOL. At the moment, we cannot be sure if any of these approaches are feasible. They may need to be treated as two different components using vastly different techniques.

We will show how we can use planning to determine which actions should be executed in order to change the state of the environment. While queries describe events which sensors should detect, action schemas define which actions could be used to change the states and state transitions. Thus, planning deals with intentional changes, each plan acts upon the environment, resulting in a change in the environment. We will in this section discuss planning as a reaction to an event. Additionally, we could also have pre-determined plans which are triggered at a certain time.

Fistly, the system needs to differentiate between expected events and severe events in need of a reaction from the system. This is the first step in Figure 4.4, which shows the general idea of how events and actions can be interconnected in CommonSens. A severe event could for instance be that the monitored person falls to the ground.This will be detected through a query performing fall detection via camera sensors. In this case the system should avoid time-consuming planning. The system should rather quickly set of an alarm directly to the health personnel. Life-threatening situations in the home should be assessed in advance, so the system can pinpoint these severe events. Due to this, there needs to be done a risk assessment a priori. This assessment must define which events fall into the severe category. This is done by calculating a quantitative risk factor for each event provided to the system. With regard to the severe category, false positives are preferred against false negatives, so all possible dangerous situations should be covered.

An example of how some events could be categorized is given in Table 4.2. In this table, the darkest fields represent a severe event and lighter fields show less severe events. The leftmost column shows differences in how critical the event is. As an example, a fairly high temperature is not regarded critical in itself. But some factors may be of importance here. If the temperature is extremely high, this may denote a fire or extreme weather. Also, the timeframe of

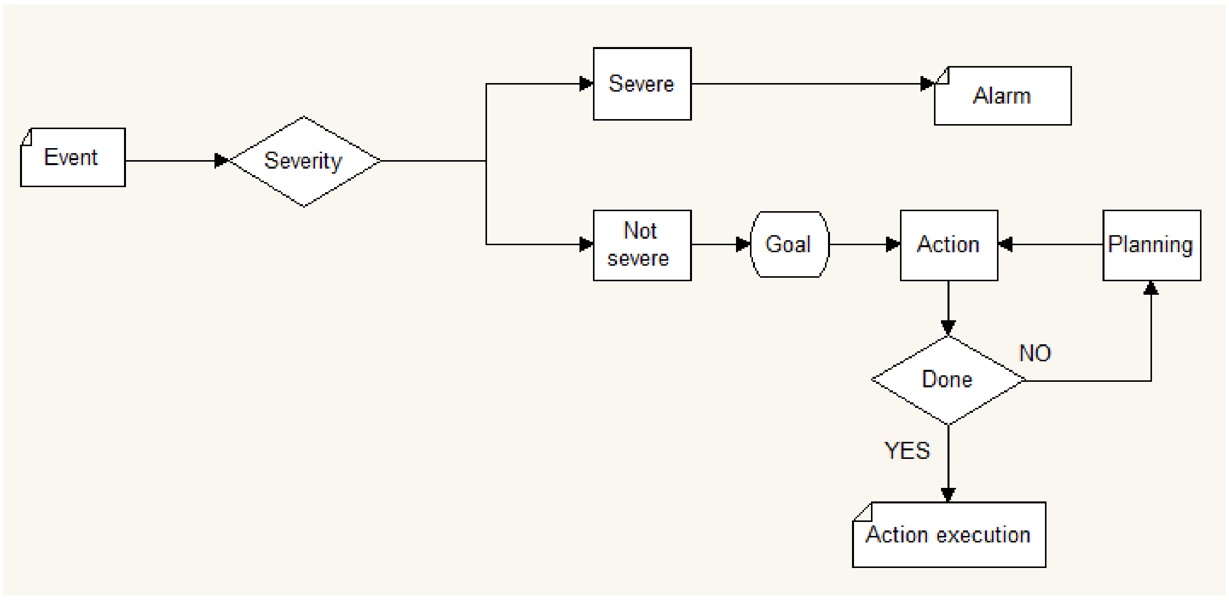

Figure 4.4: Flow diagram showing the connection between an event and corresponding action execution

which the temperature remains high is an indicator of the severity.

As we see in the table, a fall is always regarded severe, the monitored person forgetting their medicine is almost always severe, and the stove staying on for a very long time will also be defined as severe. The table is not accurate, it is only meant as an example of how to categorize the events with regard to their risk factors. We expect the number of severe cases to be minimal. The severe events could simply be defined in a list or table, and a lookup function could be executed to figure out if the current event is severe. This list or table must be provided to the system a priori.

	Fridge	Temperature	Stove	Medicine	Fall detection
Critical	░	▒	▓	▓	▓
Medium	░	░	▒	▓	▓
Normal	░	░	░	▒	▓

Table 4.2: Risk assessment of the events

When the event is not severe, the system can proceed with planning. A goal state is needed first, this is the only thing needed in order to generate a plan. Very few, complex agents can set their own goals, so this must be provided to the system a priori.

When the goal for the current event is found, the system should proceed with finding one or more actions which satisfy this goal. In Figure 4.4, this process is visualized as a recursive planning procedure. When the goal is satisfied, the system proceeds with action execution.

4.3.1 Planning in Shakey

We will continue this section with the Shakey example. Planning in Shakey's environment will provide a good basis for our understanding of how planning works. When a problem occurs we need to define a certain goal statement, describing how we want the environment to be. The goal will list one or several states of the predicates. As an example, a goal could be *On(Switch2)*. To solve this problem, we would need to perform a complex action. The complex action is dependent on the initial state. In order to know which actions should be executed, and in which order, we need to construct a plan which represents the solution to the problem.

Our goal is *On(Switch2)*. We will use the plan generation steps in Figure 4.5. Firstly, we check if the goal is already satisfied, in other words if the goal state is a subset of the initial state. In the initial state, in Figure 4.2, we see that it contains Off(Switch2). Thus, the goal state is not a subset of the initial state. Secondly, we know Shakey needs to eventually perform action TURNON(Switch2), see Figure 4.3. TURNON has On(s) as an effect, thus when called with Switch2 it includes our goal state in its effects. If all preconditions for TURNON would be satisfied, we could execute the action TURNON(Switch2) as an atomic action. If we compare the preconditions of TURNON with Shakey's initial state, we discover that the preconditions are not satisfied. Thus, we know that planning is needed. Thirdly, we call the recursive method createPlan(TURNON).

Planning in the Shakey example is visualized in pseudo code in Figure 4.6. This algorithm will for the action given in its parameters check if all preconditions of this action is satisfied in the world model. The world model will initially be a copy of the initial state. If a precondition does not hold, the method will search through all available action schemas' effects to find one matching this precondition. It will then make a recursive call to itself with this new action as a parameter. When eventually an action has all preconditions satisfied, this action is added to the plan log and the variables its effects change in the world

1. Check if goal is a subset of Init
2. Check all actions if there is one action whose effect contain all states in the goal
   (a) Check if all preconditions are satisfied
3. Call the recursive method createPlan(Action), where Action is the action that most closely fulfills the goal

Figure 4.5: Plan generation steps

```
boolean createPlan(action a)
for all preconditions p in a do
    if p holds then
        break;
    end
    else
        for all actions b in ActionSet do
            if one of b.effect is equal to p then
                result = createPlan(b)
                if result == false then
                    return false;
                end
            end
        end
    end
end
add action to log
change variables according to a.effect
return true
```

Figure 4.6: Pseudo code for plan generation

model will be updated. One of these effects should then satisfy one of the calling method's precondition for its action. When the method is completed, the log will contain the plan in the correct order.

4.3.1.1 Issues

As Figure 4.5 suggests, we need firstly to define the extent of the plan. If we only need to call one action, the problem is not hard to solve. If not, step 3 will call the recursive method proposed in Figure 4.6. The pseudo code may at first seem to solve the issue, but an informed reader might have noticed a problem of redundant actions. If several preconditions share a common action, which effects affect both, the plan will not be minimal nor optimal.

A more severe problem is when Shakey needs a plan which changes rooms. Take the example above, Shakey needs to move from its location in Room3 to the room Box2 is in, Room1. As humans, we know the plan will be {GO(LocShakey, Door3, Room3), GO(Door3, Door1, Corridor), GO(Door1, Box2, Room1)}. But how to implement this? The common denominator is the corridor. All room switches will include this constant. Therefore, we could simply add a new action GOROOM($x_1, d_1, r_1, x_2, d_2, r_2$), which will call GO three times giving {GO(x_1, d_1, r_1), GO(d_1, d_2, C), GO(d_2, x_2, r_2)}. The variables d_1 and d_2 must be of object type Door, and $d_1 \neq$d$_2$. Also, x_1 and d_1 must be located in r_1, and x_2 and d_2 in r_2. Similarly, we can add an action PUSHROOM($b, x_1, d_1, r_1, x_2, d_2, r_2$) for pushing a box to another room.

The plan for the example goal *On(L2)*, is given below.

[GOROOM(LocShakey, Door3, Room3, Box2, Door1, Room1), PUSHROOM(Box2, Box2, Door1, Room1, Switch2, Door2, Room2), CLIMBUP(Box2), TURNON(Switch2)]

In other words, Shakey goes to the Box2 in Room1, pushes Box2 to Switch2 in room Room2, climbs Box2 and turns on Switch2. This is the optimal number and order of actions. To complete TURNON(Switch2), the box must be in the same room. Shakey must first move the box to the correct location before executing CLIMBUP(B2). If Shakey does this action before moving the box, it must do a CLIMBDOWN, PUSH and the CLIMBUP again. That is where the redundant actions comes into play.

Action selection in the pseudo code for createPlan in Figure 4.6 is highly dependent on the order of the preconditions. These preconditions are selected upon installation and must be ordered carefully. We will present an example of what could go wrong, and why we wish to point out the importance of the order of preconditions. In Figure 4.3, we see the order of the preconditions for TURNON. If On(b) was placed before At(b, x), the plan would be as stated in the previous paragraph. Shakey would first go to the box, then climb up the box, climb down, push the box, and finally all preconditions are satisfied even though Shakey is eventually not On(b). Therefore, this must be implemented carefully and monitored closely before action execution occurs.

4.3.2 Discussion

As we have mentioned earlier, the environment in CommonSens is dynamic. Because of this, we can not adopt the techniques from classical planning in CommonSens. Also, we can not ignore that time is an important matter, so planning must be done in an efficient way. There could also be issues when scaling up small planning approaches to real-world sets of events and actions.

Since a system using classical planning will not consider the success of an executed action, we will never know if dynamic changes in the environment during deliberation may have come in conflict with the planned action. In order to avoid this problem, the system must somehow monitor the execution of the planned actions.

If the action execution does not result in the desired effects, the system should initiate a modification in the plan. However, the system needs to find the origin of why the action was unsuccessful; if the monitored person's dynamic behavior resulted in a situation where there is no need of action, this must be taken into account. Otherwise, if it resulted in another event in need of reaction, modifications or replanning is needed. There is some research on dynamic planning, but we choose to propose this as future work.

Ferguson[12] provides an overview of hybrid architectures using dynamic planning. We will briefly present the most interesting architecture in this work. *Dynamic reaction* is a set of techniques for manging the monitoring and acting in dynamic domains. It has been extended to interact with a planning system, the *abstraction-partitioned evaluator* (APE) architecture. APE uses action control layers similar to the two-pass layered architecture. The most interesting feature for our need is APE's *state of affairs* (SOA) structure and *monitor* components. SOA contains an up-to-date world model, along with records of the current goals for the agent. The monitors are information gathering components which will send reports to the SOA model, i.e. about observed events or plan constraint violations. Additionally, the monitors can initiate replanning and take action according to predicted exceptions.

Some of the features from the APE architecture may provide the techniques we need in order to implement dynamic planning. We leave the further research of planning techniques as future work. In this chapter, we chose environment properties and language of our agent. We also provided some insight in intelligent agents and planning with our Shakey example. The next chapter will propose an overall design of the system.

Chapter 5

Design

In the previous chapters, we have discussed different architectures and argued about which properties our agents should have. We decided upon using a hybrid agent-oriented architecture and saw how an implementation of an agent-based system will work using the Shakey example. In this chapter we approach how to integrate parts of the discussed techniques into CommonSens. Our hybrid agent architecture will be explained thoroughly, and a new proposed design will be presented.

In Section 5.1 we present our hybrid architecture, and define its layers. Each layer will represent different components of the action selection process. Section 5.2 will look at the current design models in CommonSens, and propose new models representing the action processing and actuators. Finally, Section 5.3 will propose how the life cycle phases in CommonSens need to be changed. The most important phase is the a priori phase, since introducing the proposed architecture requires more knowledge at configuration time.

5.1 Hybrid Agent Architecture

We have chosen a two-pass vertically layered agent architecture. This entails that the flow of information will surpass all layers until it reaches the upper layer, the response will then flow down to the lowest layer. The structure of the architecture is given in Figure 5.1. We will first explain the process as a whole, before going into the details of each layer.

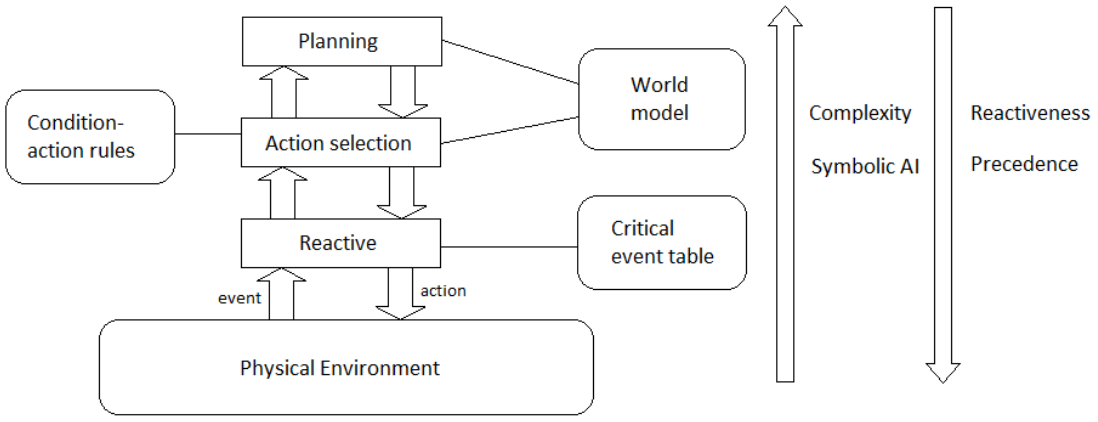

Figure 5.1: Proposed hybrid architecture for action processing in CommonSens.

5.1.1 Functionality

Our proposed architecture divides the deliberative and reactive agent functionality into multiple layers. Since our system should differentiate between critical, atomic and complex actions, we have created a three-layer hierarchy. Each layer represents one of these types of actions. The layers also have available some knowledge. The knowledge each layer has represents the different levels of abstraction in the agent. The reactive layer has a critical event layer, stating which events require an alarm. The action selection layer has condition-action rules, which are used for mapping events with goal actions, and the world model. The planning layer has a planning algorithm and the world model. The world model contains symbolic notions used in planning, as action schemas, goals and the initial state of the world. This initial state must somehow be updated dynamically, since the states may change during the time when the agent deliberates.

As we learned in Section 3.1.3, the reactive layers often take precedence over deliberative layers in hybrid architectures. Our reactive layer has the highest precedence and the higher the layer is in the hierarchy, the lower the precedence is. Thus, if an incoming event requires an alarm, the reactive layer uses its precedence and reacts directly to the event without involving the other layers. This is inspired by the bottom-up activation functionality in InteRRaP, which we presented in Section 3.2. Bottom-up activation is performed if a layer can not deal with the situation, and this is similar to the behavior we wish to achieve. The output needs to signify the action type, in order for action execution to work correctly. We have chosen to embed this functionality into the message using flags.

We need to address the information flow in this proposed architecture. The layers deliver different content in the messages they send, depending upon which layer takes precedence. In Table 5.1, we list the three possible ways action processing may take place. We have decided to use flags to denote which layer takes precedence, making it easier for further action processing and execution. The first flag is called ASL (Action Selection Layer). When ASL is set, the action selection layer does not process the message, thus it has been preceded. The same is true for the PL (Planning Layer) flag. When set, the planning layer does not process the contents of the message. Table 5.1 shows the ASL, PL pairs, these represent which flags are set. When the pair is 00, none of the flags are set, when it is 01, the PL flag is set, and finally when it is 11, both the ASL and PL flags are set.

ASL,PL pair	00	01	11
Flag(s) set by	None	Action selection layer	Reactive layer
Action type	Complex action	Atomic action	Alarm
Flag meaning	All layers process the information	Skips planning layer	Skips action selection and planning layer

Table 5.1: Description of the ASL and PL flags.

If the reactive layer detects a critical event, it sets both the ASL and PL flag, and the reactive layer sends an alarm. The other layers will never be invoked. This is illustrated in Figure 5.2(a). If the event is not an alarm, no flags are set by the reactive layer, and it needs to send the events and flags to the action selection layer, since it is not capable of solving the situation. If the action selection layer then detects an atomic event, it sets the PL flag. The planning layer will then never be invoked, and the action selection layer sends an atomic action and the flags to the reactive layer. This is shown in Figure 5.2(b).

When neither the reactive layer nor the action selection layer can solve the situation, we need planning. The action selection layer should then send the goal action or actions along with the flags, which are not set, to the planning layer. The planning layer runs some planning algorithm which results in a plan.

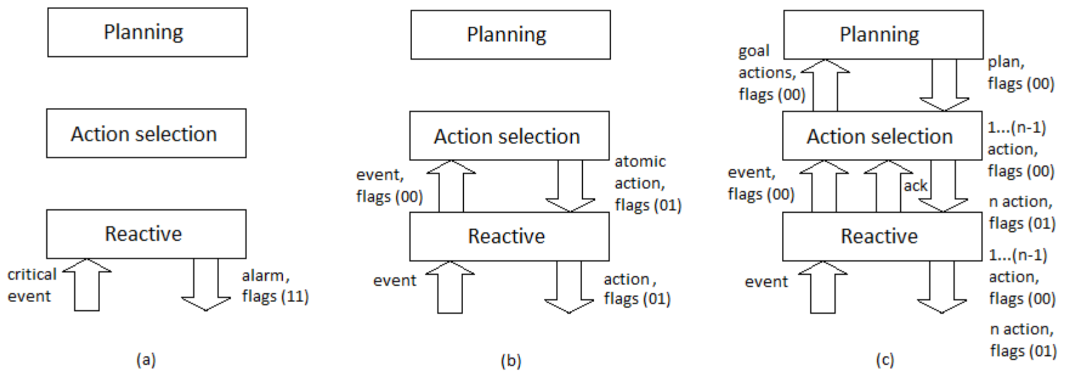

Figure 5.2: Behavior of layers and flags in three different situations: (a) Alarm (b) Atomic action (c) Plan (complex action)

The plan and the flags are sent back. The action selection layer goes through these actions, sending them down one by one. The reactive layer executes each action and, because no flags are set, sends back an acknowledgement to the action selection layer for the first 1 to (n-1) actions. When the action selection layer sends the last action of the plan, it resets the PL flag, denoting an atomic action. The reactive layer will not send back an acknowledgement after sending this action for execution. This process is found in Figure 5.2(c).

5.1.2 The Reactive Layer

The lowest layer, the reactive layer, deals with situations that require a reactive behavior. All events are relayed to the reactive layer from CommonSens, since CommonSens already has the mechanisms for recognising atomic and complex events. The layer will not have any knowledge about the world, it should simply react as quickly as possible to critical events. In CommonSens, the most important performance measure is the health of the monitored person. Because of this, the most severe events are those which are life-threatening; i.e. a fall or a long duration of inactivity. Since there are very few situations in which we need to alarm health personnel, a table-lookup will suffice here. Table 5.2 shows an example of how this table might look. If the event matches any of the critical events in the table, the reactive layer will take control over the action processing itself. Take note that if this list is large, the performance will be poor. We envision this table as a small set of events, because there should not be many situations of this severity.

CriticalEvents
FallDetected
LongInactivity
FireDetected
AlarmActivated
...

Table 5.2: Critical event table in the reactive layer.

Basically, the reactive layer checks if the event is listed in the critical event table, and a match denotes that an alarm is needed. The other layers will not be involved in this situation, and the alarm and flags, both set, will be sent for execution. See Table 5.1 regarding these flags, they implement the precedence of the reactive layer over the other layers. If the event is not severe, the reactive layer passes it on without setting the flags.

When the reactive layer receives an action passed on from higher layers, it firstly checks the flags. If only the PL flag is set, it sends this action along with the flags for execution. The special case concerning flags is when no flags are set. This implies that the current action is part of a complex action. The reactive layer must in this case inform the above layer when the action is sent for execution, possibly with a timed delay, so it can proceed with the next action of the plan. On the last action received, the above layer sets the PL flag, and execution proceeds as if it was an atomic action.

5.1.3 The Action Selection Layer

The middle layer, the action selection layer, deals with situations that require a deliberative agent. It has access to an internal model of the world, and can respond to atomic actions. It also needs to handle plan execution, as plans can be sent from the layer above for execution. Actions we could encounter here are for instance turning on or off a light switch, increasing the temperature and other similar tasks. This layer uses condition-action rules much like the simple-reflex agent in Section 2.3.2. An example of the events and resulting goal actions in the condition-action rules is shown in Table 5.3.

The behavior of the layer when receiving an event is to do a rule check for the event, using the condition-action rules. This should yield one or more goal actions per event. If there are several actions, the action selection layer imme-

Event	**Goal actions**
LightOnAtNight	{TurnOffLight}
OvenOnForTooLong	{TurnOffOven}
InactivityDetected	{TurnOnAlarmClock, ShowWarningOnMonitor}
...	...

Table 5.3: Condition-action rules in the action selection layer.

diately sends these to the above layer. Otherwise, it checks the preconditions of the action. Theses are found in the action schemas in the world model. Figure 5.3 is an incomplete example of how the world model is stuctured. If the preconditions are not satisfied in the initial state of the model, it sends the action to the planning layer. If all preconditions are satisfied, the layer sets the PL flag denoting an atomic action, and send this down to the reactvie layer. The messages that are sent downward may have a similar purpose as InteRRaP's top-down execution.

When the action selection layer receives a message from the above layer, it checks the flags. If both flags are set, it sends the control to the below layer. If the PL flag is set, it sends the action stored earlier to the below layer. If no flags are set, the layer has received a plan from the above layer. This plan must be decomposed into a linked list of some sort, and it must remove and send the first action of the list to the below layer. When the last action is removed, the action selection layer must set the PL flag, which implies an atomic action.

5.1.4 The Planning Layer

The planning layer deals with only the situations which require planning. The layer has access to the internal model of the world as well as a planning algorithm for plan creation. The planning algorithm should somehow support an interrupt function, which interrupts the deliberation process when a new critical event has occurred. As we discussed in Section 4.3.2, there is a need of reactive planning in CommonSens. How this should be implemented has not been discussed in this thesis, so Figure 5.3 is an incomplete example of how the world model could look like. We have intentionally left the preconditions and effects blank in the action schemas, because these can vary depending on how planning is done. We will not propose a planning algorithm either, because this is based on the planning approach that is chosen. We will simply state how we envision the functionalities of this layer will work.

```
Init(Off(LightBedRoom)
∧Off(LightKitchen) ∧ On(LightLivingRoom) ∧ Off(LightBathRoom) ∧
LocationPerson(LivingRoom) ∧ On(Oven) ∧ Off(AlarmClock) ∧
TempWithinBounds(LivingRoom) ∧ TempTooHigh(Kitchen) ∧
TempWithinBounds(BedRoom) ∧ TempWithinBounds(BathRoom)...)

Goal(TurnOffOven)

Action(TurnOnLight(l),
PRECOND:
EFFECT:)

Action(TurnOffLight(l),
PRECOND:
EFFECT:)

Action(TurnOffAlarmClock(a),
PRECOND:
EFFECT:)

Action(TurnOnAlarmClock(a),
PRECOND:
EFFECT:)

Action(ShowWarningOnMonitor(w),
PRECOND:
EFFECT:)

Action(TurnUpTemp(t),
PRECOND:
EFFECT:)

Action(TurnDownTemp(t),
PRECOND:
EFFECT:)

Action(TurnOffOven(o),
PRECOND:
EFFECT:)
```

Figure 5.3: World model for action selection layer and planning layer.

When this layer receives a message from the layer below, this message will contain one or more goal actions. If there are several actions, the layer checks if both actions' preconditions are satisfied in the initial state. If they are, it should somehow order the actions in a plan, and send this along with the flags to the below layer. None of the flags are set. If one or more of the preconditions are not satisfied, it will use a planning algorithm for solving the problem. This algorithm should, as our algorithm example for Shakey in Section 4.3.1, result in a plan. This plan will also be sent along with the flags.

The world model includes the initial state, a goal state and all the available action schemas. We can see the initial state in our example of a world model in Figure 5.3. If an event occurred because the oven is on, the kitchen light is off and the monitored person is not in the kitchen, a response to this could be two actions. Firstly, a warning could be displayed on the tv monitor. Also, the system could turn the oven off after a certain time if the monitored person did not react to the warning. Plans like this are complex, and the planning algorithm should handle this and many other situations.

5.1.5 Discussion

The hybrid architecture gives our system many good features. The reactivity of the reactive layer provides real-time features for the very important alarms. The agents can also execute complex actions, a functionality we strived for because CommonSens has complex events. The vertically layered structure gives scalability for control messages, so agent performance will not be poor in general. Nevertheless, if a failure occurs in one of the layers, the overall agent performance will suffer because all layers are surpassed at each run in this architecture. The question is if this is something we can fix or live with.

If multiagents will be chosen in the future, the architecture will have need of a communication layer. This layer must keep track of other layers and detect conflicts between itself and other agents. The must also be a unit of control in multiagent systems.

The layers we chose was inspired by the InteRRaP architecture, presented in Section 3.3. The action selection layer is not as deliberative as we would like, since the condition-action rules are somewhat ”hardwired”.

5.2 Action Processing Model

We now proceed with the structural modeling of our agent architecture. In order to have a duality between events and actions, we need the models to be rather similar. In Appendix B, we have the UML models by Søberg[4]. CommonSens has a package called eventProcessor, as seen in Figure B.1, which handles all events with corresponding queries and data tuples. There is also a package handling the connection of events and sensors, the sensing package shown in Figure B.2. We need to understand how these models work in order to create new models for action processing and the actuators.

Figure B.1 shows how the components in event processing is connected. Each complex query is implemented in the query pool by one or more objects of the QueryPoolElement. The classes Box and Transition create a boxes-and-arrows structure. This enhances support for logical operators and provides a simpler evaluation of atomic queries.

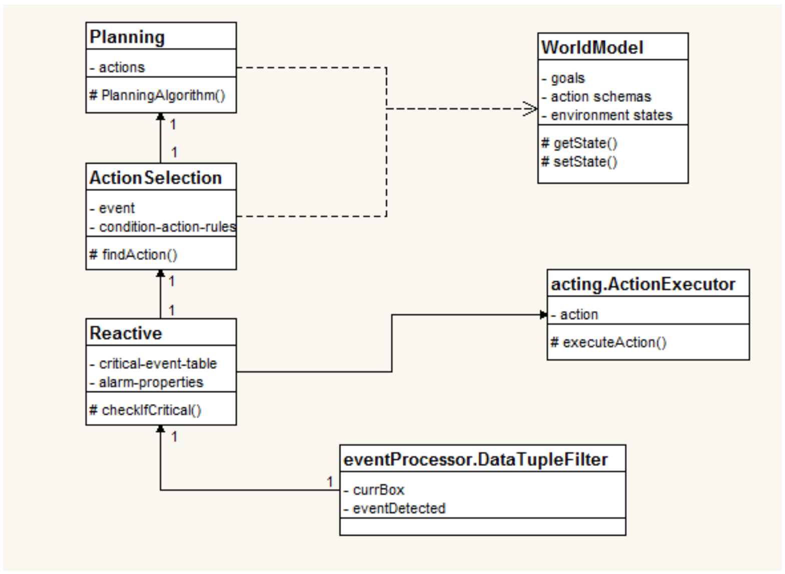

Figure 5.4: Key classes of the actionProcessor package.

The eventProcessor package will not start processing until the system is running, either as a simulation or real-world monitoring. As we see in Figure B.3, the sensors are pulled and tuples received. The system then calls evaluateBatch(), which returns a match if the tuples match the conditions in the queries. This information is then returned to DataTupleFilter. Thus, DataTupleFilter is a good starting point for the action processing. The idea is that this class will create a new thread which will start the action processing when needed.

As a corresponding package, we introduce the actionProcessor as showed in Figure 5.4. It covers the hybrid agent architecture we discussed in Section 5.1, with the layers Reactive, ActionSelection and Planning. All the layers have internal knowledge, but the action selection layer and planning layer also uses a model of the world, WorldModel. In Figure 5.1, the reactive layer was connected to the environment. This was a way of illustrating how the architecture should work. In fact, the reactive layer will receive events from the DataTuple-Filter. Additionally, we have a class ActionExecutor which is responsible for the actual execution.

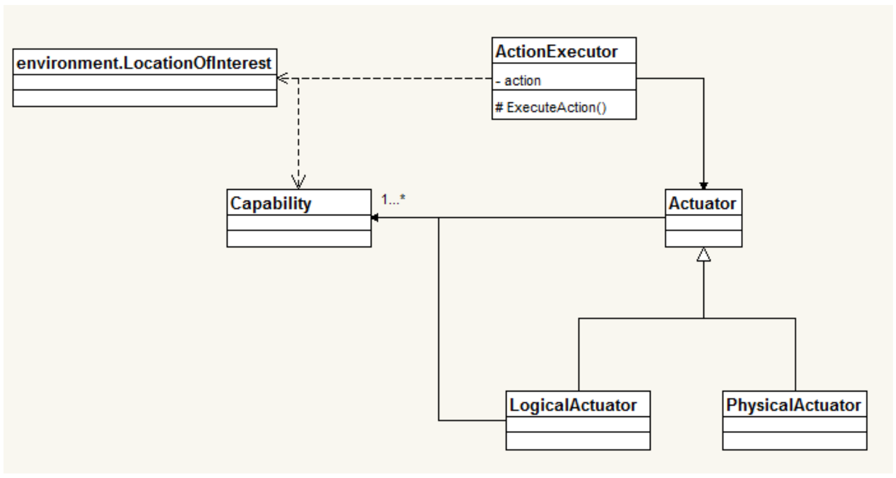

Figure 5.5: Key classes of the acting package.

The next package, acting, resembles the sensing package. Similarly to sensing, which implements sensors and their capabilities used by the events, acting implements actuators and their capabilities. In the sensor model, a capability is a description of all things a sensor can observe. For actions, a capability should describe all thing an actuator can do in the environment. The queries never address sensors directly, they address the capabilities the sensors provide. This should also be the case for actions, these should address an actuators capabilities. Nevertheless, the capabilities need to be differentiated, there should be one set of capabilities for the sensors and another for the actuators. This because no sensor and actuator can provide the same capability.

Figure 5.6 shows the control flow when an event is detected. The data tu-

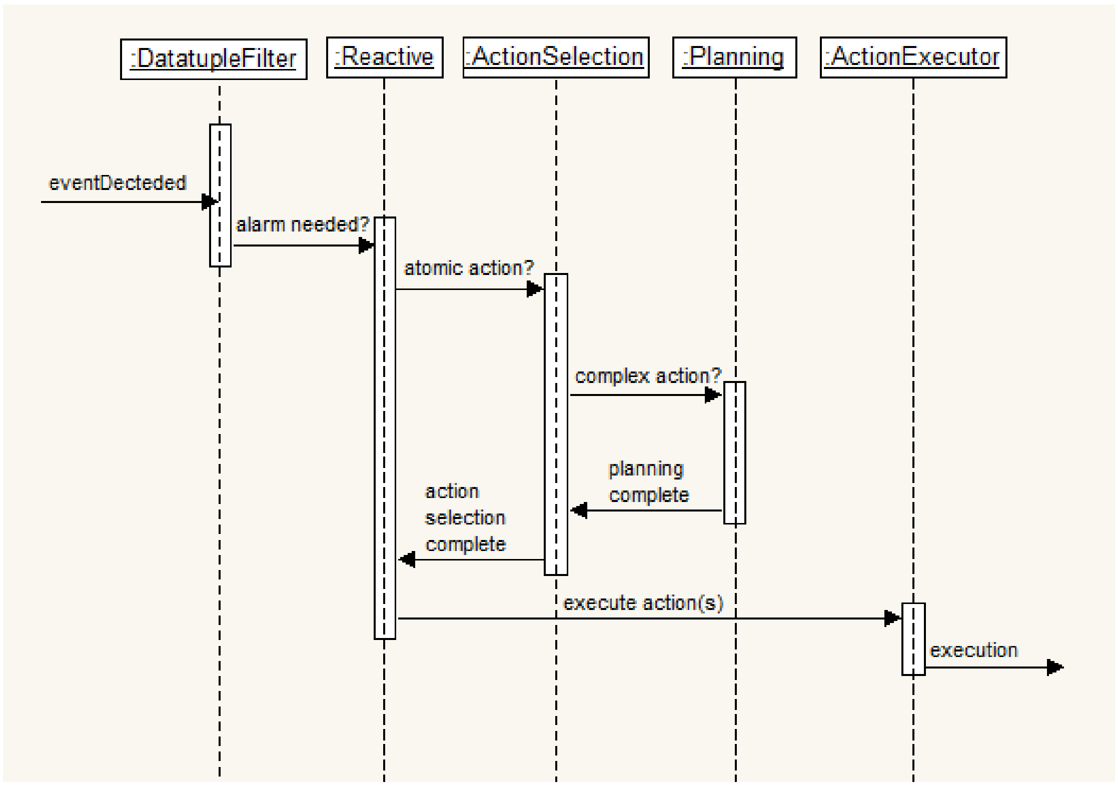

Figure 5.6: Sequence diagram of the control flow when an event is detected.

ple filter will call the reactive layer if the event has occurred, in other words if the tuples matched the conditions in the query. The reactive layer checks if the event requires an alarm, the action selection layer checks if we need an atomic action, and the planning layer checks if we need a complex action. The resulting action or actions will be sent from the reactive layer to the action executor, which should find the corresponding capability for action execution.

5.3 Life Cycle Phases

We continue with a discussion regarding the different life cycle phases of the system. CommonSens already has defined how the life cycle phases should be. This is shown in Figure B.4. When introducing actions and planning, we have throughout this thesis noticed that prior knowledge is needed in order for the agents to do reasoning when performing action selection. As we mentioned in Section 5.1, the layers in the architecture need a critical event table, condition-action rules, action schemas and an internal world model. This information needs to be set by an application programmer a priori, in order for the proposed architecture to work.

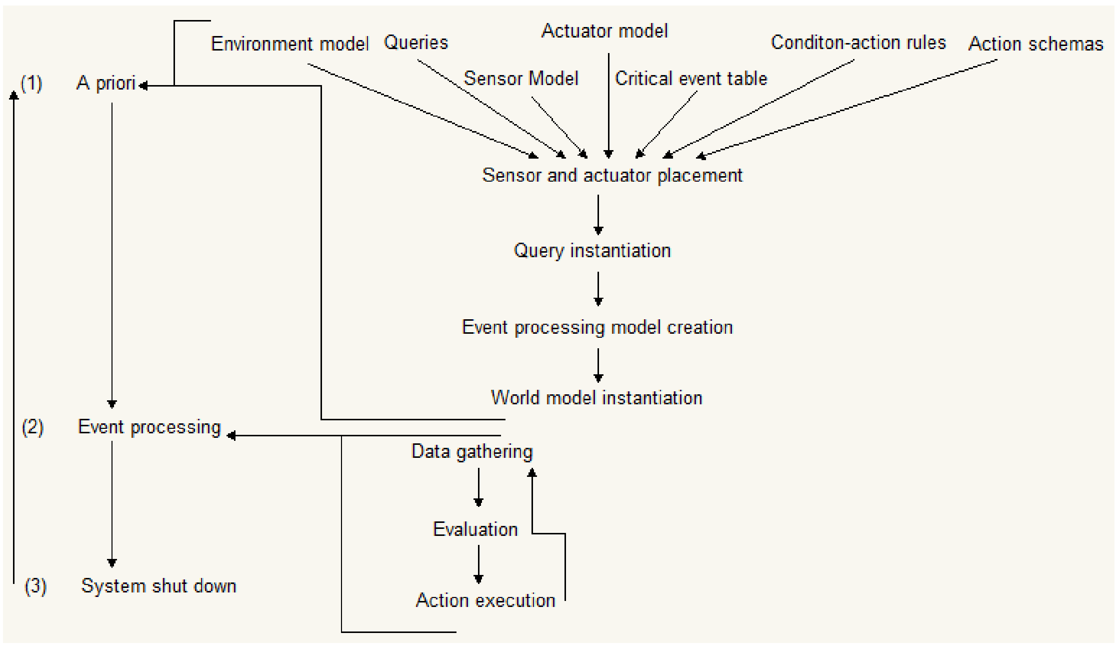

Figure 5.7: Life cycle phases in CommonSens.

In Figure 5.7 we present the new life cycle phases in CommonSens. This figure is inspired by Figure B.4. In addition to details regarding queries and sensors, we need knowledge about critical events, condition-action rules and action schemas. This information must be added off-line, so if new actions should be introduced to the system it must be reconfigured. Both sensors and actuators must be placed as seen fit. The final change in the a priori phase is that the world model should be instantiated. This model will be updated statically each time an action is needed. The event processing phase is expanded with action execution after evaluation.

Chapter 6

Conclusion and Future Work

In this last chapter we will provide a summary of the work in this thesis and its contributions. We will also discuss the possible solutions and improvements on the observed problems and flaws we encountered. Finally, we look at how our work may be continued, and the possibilities available for future work.

Many decisions had to be made with regard to agent properties and architecture. We wanted to present the fundamental principles.To have a manageable starting point we chose as simple properties as possible, that met our reqirements. The idea was that the resulting design of the architecture would demonstrate proof-of-concept, as a first step towards an adequate architecture.

The first task was to choose a methodology for implementing actions in CommonSens. Similar systems encountered used intelligent agents for action processing. Some basic knowledge in this field was provided in the course INF5390 about artificial intelligence at the University of Oslo. Since there, to our understanding, does not exist an adequate alternative to intelligent agents for the purpose of action processing, we researched the possibilities they provide.

We needed to research the background material for CommonSens as well. In order to find out how we can integrate actions in the system, we needed to understand the functionalities of events, the counterpart of actions. Our main material for details on intelligent agents was provided by the textbook in the INF5390 course. As we dug deeper into the subject, the focus of an agent architecture appeared. Many authors in AI had published work related to the choice of an agent architecture. The possibilities and concerns were many. We realized that an agent architecture classification is an important first step in the development of intelligent systems. We found a step-by-step classification on how we can choose an agent architecture. This, along with our knowledge about CommonSens, helped in choosing a hybrid agent architecture. This decision narrowed down the amount of possible designs. We found inspiration in three of these, TouringMachines, InteRRaP and 3T.

We concluded that some of the elements in the design of the event model in CommonSens could be extended with properties of actions. Events and actions represent a change in the environment, sensors and actuators are the hardware entities which can sense or perform these changes, and queries and action schemas represent the knowledge needed. With regard to the environment properties, most of these could easily be chosen due to the CommonSens environment. The simple structure of the environment in CommonSens gives us a fully observable, deterministic, episodic, dynamic, discrete and know environment. An exception was the choice of a multiagent vs. a single agent environment. A multiagent environment consists of multiple agents which are either cooperating or competing against each other. This entails that these agents need communication functionalities. Agent communication can be difficult to implement and real-time messages between agents may result in poor overall performance. Most importantly, this is a complex design problem, since they need to cooperate in a correct manner. In multiagent systems there may occur conflicts, both in goals and actions. Because of this there is need of a control unit, which decides which agent should have control over the other. Single agent environments concentrate all agent functionality into one single entity. All decisions are then centralized, thus overall action control is ensured. Because of this, we chose to use the single agent approach. The choice of the easiest solution will also give us an opportunity to get some experience in the behavior of the system.

In order to help the reader to understand the core concepts of intelligent agents, we introduced the example Shakey. When introduced, we may have underestimated the number of issues that would arise when this example was used to visualize planning. We chose to include the example regardless of this, because it illustrates all the elements needed in planning; action schemas, initial state, goals and planning algorithm.

Another fundamental decision we had to make was the choice of language used to describe our model of the world. Actions are often described using some logic, most commonly propositional logic or first-order logic (FOL). Since our Shakey example used classical planning, we chose to use a subset of FOL, the planning domain definition language (PDDL). Ideally, to best preserve the duality between events and actions, we would like both to be described using the same language. Since the language CommonSens uses for its events is not a commonly used language, it was not clear to us how it can be extended to define actions. This language was developed along with CommonSens, as a completely new language tailored to the needs of the system. We would have to do extensive research and testing in order to figure out how this language could be used for actions. This was not done due to a limited amount of time. PDDL is a well known language for intelligent agents, it can also be applied in planning, so we chose to use this as a basis for describing agent functionality.

Planning was only described in this thesis in the Shakey example. This example uses classical planning, which we at an early point viewed as a good choice for CommonSens. After some time, however, we realized that classical planning requires a static environment. The environment in CommonSens is dynamic. Thus, classical planning will run as though the states in the environment can not change during the time when the agent deliberates. In order for planning to work in CommonSens, the world model must somehow be updated dynamically, so that any changes during deliberation are taken into account before any action is executed.

We chose to use a hybrid agent architecture, this was based on the fact that CommonSens could make use of both goal-directed planning and reactive behavior. The latter was viewed as a good solution for alarms. The alternatives were a deliberative architecture, where alarms would have to be stated as an action without a real-time guarantee, and a reactive architecture, where we would not have been able to use planning and thus complex actions.

Our problem statement was as follows. *In this thesis we consider how CommonSens could be extended to include the functionality of intelligent agents. Our goal is that the resulting architecture will reflect a duality between events and actions.* We now need to assess if we achieved these goals.

At the beginning of this thesis, our main goal was to introduce actions in CommonSens. Initially, we did not know how to do this, so a lot of work went into actually finding techniques that supported actions and actuators. Intelligent agents were presented to me in the course INF5390. Besides this course, there has not been much research in this field at our University. Therefore, we found that much of the time we had in the beginning were used learning how intelligent agents, and especially their architectures, work in practice.

We have chosen environment properties to our best knowledge, with some assumptions. Since we have not researched the functionality of advanced sensors, as for example camera monitoring, it was difficult to address which environment properties these might affect. Also, when choosing single agents we aimed for a simple structure, there may be advantages in using multiagents because of cooperation in planning. Multiagents would provide support for concurrent and consecutive action processing, which is currently not provided.

The agent architecture was chosen as a basis for the design and structure of the intelligent agent. We chose a hybrid agent architecture, as we think is the best solution for the system. This because alarms are handled in a reactive way by the system. The decision was made due to the classification we made of the agent. We state that we have an autonomous hardware agent. If this should prove to be incorrect, the architecture needs to be changed accordingly. If multiagents should be implemented, there would be need of some functionality which handles agent communication and a control unit must delegate control to these agents. There would probably be many more complications due to this. If the classification or environment properties are incorrect, or advised to be changed after a certain time, the decision of an agent architecture must be revisited.

The duality of events and actions is only partially realized. They both represent the states of the environment, which form current or future situations. In the system, because of the language, the information is not similarly represented. Events are represented using queries, and actions uses action schemas as their knowledge. We could not find any earlier research which had united an existing CEP system with intelligent agents, thus we were uncertain if the whole system should be reprogrammed. Nevertheless, when this information is translated into low-level data, we proposed using similar data tuples. Intelligent agents use capabilities, as the sensors of CommonSens do, and therefore this would not be a problem to implement.

Planning requires much more research before this can be implemented. Looking back, we regret the fact that we did not contribute more on this important matter. We found research on dynamic planning, but due to the short amount of time we had left, we had no choice but to leave this for future work. It is also important to discover how the world state should be updated in a dynamic environment like CommonSens'. There are many pitfalls in planning in general, as we discovered in the simple Shakey example. This example also gave insight into the fact that condition-action rules had to be presented to the system a priori, in order for the system to know the goal actions for each event. We do not propose that this is the most efficient, nor the easiest approach. Perhaps the plans could be constructed a priori, as a way of getting experience with how plans may be remembered by the system and found in a lookup function after they are made. These possibilities, among many others, are also considered future work.

Due to the amount of research needed to get this far, we did not have time to conduct any implementation or testing of our proposed architecture. More knowledge in the implementation of intelligent agents is needed in order for the agent functionalities to work correctly. CommonSens is a complex system and in order for us to be able to test the new functionalities, we need more knowledge on the current possibilities of testing in CommonSens.

Appendix A

First-Order Logic

Symbol	Formal Name	Connective
$\neg$	negation	not ...
$\wedge$	conjuction	... and ...
$\vee$	disjunction	... or ...
$\Rightarrow$	conditional	if ... then ...
$\Leftrightarrow$	biconditional	... if and only if ...
$\forall$	universal quantifier	for all ...
$\exists$	existential quantifier	there exists at least one ...

Table A.1: Connectives in first-order logic.

$$\begin{array}{rcl}
\mathit{Sentence} & \rightarrow & \mathit{AtomicSentence} \mid \mathit{ComplexSentence} \\
\mathit{AtomicSentence} & \rightarrow & \mathit{True} \mid \mathit{False} \mid P \mid Q \mid R \mid \ldots \\
\mathit{ComplexSentence} & \rightarrow & (\ \mathit{Sentence}\ ) \mid [\ \mathit{Sentence}\ ] \\
 & \mid & \neg\ \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \wedge \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \vee \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \Rightarrow \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \Leftrightarrow \mathit{Sentence} \\
 & & \\
\textsc{Operator Precedence} & : & \neg, \wedge, \vee, \Rightarrow, \Leftrightarrow
\end{array}$$

Figure A.1: The syntax of propositional logic, specified in Backus-Naur form.[7]

$$
\begin{array}{rcl}
\mathit{Sentence} & \rightarrow & \mathit{AtomicSentence} \mid \mathit{ComplexSentence} \\
\mathit{AtomicSentence} & \rightarrow & \mathit{Predicate} \mid \mathit{Predicate}(\mathit{Term}, \ldots) \mid \mathit{Term} = \mathit{Term} \\
\mathit{ComplexSentence} & \rightarrow & (\ \mathit{Sentence}\ ) \mid [\ \mathit{Sentence}\ ] \\
 & \mid & \neg\ \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \land \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \lor \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \ \Rightarrow\ \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \ \Leftrightarrow\ \mathit{Sentence} \\
 & \mid & \mathit{Quantifier}\ \mathit{Variable}, \ldots\ \mathit{Sentence} \\
 & & \\
\mathit{Term} & \rightarrow & \mathit{Function}(\mathit{Term}, \ldots) \\
 & \mid & \mathit{Constant} \\
 & \mid & \mathit{Variable} \\
 & & \\
\mathit{Quantifier} & \rightarrow & \forall \mid \exists \\
\mathit{Constant} & \rightarrow & A \mid X_1 \mid \mathit{John} \mid \cdots \\
\mathit{Variable} & \rightarrow & a \mid x \mid s \mid \cdots \\
\mathit{Predicate} & \rightarrow & \mathit{True} \mid \mathit{False} \mid \mathit{After} \mid \mathit{Loves} \mid \mathit{Raining} \mid \cdots \\
\mathit{Function} & \rightarrow & \mathit{Mother} \mid \mathit{LeftLeg} \mid \cdots \\
\textsc{Operator Precedence} & : & \neg, =, \land, \lor, \Rightarrow, \Leftrightarrow
\end{array}
$$

Figure A.2: The syntax of first-order logic, specified in Backus-Naur form.[7]

Appendix B

Event Processing Model and Sensor Model

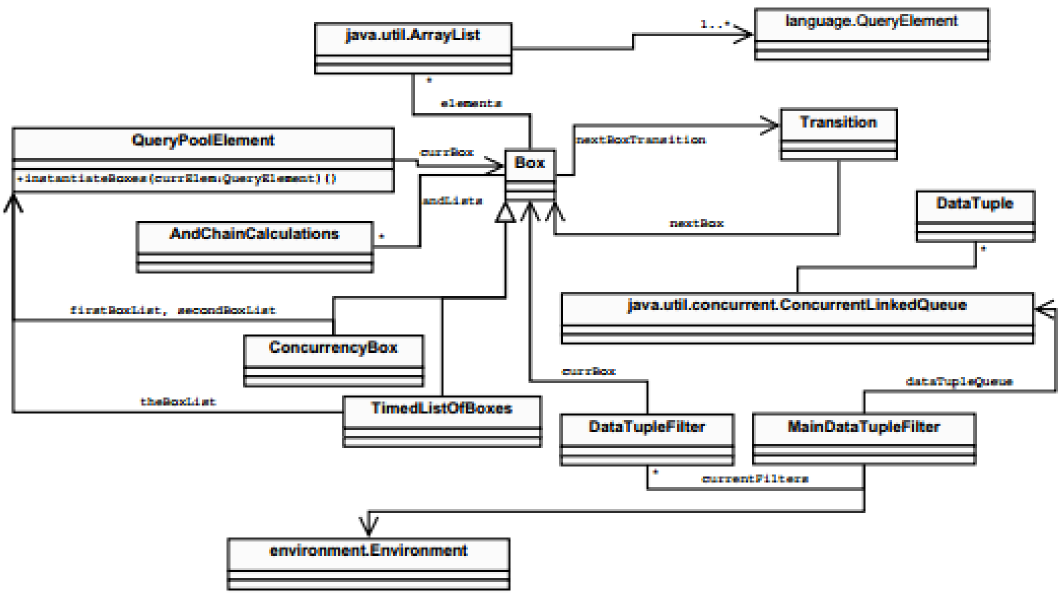

Figure B.1: Key classes of the eventProcessor package.[4]

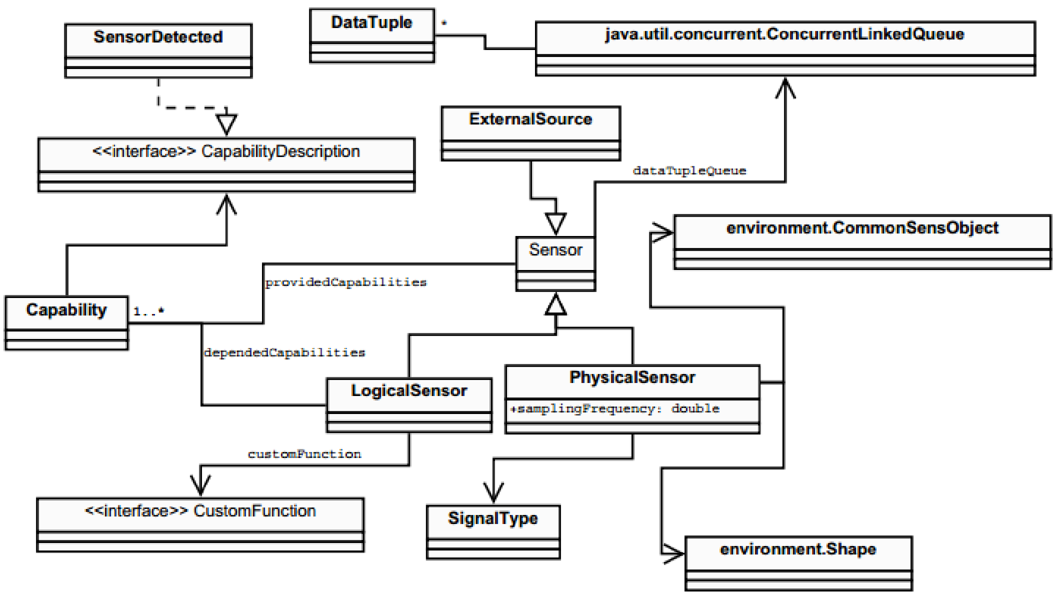

Figure B.2: Key classes of the sensing package.[4]

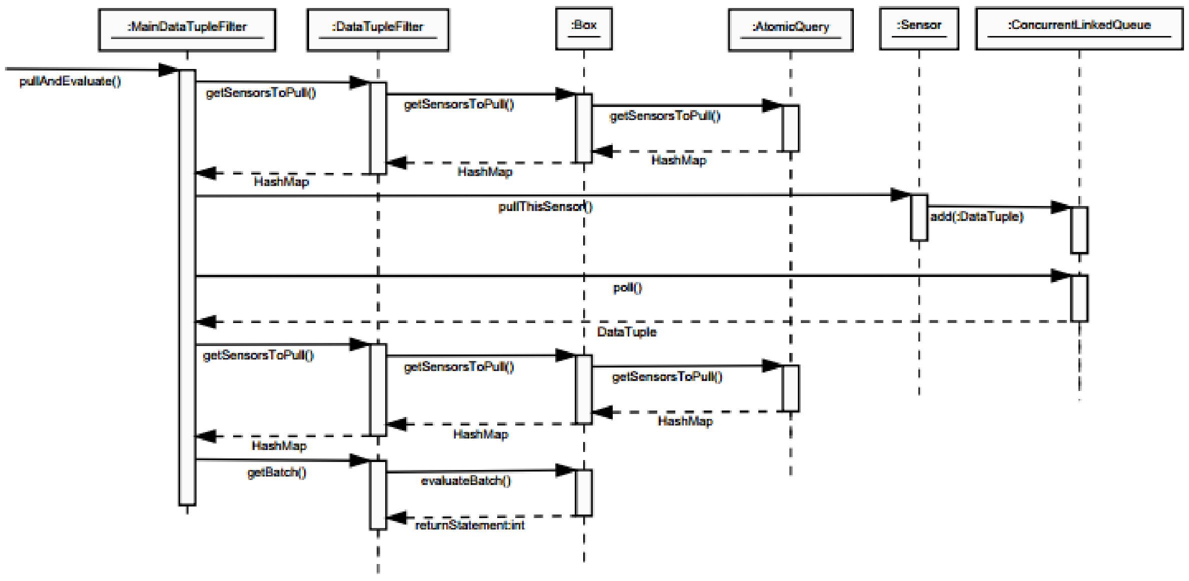

Figure B.3: Overview of the event processing phase.[4]

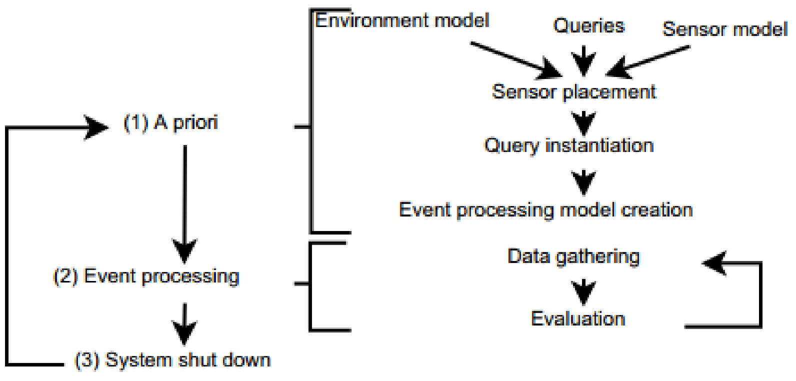

Figure B.4: Life cycle phases in CommonSens.

Bibliography

[1] T. E. Parliament and the Council of the European Union, "Decision no 742/2008/ec of the european parliament and of the council of 9 july 2008," *Official Journal of the European Union*, 2008. [Online]. Available: http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=OJ:L:2008:201:0049:0057:EN:PDF

[2] D. Luckham, "The power of events: An introduction to complex event processing in distributed enterprise systems," in *Rule Representation, Interchange and Reasoning on the Web*, ser. Lecture Notes in Computer Science, vol. 5321. Springer Berlin Heidelberg, 2008, pp. 3–3. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-88808-6_2

[3] A. Gal and E. Hadar, *Principles and Applications of Distributed Event-Based Systems*. Information Science Reference (an imprint of IGI Global), 2010.

[4] J. Søberg, "CommonSens: A Multimodal Complex Event Processing System for Automated Home Care," Ph.D. dissertation, Faculty of Mathematical and Natural Sciences, University of Oslo, 2011.

[5] N. J. Nilsson, "Shakey the robot," Artificial Intelligence Center, Computer and Technology Division, SRI International, Tech. Rep. 323, 1984.

[6] K. Kinsella and W. He, "An aging world," U.S. Department of Health and Human Services, 2008. [Online]. Available: http://www.census.gov/prod/2009pubs/p95-09-1.pdf

[7] S. J. Russell and P. Norvig, *Atificial Intelligence - A Modern Approach*, 3rd ed. Pearson Education, Inc., Prentice Hall, 2010.

[8] J. F. Allen, "Maintaining knowledge about temporal intervals," *Communications of ACM*, vol. 26, no. 11, pp. 832–843, Nov. 1983.

[9] C. Hendahewa, "Types of intelligent agents," 2009. [Online]. Available: http://digit.lk/09_dec_ai

[10] J. P. Müller, “Architectures and applications of intelligent agents: A survey,” *The Knowledge Engineering Review*, vol. 13, no. 4, pp. 353–380, 1998.

[11] M. Wooldridge and N. R. Jennings, “Intelligent agents: theory and practice,” *The Knowledge Engineering Review*, vol. 10, no. 2, pp. 115–152, 1995.

[12] I. A. Ferguson, “Touringmachines: an architecture for dynamic, rational, mobile agents,” Computer Laboratory, University of Cambridge, Tech. Rep. 273, 1992.

[13] M. Wooldridge, *An Introduction to MultiAgent Systems*, 2nd ed. John Wiley & Sons Ltd, 2009.

[14] D. P. M. R. Peter Bonasso, David Kortenkamp and M. Slack, “Experiences with an architecture for intelligent, reactive agents.” *Journal of Experimental and Theoretical Artificial Intelligence*.

[15] A. Rammal and S. Trouilhet, “Keeping elderly people at home: A multi-agent classification of monitoring data,” in *Smart Homes and Health Telematics*, ser. Lecture Notes in Computer Science, S. Helal, S. Mitra, J. Wong, C. Chang, and M. Mokhtari, Eds. Springer Berlin Heidelberg, 2008, vol. 5120, pp. 145–152.

[16] D. Cook, M. Youngblood, I. Heierman, E.O., K. Gopalratnam, S. Rao, A. Litvin, and F. Khawaja, “Mavhome: an agent-based smart home,” in *Pervasive Computing and Communications, 2003. (PerCom 2003). Proceedings of the First IEEE International Conference on Pervasive Computing and Communications*, 2003, pp. 521–524.