Robots doing as they're told: a flexible task execution system taught through human-robot dialogue

Author:
Marko Doornbos

Supervisors:
Dr. C.M. van der Zant
Dr. Bart Verheij

Master Thesis

Institute of Artificial Intelligence and Cognitive Engineering

University of Groningen

April 7, 2014

Abstract

Domestic robotics has become a key part of present-day AI research, with clear real-world problems which robotic systems could solve. However, much of the research in this field still focuses on laboratory environments and the created systems generally cannot cope with the complex circumstances within the real world. There are several notable issues with operating in an unknown environment and three of them will be addressed here:

The first is that the creators of a robotic system cannot be sure which way of executing a task will work best in the environment a specific robot will end up in.

Here, a system was implemented which allows the robot to learn which way to execute a task proves most effective by trying each option several times. This system was implemented as a core part of the robot control architecture used. The consistency in results and statistical basis of this system were tested. This was done by executing three ways to perform a navigationless search task a set amount of times. This was repeated several times, with consistent results. A statistical distinction between the versions used could also be seen in the data gathered most of the time. This demonstrates the viability of this system for certain types of tasks.

A second notable issue in domestic robotics lies in the quality of human-robot interaction using natural speech. A third issue is that robots cannot generally be taught new things once in the hands of an end user. A dialog system, which allows a user to give orders to and teach robotic systems is implemented in the previously mentioned robot control architecture. This makes it one of the first domestic robotic systems which can actually be taught by its end users. This system is subsequently tested through interaction with human operators to determine the ease of use. Improvements are made based on the user feedback, followed by another round of testing. The final version appears to be well-liked by its users, although there are still issues, mainly with speech recognition.

Contents

Chapter 1

Introduction

For decades, robotic assistants have been a staple of science fiction. Having a capable servant who will obey your every command is something a lot of humans want and robotics research seems capable of making that dream come true. Those people involved in robotics research know we are still quite a way from fulfilling that dream. Developing an intelligent and multi-purpose robot turned out to be a lot more difficult than the earliest Artificial Intelligence researchers expected. There are a lot of obstacles to overcome. To make things worse, actually combining solutions to problems we have found turns out to be one of the biggest obstacles of them all. A personal robotic butler of maid will not be available for quite some time.

Even with all of these difficulties, progress is being made. While things like robots with human-level intelligence are still something science is not even close to, simpler but still very useful systems are being developed. Especially in recent years there has been a lot of progress in the field of domestic robotics, which aims to develop robots capable of performing a variety of household tasks. A lot of this progress is driven by and showcased at the yearly Robocup@Home competitions [1]. The first commercially viable domestic service robots are likely to go on sale in the next few years.

A key factor in building a domestic service robot capable of performing a variety of tasks is having a task planning and execution system. Such a system should be both flexible and efficient so the robot does not fail because one small thing is not as expected but does not spend half an hour planning before getting started either. This has turned out to be quite a challenge over the years, but the system presented here offers a possible solution.

1.1 History

Planning is one of the oldest areas of AI research. Over the years, a variety of approaches has been tried:

1.1.1 Early days: state-based reasoning

In the early years of AI, planning systems were mostly based on the then-dominant mindset: logical reasoning and raw processing power are what's needed for intelligent action. Most of the planning algorithms were state-based. They assumed the agent is in a particular state and executed planned actions based on that state and the goal state. These actions are assumed to move the agent from one particular state to another, finally reaching a goal state in which the desired task has been performed.

A good example of this is the STRIPS, that is, the Stanford Research Institute Problem Solver. (Chapter 11 of [2]) [3] Any instance in the STRIPS language consists of the following:

- An initial state for the agent to start in. This is defined by a set of conditions.
- A set of goal states, each defined by a set of conditions. The agent should try to reach one of these.
- A set of actions. Each action has a set of preconditions which must be true to execute the action and a set of postconditions which are true once the action has been executed.

A planner will take such an instance and try to determine a sequence of actions which will change the conditions from the initial state to match them to one of the goal states.

While a planner was developed alongside the STRIPS language, the language has seen much more use than the planner. The STRIPS language and its descendants, like the Planning Domain Definition Language [4] are examples of factored representation. In these, the world is represented by a collection of variables (the conditions mentioned above). When using a factored representation, it is assumed that all variables not mentioned are false. This means that states can be described very concisely; any irrelevant variable does not need to be mentioned. This assumption is called the closed-world assumption.

The STRIPS language was used on the robot Shakey, the first general-purpose mobile robot with reasoning abilities about its own actions. Performance was still limited by the complexity of the reasoning and limited processing power. This meant Shakey spent most of its time standing still while a remote computer performed calculations about its next action before performing another small part of its tasks.

Some planning algorithms also represent the state space and possible actions as a graph. The most obvious way to do so is to represent all possible states as nodes, with actions which indicate reachability through the use of an action as unidirectional edges. However, there are other ways to represent this, which may be more useful.

One of the most common alternate graph representations is the one used by GRAPHPLAN [5]. In GRAPHPLAN, the world is represented as a layered graph. What a node represents is alternated between layers. Sometimes, they represent actions and sometimes they represent atomic facts. There are also two types of edges. The first type connects atomic facts to actions they are a condition for. The second type connects actions to the atomic facts they alter the truth value of. This graph is built up layer by layer, while the system checks if the goal can be reached with the current depth before adding another layer. This checking for solutions is done by backwards chaining. Assuming that the facts that define the goal state are true, it checks if there is a sequence of actions which leads back to the initial state. In each layer, actions which would not be possible because of incompatibility restraints are pruned to speed up this process.

Many different state-based planners have been developed over the last few decades. While some of these plan tasks in a linear manner, many are also capable of planning simultaneous tasks and interleaving actions from different sub plans. Such interleaved planning is necessary to solve certain types of problem, where completely solving one part before solving another is not possible or extremely inefficient.

One way to solve the problem of creating a plan in which sub plans are interleaved is to first create a complete plan where the sub plans are not interleaved. Once this plan is complete, the steps in it are re-ordered to make sure there is no interference between the actions taken to perform the various sub plans. This method was used in WARPLAN [6], a logic planner written in the logic programming language PROLOG. The concept was formally introduced later by Waldinger [7].

Many planners also use heuristics to make the process of finding a suitable plan more efficient. The system which re-popularized this heuristic-based state space planning was UnPOP [8]. This program used the ignore-delete-list heuristic, which relaxes some of the planning constraints. While this approach has its limits [9], it is still a very powerful tool. A very successful example of state-space planning is Fast Forward (FF) [10, 11], which, among other things, won the major AIPS planning competition in 2000. This systems starts off by using a GRAPHPLAN-style algorithm using relaxed constraints to find an explicit solution, which is then used as a goal distance estimate. From then on, the system uses an enforced local hill-climbing algorithm to perform a local breadth-first search, which is limited by the state information from the relaxed solution to prune useless actions. This is done iteratively until a solution is found. While the use of a hill-climbing algorithm may lead to some dead ends caused by local maxima, the system has a few contingencies to deal with those situations.

The main problem with state-based systems is that they require a lot of information about the environment to determine the state they are in. Typically, part of that information is not available since the agent cannot observe the entire environment. While that can be compensated for, these kinds of systems generally still have problems with incorrect assumptions and states changing without actions from the planning agent.

One further issue is that environments with a realistic level of complexity generally have an enormous state space and many actions used to traverse this space. The resulting combinatorial explosion means that massive amounts of processing power are needed to generate plans.

Another approach in the same direction is using a Belief-Desire-Intention framework to guide the choice of actions, like with the Procedural Reasoning System [12]. This uses a multi-layered system where the agent has a set of beliefs about the world, a set of desires which basically form the agent's goals and a set of intentions which describe the actions the agent has decided to perform. The agent also has a set of knowledge areas which describe possible actions to change the world, used for executing the intentions. The biggest problem with this kind of system is that possible beliefs, desires and intentions have to be developed by a programmer alongside the knowledge areas, limiting the agent to what it knowns and without room to learn from its experiences.

One way of reducing the complexity issues for these planning system is to use hierarchical planning.

In such a system, actions in a plan are collected in higher-level sub plans. By splitting the planning task like this, the planning for each subtask has considerably fewer possible actions, which can speed up the search for an effective plan tremendously. These sub plans are generally referred to as High Level Actions (HLA). Typically, there are several hierarchical layers of sub plans, nested in each other. How many of there layers there are might also vary depending on the task being planned.

However, hierarchical planners usually still need to plan the entire task execution to make sure the goal can be reached. This means the entire plan usually needs to be redone when assumptions turn out to be false and the current plan can no longer work.

The solution to this is to make sure that all possible executions of a HLA have the same preconditions and postconditions, or at least a clearly defined and consistent set for each HLA. If this is the case, there is no need to plan the entire execution in-depth since the consequences of executing a sub plan are already known and can be taken into account when deciding on the next sub plan to use. The planning within each HLA only needs to be done when that portion of the task is reached. Since detailed planning only starts when the HLA needs to be executed, the system can use the most up-to-date information about the situation to make sure an effective sub plan is created. This method of hierarchical planning is sometimes called just-in-time planning.

One disadvantage of a hierarchical planning system where all possible executions of a sub plan have the same preconditions and postconditions is that the system and its possible actions need to be deliberately crafted to make sure this is the case. This means more work and more restrictions for the developers.

The first steps towards this kind of planning were already made within STRIPS [13] when a macro-operator functionality was introduced to it. Since then, many planners using hierarchical methods have been made, like O-PLAN [14].

1.1.2 A different direction: reactive robotics

After a while, a different approach to controlling robotic agents started to gain popularity. In this approach, the focus was on systems which would be more reactive, basing most of their low-level actions on the immediately observable state of the world. Higher-level behavior would simply be built on top of these basics and take over when necessary. This behavior-based robotics approach [15] was quite a different way to develop agents capable of intelligent action.

The basic concept underlying all behavior-based architectures is that an agent has access to various behaviors to perform and that it decides which one to use at any point in time based on the data it has access to. The typical example is Brooks's subsumption architecture. [16] In this case, the agent's behaviors are layered. The lowest-level behavior is executed by default. Under particular circumstances, the next-level behavior will take over and determine the agent's actions instead. For this behavior, the next level can take over in particular circumstances as well, and so on for each layer. In this way, a basic and functional agent can be built upon so it can do more complex things. The biggest advantage to this approach is that it allows an incremental build-up of capabilities. An agent can start off just driving around avoiding obstacles. Once this behavior is working properly, a new layer can be built on top of it. Once that layer is complete, an additional layer can again be added. Having functional agents at each level of development makes it easier to test for errors, since a problem can only be caused by a newly added behavior.

Of course, this approach also has its problems. The biggest of these lies in how one would determine what behavior to use. In a classic subsumption architecture, the limit of a single behavior per layer means performing different tasks would require complex layers of subsumption which are not easily developed. Other ways to determine the action to be executed would be needed.

Something which often occurs in simple reactive agents like this is emergent behavior. [17] While there are many definitions of emergent behavior, its core concept is that simple rules in a complex environment can lead to complex behaviors. For example, many complex-seeming animal behaviors such as birds flying in formation, fish swimming in schools and ants traveling across seemingly flat areas in a complex pattern can be reproduced using a few simple rules. In fact, the concept can also be applied to various high-level structures in human society or even unregulated human-maintained networks such as the Internet. In the case of fish swimming in schools, the behavior can be reproduced using three simple rules, applied individually for each fish:

- If you are within distance a of another fish (or an obstacle), change your swimming direction away from them.
- Else, if you are not within distance b (usually larger than a) of another fish, change your swimming direction towards them.
- Else, swim straight ahead.

These simple rules will generate school-swimming behavior, although the exact nature of the behavior will depend on swimming speed, the degree by which the swimming direction is modified, the number of fish and the obstacles present.

Another classic example of reactive robotics are Braitenberg's vehicles [18] which display distinct behaviors through simple combinations of light sensors which determine the motor output for the robot's wheels.

Another strength of reactive robotics is that it is generally not necessary to have a complex world model. Basic behavior such as obstacle avoidance does not need a world model, only sensor information about the current environment. Complicated high-dimensional analysis of situations is typically not needed either. Instead of relying on complex and detailed planning, reactive systems can try a simple approach, with basic responses to common problems such as obstacle avoidance or inaccuracies in odometric data.

Sometimes, developers will find that these simple responses also allow the system to avoid more complex problems through a combination of their effects in a manner they did not foresee. This is generally classified as a form of emergent behavior.

However, reactive architectures also have their fair share of problems. The biggest of these problems is that the interplay between behavior levels needed to obtain complex-high level actions requires an extremely intricate set of behaviors. Creating such a set is impossible for most scientists because of the level of complexity required. Another issue is the inflexibility of reactive systems. They are typically made to perform a single task. Allowing a system to switch to another task would generally require a high-level control behavior, which would run into the complexity issues described previously. Finally, reactive systems generally rely on sensor data to determine their course of action from moment to moment. If the available sensor systems are inaccurate or unreliable in some other way, reactive systems can have serious issues maintaining their intended behavior over time. Even if the sensor issues are taken into account during creation, they can impact the complex interplay often found in reactive systems.

Arkin [19] provides an excellent overview of various other reactive systems.

1.1.3 The current state of the field

Most modern planning systems lie somewhere on the spectrum between these two approaches. How pro-active or reactive these systems are can vary quite a bit, but there is generally a trade-off between the two. Too strong a reactive focus can seriously impede the execution of a task since the system can keep getting distracted. Too strong a proactive focus can result in failure by not taking the environment into account sufficiently. See Chapter 5 of [20] for a more detailed analysis of that trade-off.

A common and interesting approach here is just-in-time planning, which was mentioned previously. Using this method only a general, high-level plan is formed at first, with the details filled in once it becomes necessary to do so. In this way, plans do not need to be rewritten every time something changes from the initial assumptions made by the agent while there is still a general plan to be followed. Both the problems of a changing environment and the handling of failed actions can be addressed this way.

These low-level plans will often include reactive components, such as reactive obstacle avoidance as a default part of all movement execution.

A typical hybrid system is the three-layer architecture. The three layers in this architecture are the reactive layer, the executive layer and the deliberative layer. The deliberative layer determines the higher-level plans, typically in a traditional planning method. The executive layer determines which actions need to be taken to execute the current part of the plan. It typically includes navigation and path planning. The reactive layer receives these instructions from the executive layer and executes them.

All three of these layers re-evaluate their current actions at different speeds. The reactive layer, which has very simple instructions, processes them very quickly, often dozens of times per second. For example, it might alter the direction of movement when an obstacle is detected in the robot's path. This rapid processing is often necessary to handle sudden changes in the environment, such as a person walking by the robot. The executive layer re-evaluates less often, typically around once per second. It determines if the current course of actions is complete, still going on or no longer viable and instructs the reactive layer accordingly. The deliberative layer re-evaluates even less frequently. It can often take minutes before a re-evaluation occurs at this layer, since they are only needed if things are not going as planned.

The three-layer architecture is fairly loosely described, and does not necessarily have three layers. The core element of the architecture is the separation of low-level reactive actions, controlled by high-level planning. In this way, it can take advantage of the strengths of both classical planning and reactive systems, while avoiding many of their pitfalls. The system was already used on Shakey, although it was still severely limited back then. [21]

A different but also common approach is the pipeline architecture. In this, there are several systems working in parallel.

Some parts of the system process sensor data and use this to update the agent's current world model. This can include both things like obstacle locations for low-level avoidance and higher-level information like mapping data or the observation of objects of interest.

Other parts use the most recent available data in the world model to evaluate the current high-level plans and modify them if necessary.

These plans are again executed by other parts of the architecture. These are often low-level, reactive systems, much like those in the reactive layer of a three-layer architecture.

Finally, there are parts in the system which transmit the data between parts and take care of the most basic control portions, such as transforming movement commands into input for the agent's motors.

The main difference with the three-level architecture is that all parts of the system are always running in parallel and asynchronously. The various parts do not wait for new sensor data from the other parts, but simply use the most recent processed data available. This results in a robust and fast system. The architecture will not get stuck waiting for new data from a certain sensor, but will simply keep going using slightly older data. This asynchronous parallel processing also allows for systems to be optimized for parallel computers or even multi-computer processing.

The software architecture used in the experiments done here uses a variation of the pipeline architecture using a just-in-time planning system fully integrated into the architecture.

Recently, the importance of robot-human interactions has become more prominent. For example, Fischer has shown the usefulness of having a natural dialogue with the robot in order to improve the quality of instructions [22]. Of course, these kinds of dialogue have their own challenges. An excellent overview of these challenges has been given by Scheutz, Cantrell and Schermerhorn [23]. Peltason and Wrede give some examples of current dialogue systems with their strong and weak points in their article [24]. All three articles were published in *AI Magazine volume 32, number 4*, a special issue on dialogue with robots. It also contains other articles which might be of interest to those focusing on human-robot interaction.

In recent years, many innovations in the field of domestic robotics have been created through the Robocup@Home competitions. [1] The difficulty of the challenges in these competitions is increased each year, based on the performance of teams in the most recent competitions. However, progress is limited somewhat because there is no requirement for the teams to publish the methods they used to achieve their results. Still, many teams do publish parts of their research for the competitions. A list of publications by the teams is available at the Robocup@Home wiki. [1]

A rare example of a published modern robot control architecture created for the Robocup@Home challenges is SitLog [25], which is used by the Golem Robocup@Home team. This system uses multiple logic interpreters to analyse a situation and the tasks to be performed in a mostly hierarchical manner. In this, it takes advantage of the high-quality logic processing of the Prolog programming language.

1.2 Goals of this project

The primary goal of this project was to further develop the task planning and execution system developed at the Robotics Lab at the University of Groningen. [26] This system is described more fully in Chapter 2. It functions like a hierarchical system where reactive portions are generally built into low-level actions.

While its core functions have been implemented, there were several potential features considered to improve the system's performance. These additions were also to be used for the Robocup@Home competition, in which the University of Groningen's BORG team participates. [2] This competition would serve as a final test of sorts.

This project consisted of multiple stages, with each stage having multiple sub-goals. All of them serve to add more functionalities to the task planning and execution system. Each of the phases would also have an experiment to demonstrate the added functionality.

Before the first phase, the whole system would be checked to ensure it still functioned as intended, as further development of the in-house BORG robot control architecture could have resulted in some faults. After this checking was completed, the development proper would start.

The first phase in the project would be to add a method by which the system could have multiple possible executions for a behavior. By learning from experience, the generally most suitable version could be selected. This learning would allow the system to pick the optimal behavior based on which approach works in the actual environment the agent has to work in. This kind of selection is a must for properly adaptive systems meant to operate in a domestic environment. Developing the technology for such systems is the primary goal of the Robocup@Home competition. An experiment was run to see

[1]http://robocup.rwth-aachen.de/athomewiki/index.php/Publications

[2]http://www.ai.rug.nl/crl/

if this system was indeed consistent in its results, and if the information collected was sufficient to statistically distinguish the tested versions. This phase is described in Chapter 3.

In phase two, detailed failure reporting and logging would be implemented, alongside a test of the system's human-robot interaction capabilities. This failure reporting is not only very useful information for development, but can also help users understand why their instructions could not be executed. This reporting would also allow the user to give new instructions taking the reasons for failure into consideration. To determine if the interaction and failure reporting were at a sufficient level for real-world use, an experiment was performed where users tried to interact with the system. After a series of improvements based on the results of that experiment, the system was tested again in a similar manner. This part of the project is discussed further in Chapter 4.

Chapter 2

System overview

The just-in-time planning and execution system used here relies on two types of elements: The first are behaviors, which are the basic building blocks of a task like moving to a certain location or grasping an object. New behaviors cannot be learned while the system is running, at least in its current implementation. However, the system can know multiple behaviors which do the same thing in different ways and choose which one to use. Scripts are the second type; these combine the execution of various behaviors and other scripts to perform more complex tasks and can be learned.

This system was designed by Bas Hickendorff, who also created the first implementation. [26] It was specifically developed for use with domestic robots, which generally need to operate in highly complex environments which cannot be tested in during development.

With behaviors as the core building blocks and scripts as a way to tie them together, there is a clear distinction between the mutable part (the scripts) and the less-mutable but still flexible part (the behaviors).

This gives the user a fairly clear idea of what they can and cannot teach the system. It cannot be taught wholly new actions, but it can combine the things it already knows in different ways.

2.1 Task execution

When the agent receives a verbal instruction, it will check for scripts matching that instruction. Currently, this is done using a keyword dictionary. This means the detected speech is matched to known ways of expressing a command. If there are multiple matches, the most specific version is chosen.

If no matching script was found, the agent will ask the user to explain how the task should be performed. How this works is explained in detail in the next section.

Once the agent has a matching script, it will begin executing that script. A script is effectively a step-by-step plan for the execution of a task, where each step is either a behavior or another script. Whenever a step in a script is another script, that script will be executed step-by-step as well before continuing with the higher-level script. This happens recursively, so the behaviors in the task structure are executed in sequence. One can imagine this structure as a search tree. The script for the original instruction is the root. Branching happens when a script contains multiple subscripts/behaviors. Each leaf is a behavior. See Figure 2.1 for an illustration of this concept. Task execution proceeds depth-first through the entire tree, only continuing through it once the most recently found behavior has been executed.

By only looking further into a script when it needs to be executed, planning remains efficient. Only the things that need to be executed immediately are planned in detail. This is called just-in-time planning.

The just-in-time part of the planning structure is useful because creating a detailed plan generally requires some time, especially if there are lots of options. Choosing between these options later in the plan generally requires knowledge of the state of the world at that point in the plan execution. This kind of knowledge is generally not available in real-world situations. Just-in-time planning sticks to a general plan, detailing sections only when those sections are actually reached and the relevant state of the world can actually be observed. In this way, there is less backtracking because of wrong assumptions and less time wasted on making useless plans.

There are situations where just-in-time planning is inefficient. Mostly, these are situations where one of the last parts of a plan cannot be executed and a large part has to be redone. However, preventing these usually requires the agent to have detailed knowledge about the environment during later stages of the plan. Such detailed knowledge is generally not available to agents operating in a realistic, dynamic environment. Furthermore, the system can be taught conditional preferences to avoid these pitfalls if they occur frequently.

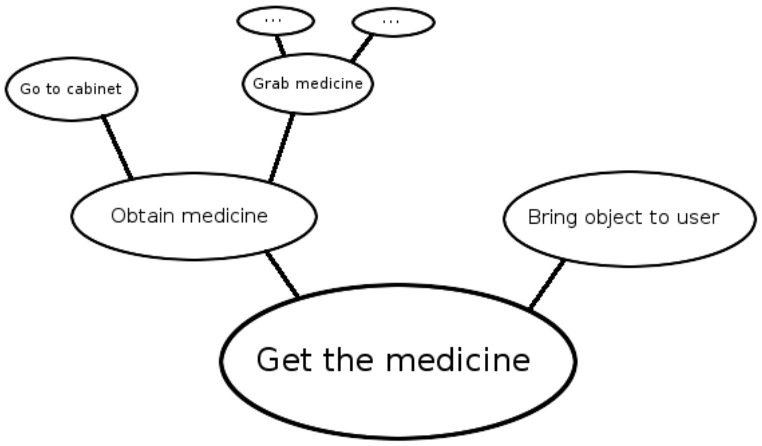

Figure 2.1: An example of the task-execution structure used by this architecture. This tree is explored depth-first. Any behaviors that are reached in this exploration are executed.

2.1.1 Script selection

There can be multiple scripts which perform the same task or subtask. In that case, the system will have to choose one to execute. This is done based on learned preferences. For example, the user may prefer getting coffee over getting tea when he asks the agent to get him a drink. Other preferences can have conditions based on the current state of the agent or the world. For instance, the user might prefer hot chocolate over coffee if he asks for a drink in the evening. These conditions will act as additional preconditions for the execution of those specific scripts.

Preferences are learned in two ways:

If the user tells the robot something like "I prefer to have coffee instead of having tea" it will recognize this as an instruction about preferences, determine which scripts this instruction refers to and register the new preference for them. Determining the scripts the user refers to happens in the same way as when the user tells the agent to do something. Currently, this is by using a keyword dictionary.

The preferences themselves are simply stored as statements which indicate the execution of one particular script is preferred over the execution of another. If the condition is based on the current state, the way to check for that condition is stored with it.

Preferences are defined transitively, so checks for loops happen. If a newly learned preference contradicts an older preference, the new one overwrites it. In this way, newer knowledge is preferred since the user might have changed his mind about something.

If the instructions given by the user are underspecified, the system's first recourse is to check the known preferences. For example, if the user asks for a drink and his preference for coffee is known, the agent will get some coffee for the user. If this is not enough to resolve the instructions, the agent will ask the user to specify the instructions further. For example, the agent might ask the user what he wants to drink. The answers to questions asked in this way are also stored as preferences. This is the second way preferences are learned.

2.1.2 Exceptions

Behaviors and scripts can also have exceptions for when unexpected things happen to the execution of a task or subtask. Exceptions on a behavior level are handled within the behavior and do not necessarily cause the behavior to fail as they can be resolved internally and might not be mission-critical. For example, the failure of a simple logging function does not prevent the successful execution of a task.

Exceptions on a script level cause a script to fail if it is currently running and prevent it from being run if it is not yet running. Exceptions are stored within the scripts they are for and are checked whenever a script is updated while running and whenever it is eligible to be executed. They are there to ensure no time is wasted in trying to execute a script which has no chance of succeeding. For example, a script for navigating might have an exception for when it encounters a closed door in its path if the robot is unable to open doors. In that case, it would be better to use another script which is able to find a different route without closed doors.

2.1.3 Handling failures

When a behavior or script fails, alternative options for the same task will be executed. These are the other options which could have been chosen but which were not taken because of preferences. When more preferred options fail, the others are tried as alternatives. For example, if there is no coffee left, the agent might get the user some tea instead, even though the user prefers coffee. For behaviors, a limit to the number of tries can also be set.

If all the options for a script or behavior fail, the failure will move one step higher in the hierarchy. The script which caused the failed one to run will try to run alternatives, also failing if all of its options fail. In this way, a task will only fail if none of the known ways to execute the task can be completed.

2.2 Additional functions

The previous section explained the general task-execution structure of the architecture. There are several systems which support this structure.

2.2.1 Task instruction

The system uses a simple speech recognition package using the Sphinx system[1] to detect the user's instructions. If speech is detected, the system first determines the nature of the command based on the presence of certain structures or keywords. This way, commands to perform a task are distinguished from instructions about preferences and instructions about performing tasks.

To teach the system how to perform a certain task, the user simply tells the robot. For example: "To perform A, first do B, then do C and then do D." In this, A, B, C and D can be any script or behavior. If any are unknown, the system will ask the user to tell them how to do that part. Then, the user can simply instruct them on that part as well. The instructions are simply stored as new scripts.

While this system does not allow the user to teach the system new behaviors, these are typically too complex and low-level to have end users instruct the system about them. That kind of new functionality would require actual programming work, at least with today's standards for robotics. Actually teaching a robot a new way to walk, for example, is not something done during runtime.

2.2.2 Behavior selection

The selection of behaviors is done using the Interval Estimation machine learning algorithm. [27] This means that each available version of a behavior is first tried several times to get an indication of success rates and completion times. After that, statistical analysis of the collected data is used to determine the best version when the behavior needs to be executed again.

[1]http://cmusphinx.sourceforge.net/

This new execution adds another data point, which means the statistics for that version can be updated. In this way, the system keeps learning during runtime.

There are several different ways to determine the version to be used based on the distribution of completion times. The current implementation supports both optimistic and pessimistic suggestions based on completion times. These are all based on a 95% confidence interval for the distribution of completion times per version. For a behavior where being faster is better, the version with the lowest lower bound is the optimistic choice, since it is likely to be done quickly. The version with the lowest upper bound is the pessimistic suggestion, since it has the best chance of being done before that point.

For behaviors where lasting longer is better, the version with the highest upper bound is the optimistic suggestion, while the one with highest lower bound is the pessimistic one.

Which criterion is to be used can be set per behavior. Several other things can be tweaked, like the size of the confidence interval and the minimum number of executions per version to get a reliable statistical distribution estimate. This allows the developer to still use the machine learning for behaviors where the normal settings are not adequate.

For a more detailed explanation of Interval Estimation, read the next chapter.

2.3 Physical implementation

The system is implemented as part of the custom architecture of the BORG Robocup@Home team. This means it can be used with any robot that can be controlled by the architecture. At the time of writing, there are several platforms available. The primary ones are listed here.

2.3.1 Nao

The Aldebaran Robotics Nao[2] (See figure 2.2) is a mid-sized humanoid robot carrying a wide variety of sensors including cameras, sonar and pressure sensors. Its limbs have a wide range of articulation. It also has a speaker system and the default software supports speech generation. While it is not suitable as a household service platform by itself, it does provide a small and reasonably reliable platform for testing some software.

2.3.2 Sudo

The Sudo (See Figure 2.3) is the BORG team's custom robot for the Robocup@Home competitions. It is based on a modified MobileRobots Pioneer[3]. A metal frame has been added on top, which allows the mounting of one or more laptops for control/processing in addition to various sensors. Available sensors include Kinect camera/depth sensors, webcams, microphones and a rear-facing laser range finder.

[2]http://www.aldebaran.com

[3]http://www.mobilerobots.com/

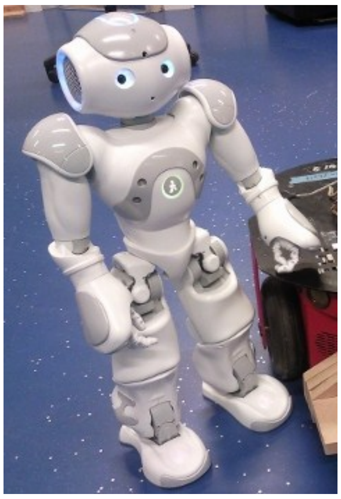

Figure 2.2: The Aldebaran Robotics Nao.

Of particular note is the option to place a Nao onto the Sudo, providing the platform with arms to manipulate objects with as well as the Nao's sensors.

The Sudo's default configuration consists of a Nao to interact with people and objects, the laser range finder for navigation, one Kinect looking down for obstacle avoidance, one Kinect looking forward, a high-quality microphone to pick up speech commands and a single laptop to control the system.

The whole platform has thick cardboard panels covering the interior and the superstructure. This is primarily to keep cables and other parts protected, but it also improves the robot's aesthetics.

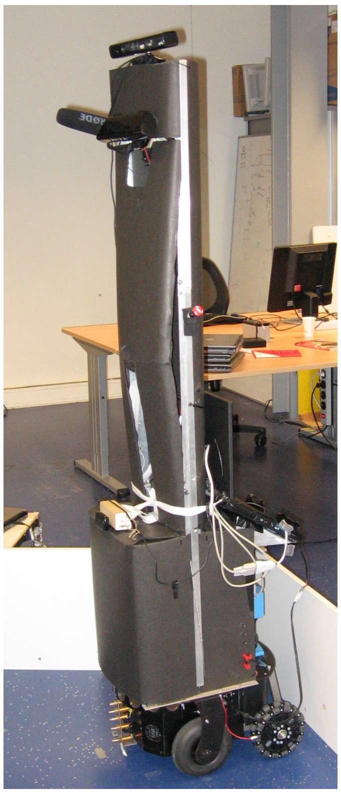

Figure 2.3: The Sudo platform of the BORG RoboCup@Home team. Note the Kinect sensors mounted at the top of the frame for both vision and depth perception, along with the microphone used to hear the user's commands. There also is a webcam mounted on the front of the lower frame to assist detection of objects at table height.

Chapter 3

Phase 1: multiple behavior executions

In this phase, the functionality to have different versions of the same behavior was added. To determine which of these versions works best in the environment the agent is in, the Interval Estimation machine learning algorithm [27] was used. In the following experiment, the performance of the Interval Estimation in different situations was compared.

3.1 Introduction

There was a very simple reason for adding the ability to learn the best way of completing a task for a specific robot in a specific environment.

A system of this kind will be critical for domestic-use robots, especially for platforms which may end up in a wide variety of different environments.

In a situation like this, it becomes nigh impossible to decide on the optimal way to execute a certain task, since the performance of different versions may vary wildly between different environments.

Shipping the robot with a version which works decently in most environments seems to be the obvious solution. However, finding a version which is good enough for the vast majority of environments is a massive challenge as well. When one considers that this kind of selection should be done for a wide range of behaviors for performing various tasks, things get even more tricky. Finally, to determine how well different versions work in various environments, those environments first need to be studied and modeled correctly. This is another difficult and costly challenge.

Having a system which can use multiple versions of each behavior and learn which one works best for the environment it ends up in makes this issue a lot less complicated. While research and modelling of the possible environments is still necessary, less testing and decision making will be needed.

Instead of having to pick a single version of the behavior, developers can simply pick the versions which work best for various environments, possibly alongside one which has decent all-round performance. Once a robot has been delivered to its environment, it can just try the various versions it has access to and determine which works best.

In a properly-developed system it should even be possible to send new versions of a behavior to the robot if they are developed later in the system's life cycle. At that point, the robot can simply try that new version several times and incorporate it into its learning.

At first glance, the disadvantage of this system seems to be that multiple versions of a behavior are needed. However, during development of a behavior, multiple versions taken into consideration would need to be implemented and tested anyway, to determine which is better. So there is little to no extra work involved there.

3.1.1 Research Use

As an added bonus a system like Interval Estimation can also be used for research purposes. By using this algorithm, one can obtain statistically valid data about the performance of different behaviors/algorithms for a task during runtime. In fact, the implementation used here keeps all the data which would be needed to perform a statistical analysis of the performance of different behavior versions. This also makes it easy to weed out bad implementations of a behavior during development, simply by studying the performance compared to other versions.

3.2 Methods

As mentioned in the previous chapter, Interval Estimation uses statistical analysis of behavior completion times to determine which version of a behavior works best for the agent.

While not all versions of a behavior have been executed a set minimum number of times, one is selected randomly from the ones that have not been executed enough. In this way, data points are gathered for each version. Once the minimum number of executions has been reached for each version, the completion times for those executions are used to estimate the statistical distribution of completion times. This is done individually per version. The resulting normal distribution estimates can then be used as a measure to compare the different versions of the behavior.

There are several ways to use these distribution estimates to select a version to be executed. All of them make use of confidence intervals for the distributions, usually of 95%. But first a distinction has to be made between the different execution time/success values that can be used.

While this system uses unweighted completion times, it is also possible to weight the data points differently. This weighting would be done by multiplying completion times by different factors, based on some criterion. This makes some data points more important than others.

For example, a discount factor could be applied to older results, so that recent results are taken into account more heavily when deciding. As an illustration of the usefulness of this, moving around the furniture in a home could drastically change which path-planning method works best. A system which discounts older results will adapt to these changes more quickly, since the older information, which has a larger chance of being out of date, is not considered as important.

Another thing which could be done is to weight the completion time for a navigation task by the (known) distance between the starting point and the goal.

A different option would be to simply use success/failure data, by weighting all successful attempts at 1 and all failed attempts at 0. This would be a natural solution for behaviors where completion times vary wildly without a good weighting option or are unreliable for some other reason.

Even with just unweighted completion times, there are several ways to interpret the data.

First off, a distinction needs to be made between behaviors where lasting longer is better and ones where being faster is better.

For example, a goalkeeping behavior for a soccer robot would need to keep the ball out of the goal as long as possible. In this case, a goal being scored is considered a "success" and stops the behavior. There are now at least two ways to decide on the best version of a behavior. The first, optimistic way is to use the version with the highest upper bound. This is the version that is likely to run the longest. The pessimistic alternative is to pick the version with the highest lower bound. That version is likely to run at least that long, giving a decent minimum performance.

For many other behaviors completing a task more quickly is better. In this case, picking the behavior version with the lowest lower bound is the optimistic version and picking the version with the lowest upper bound is the pessimistic option. Here, the distinction is between the version which is possibly the fastest and the version which is least likely to be the slowest.

Whether optimistic or pessimistic choices are used would depend on the nature of the task the behavior is used for. At least so far, there is no easy selection criterion. In our implementation, the optimistic choice is currently always taken.

3.2.1 Quality of Version Selection

It is mathematically proven that the Interval Estimation algorithm converges to the optimal solution given infinite time. However, assuming one has access to infinite time in a real-world application is ridiculous.

The greatest strength of the algorithm lies not in this theoretical convergence, but in the fact that it selects the version which is the best according to current data. This means that a version which has not been performing well will not be selected if there are better alternatives. Even if the version that is chosen at a certain point is not the optimum given infinite time, it has performed at least just as well as the optimum up to that point. In this way, only options that appear good at the time are chosen and executed. The bad options are rooted out and the good options keep getting used.

3.2.2 Implementation

This addition was implemented in the architecture used by the Robocup@Home team of the University of Groningen, BORG. This is the same architecture the previous parts of the system were implemented in. Previously, users had to add additional layers of coding to use the functionalities. Now, this addition was made as a part of the core functioning of the architecture, making it easy to use for all tasks.

3.2.3 Setup

During the experiment, the robot was set up at a fixed point in the arena. From its starting position, the system had 120 seconds to find its target. This target was alternated between a black silhouette of a bicycle and a black and white version of the symbol for radioactivity. Both were printed on white sheets of paper. These images can be seen in Figure 3.1. Such a simple target was chosen because this experiment was about testing the Interval Estimation and not the object detection.

Whenever the system succeeded or failed to find the target, it simply started the next search, using the other target. The search was started from the position the previous search attempt had ended at. This meant there was no pre-defined starting location for the second and later search attempts. While this might introduce a lot of noise to the search times, it is likely a search situation in the real world will have a similar degree of variation.

The system was set up with three different navigation-less search behaviors. Default obstacle avoidance was on for all three behaviors, keeping the robot at least 10cm from any detectable obstacle by changing the direction of movement away from it. Small direction changes away from an obstacle start when the robot gets within 50 cm of it and is driving towards it. This direction change is increased as the robot gets closer to obstacles. This means the robot will steer away from obstacles in a fluid manner. This avoidance uses a vector field method. [2]

The robot does not turn back to its original heading after avoiding an obstacle using this method. So if the robot is programmed to drive forward and it approaches a wall, it will start turning away from that wall while still going forward at a lower speed. At some point, the wall is no longer in the way and the robot will continue moving forward in the direction it ended up pointing in. In this case, it would probably be roughly parallel to the avoided wall.

The following search behaviors were chosen:

1. Drive forward for 10 seconds at the maximum stable speed for acceleration and deceleration, while avoiding obstacles. Moving too fast risks the robot falling over and instability would hinder the target detection. After stopping, make a random turn (a random angle was chosen each turn) while stationary. This process was repeated until the target was found or time ran out.

2. Drive forward at the maximum stable speed, while avoiding obstacles.

3. Drive forward at half the maximum stable speed, while avoiding obstacles.

These behaviors were chosen based on the results of an earlier experiment with navigation-less search algorithms, the results of which are unpublished. The behaviors used here are variations of the most successful ones in that experiment.

Version 1 was chosen because the random turns make it likely it will not get stuck in a pattern of movement, like running in circles between several obstacles. The disadvantage of using these random turns is that the robot can spend a lot of time in an area of the search space if the random turns happen to point it towards that area. That makes this version most useful for smaller search spaces, especially if other search methods have a high risk of getting stuck in movement patterns which do not take them to the targets.

Version 2 was most successful in previous experiments with navigation-less search, unless it got stuck in a movement path which did not take it to its targets. The combination of high-speed movement and obstacle avoidance allows this strategy to effectively search through many different arenas of various sizes. It was expected to be the best in this experiment as well.

Version three was similar to version 2, but with a lower movement speed. This improved the chances of detecting the target if it got into view because of the more stable video image. Because of the interaction with the obstacle avoidance, this version would also take different paths from version 2, staying further away from obstacles.

Each of these behaviors would turn towards the search target if it was detected, to make sure the detection was correct. If the target was detected in 4 distinct frames at the same position within 3 seconds, it was considered found. Requiring this strict a criterion meant not a single false positive was encountered in the duration of the experiment.

The Interval Estimation algorithm first had to try each version a minimum number of five times. After that, it was allowed to keep going until a total of forty searches had been completed. At that point, basic training is assumed to be complete. From that point onwards, the algorithm would keep running. It would select the version which has had the best results up to that point in each new search attempt and update those results once the search attempt had ended.

This was repeated six times, under circumstances detailed below. Each time, the Interval Estimation started untrained.

The goals in this experiment were twofold: The first was to determine if the Interval Estimation consistently selected the same version in several experiments using similar circumstances. The second was to determine if the relatively small number of forty search attempts would be enough to obtain statistically distinct completion time distributions.

Based on earlier experiments using Interval Estimation, it was expected that the winning version after forty search attempts would be fairly consistent. There was less certainty about the statistical distinction between versions. While earlier experiments did have apparent winners after a relatively small amount of attempts, like twenty, that amount of data was not enough to determine if the distributions of the completion times were statistically distinct.

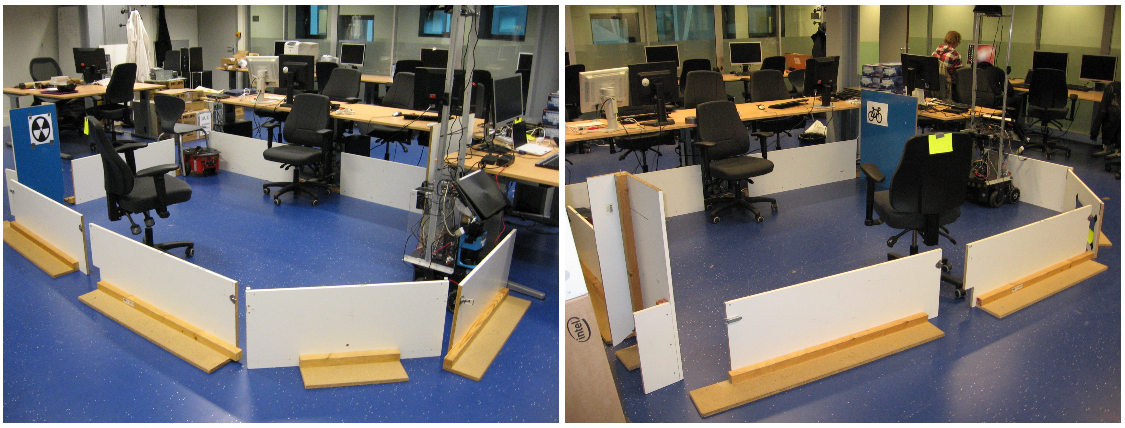
Figure 3.1: The experimental environment. The Sudo robot used in this experiment is placed within it, at its starting position.

Hardware

In this experiment, the BORG team's Sudo platform was used. As described in the previous chapter, this is a Pioneer robot which has been strengthened to support the metal and Plexiglas frame on top of it and the various laptops and sensors which can be attached.

For this experiment, a single laptop was used to control the entire system. Sensors used were a generic webcam mounted on the front of the robot to detect the targets and a Kinect distance sensor [1] to detect obstacles.

The environment

The environment used in this experiment was set up inside the Cognitive Robotics Lab (CRL) at the University of Groningen. See Figure 3.1 for pictures of the setup.

Use of a larger arena was considered, but the robot's limited battery life and the limited amount of time available to run the experiment prevented the use of the entire CRL as the arena.

3.3 Results

3.3.1 Consistency

In five out of six experimental runs, version 2 of the search behavior was used almost exclusively after the three versions had all been tried five times. However, version 1 was primarily used in the fourth run, with version 1 still used often as well.

[1]The Mircrosoft Kinect is a camera system which also uses a depth sensor to detect an array of distances alongside the regular video feed. Pointed downwards, this allows the robot to detect obstacles in the space in front of it.

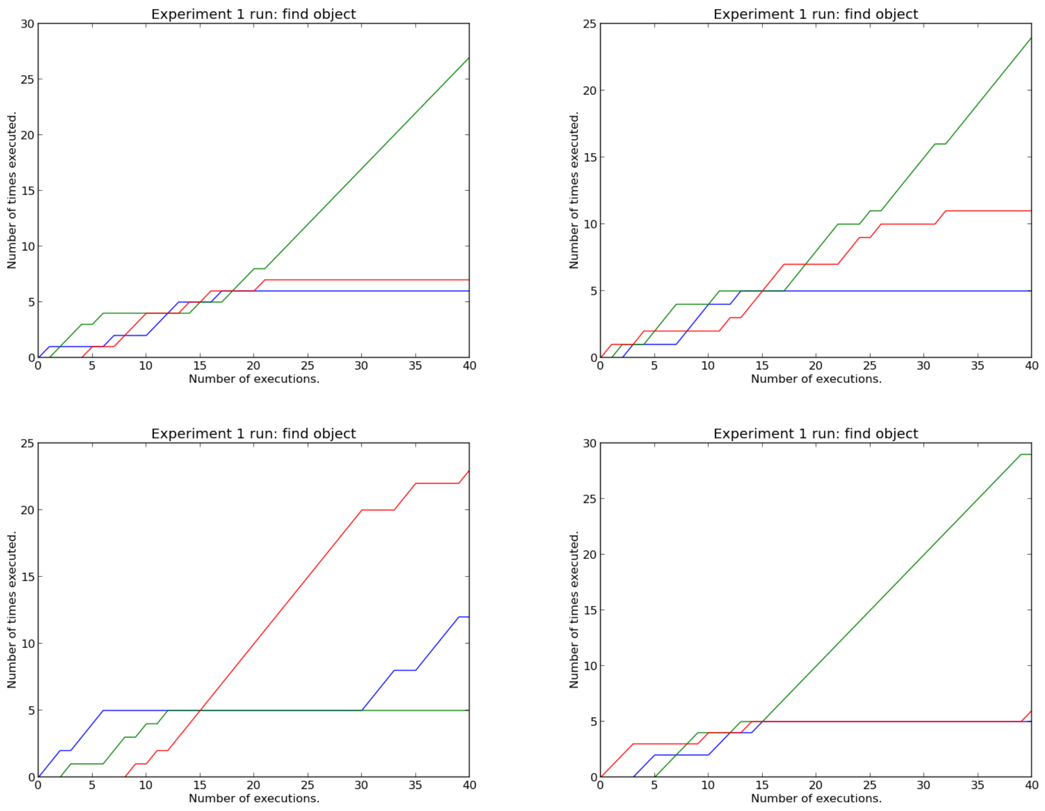

Figure 3.2: Graphs of the version executions for runs 2, 3, 4 and 5. They show the number of times each behavior was used over time. The blue lines are for version 1. The green lines are for version 2. The red lines are for version 3. The x-axis shows time, measured in the number of search attempts made. The y-axis shows the number of times each version was used. Graphs of runs 1 and 6 were left out because only search version two was ever executed in those runs after initial training.

See Figure 3.2 for example graphs of the versions used in several of the runs.

Although many more runs would have to be executed to determine if this kind of consistency is statistically significant, there appears to be a reasonable degree of consistency.

3.3.2 Statistically distinct versions

To determine if the completion times of the different behaviors were statistically distinct, a set of ANOVA tests with a significance level of 95%, that is, a required $P < 0.05$ for significance, was run. All but one of the distributions of the completion times were statistically distinct within each experimental run. See Table 3.1 for the numbers for each run:

Experimental Run	P
1	0.004
2	0.027
3	0.034
4	0.349
5	0.032
6	0.037

Table 3.1: The results of the ANOVA tests for each of the experimental runs. Using a 95% significance level ($P < 0.05$), the sets of completion times for the three different search behaviors were compared to see if they were statistically distinct. This was the case in all of the runs, except run 4. This was the run where version 1 performed slightly better than version 2.

3.4 Discussion

All in all, the Interval Estimation proved quite effective in its primary function. That is, to test various ways to perform a task and eliminate the versions which did not perform well.

While convergence to a particular algorithm is interesting, it is not necessary for optimal performance. In fact, if two methods are very close in terms of effectiveness, there is no reason to force the system to choose one over the other. The key part of the system is to make sure an effective method (or small set of methods) is chosen for the situations this specific agent finds itself in.

Of course, the selection algorithm should still be reasonably consistent if the circumstances are similar. After all, similar circumstances generally mean that the same methods would function well. And this was shown in the experiment.

In a fairly consistent environment, the Interval Estimation algorithm will generally converge to the same version or versions of an implementation.

As for the usefulness of Interval Estimation as a tool to gather statistical data, it appeared to be effective. If one version keeps getting chosen, it is statistically clearly distinct from the other versions. However, an increase in the minimum number of attempts for each implementation might be needed to make sure a sufficient number of data points is gathered.

The initial expectation that search version 2 would perform the best turned out to be correct most of the time, although version 1 performed well at times as well but was limited by the randomness inherent to its search pattern.

Chapter 4

Phase 2: failure reporting and general interaction

4.1 Introduction

The ability to learn online both with and without feedback from the user is very important for a robotic system in the real world. Another extremely important factor for a robotic system is the ability to interact with a human user naturally, both when receiving instructions and when reporting failures.

That ability was the focus of the second phase in this project.

While the existing system already had some basic interaction through natural speech, this had never been tested with non-expert users. The system did not report any failures to the user either. It just quietly processed them before waiting for a new command. Of course, this was very likely to confuse non-expert users. Even expert users could sometimes be confused about what the robot was doing.

4.1.1 Planned improvements

There were two main improvements made to the system at the start of this phase.

The first was to expand the interaction part. Previously, it was only really used with very limited sets of possible commands. Now, it had been expanded to accept a wide variety of commands, based on those used in the General Purpose Robot Challenge in the Robocup@Home. [1] In fact, this interaction setup would end up being used in the Robocup@Home World Championships 2013, with only some small changes to the available object names and location names.

The second initial improvement was to report each failure by having the robot say so when a plan (script) or action (behavior) failed. This way, the user would get some indication of what went wrong if an error occurred.

4.1.2 Further improvements

After the first set of improvements had been implemented, they would be tested. Based on the feedback of the non-expert users, another round of improvements would then be added. These improvements would then be tested as well.

4.2 Methods

4.2.1 The improved interaction options

The improvement of the interaction was done primarily by expanding the grammar used by the speech recognition.

Such a grammar defines a structure where certain strings are defined as objects, locations, actions et cetera. Such defined sentence parts can then be used as parts of bigger sentence parts or entire sentences.

The grammars used up to that point had had some fairly complex possible structures, but only a few possible elements for each part within such a structure. For example, there would be only three to five defined objects and only a single way to tell the robot to get an object.

Now, the set of core commands was expanded to five:

- Go to location
- Introduce yourself
- Get object
- Follow someone
- Memorize a person

Each of these commands had two to four basic ways of being expressed. For example, the command to "go to" a location could also be given as "navigate to" or "move to". A few more complex expressions like "take the -object- from the -location-" were also available.

The number of known locations and objects was also expanded. About a dozen of each were available.

All of this combined provided an immense variety of possible commands, especially with the sequencing of commands which was possible.

Next, the keyword file which links the recognized speech to the available scripts and behaviors was also expanded to include all of the new options. Finally, some new scripts also needed to be generated to fit the new options.

4.2.2 The failure reporting

The first version of the failure reporting was based on the existing internal failure-registration measures. These would register the failure, stop the behavior and continue running the architecture. Several additions were needed here.

The first of those was a logging feature. Whenever a script or behavior started or stopped (including failing), some basic logging information like the name of the behavior/script, the version being used and the time of the start or stop would be written to a CSV file which was both easy to read for human users and easy to parse for software. Starts and failures used separate files.

This addition would primarily be useful for developers who are debugging a large system. In a situation like that, it is possible to have several behaviors running at once. That can make spotting the exact cause of an error or crash fairly difficult. By logging which behaviors have started and which have stopped, pinpointing the location of an error can become a lot easier.

This logging would also form the basis for the actual reporting: when a handled failure (so not a total crash of the code) occurs, the system would verbalize that that specific plan (in case of a script) or action (in case of a behavior) had failed. For example: "Action go-to-location has failed." Since more information about the failure was generally unavailable within the architecture, it was not possible to verbalize a more detailed explanation of the failure at that point.

4.2.3 Experimental goals

The purpose of this experiment was to test the current level of interaction and failure reporting.

Specifically, to test this with subjects who were not intimately familiar with the system, to make sure an inexperienced user would be able to interact with the robot and give it commands. This is one of the critical features needed in a household service robot.

4.2.4 Setup

For this experiment, the setup was a lot more limited than in the previous one.

Hardware

Since using the Sudo robot would mean sharing time with other experiments being conducted, only a Nao was used here as the robotic platform.

A laptop was used as the processing platform. A headset microphone was also connected to this to receive the speech input. While the Nao does have a microphone, its quality is limited. If the experiment were run on the Sudo platform, its high-quality microphone could have been used instead.

Software modifications

Having the robot drive or walk to various locations when instructed and letting it try to find and grasp objects would take a lot of time. Furthermore, there would be a high chance of getting too many failures. While some failures would, of course, be needed to test the failure reporting, there was a fairly high chance of getting a failure during the execution of each command when trying to perform them all physically. In such a situation, the user would not have a baseline for correct execution, which could skew the results.

That is why the robot was only used to vocalize the system's statements and form an interaction target for the user. The execution of the commands would be done in the architecture, but using a very simple form of simulation.

When a bottom-level behavior used for executing the available core commands was started, it would not try to execute the usual physical movements on the robot. Instead, the behavior would decide if the task had succeeded or failed after several seconds and announce this result if successful. In case of failure, the behavior would, of course, set itself to the failed state.

This choice between success and failure was based on a random number generator; each behavior had a specific chance to fail. The behavior to get an object could actually fail in two ways: by not finding the object or by failing to grasp the object. While the outward reporting of those failures would not be different, the distinction *was* made internally. This would be useful later.

Although this modification would make the interaction slightly less natural since the user could not actually see the robot perform the tasks, the limits on time, success rate and room made it necessary.

The instructions

After a short introduction, a test subject would have to give the robot five different commands, one after the other. After the robot had tried to execute the command, they would have to indicate their agreement or disagreement with a few statements about the interaction and the executions. This was done using a standard five-point scale (–, -, +/-, +, ++).

Those statements were the following:

- The robot understand the command correctly.
- The robot tried to execute the command correctly.
- I understood what the robot was doing or trying to do.
- The interaction with the robot felt natural.
- *If something went wrong:* The robot was clear about what went wrong.

After filling those in on the form they were provide with, they were to give the next command.

The sequence of commands started with two simple, one-piece instructions. Those were followed by two complex commands consisting of three parts each. Finally, the user had to instruct the robot about how to perform a specific task and then tell it to perform it.

The commands were:

1. Sudo, introduce yourself
2. Sudo, navigate to the living room.
3. Sudo, go to the kitchen, take the crackers and leave the apartment.
4. Sudo, navigate to the bar, tell something about yourself and exit the apartment.
5. Sudo, go with him. *followed by:* Sudo, to go with him, first memorize the person and then follow the person in front of you. *then, once more:* Sudo, go with him.

The robot would ask for a confirmation after each received command. The user could reply to those with "Sudo yes" or "Sudo affirmative" as a positive response and "Sudo no" or 'Sudo negative" as a negative response.

After receiving a positive response for a given command, the robot would proceed with executing it.

The participants were allowed to try giving the instructions as often as they would like. If they felt the robot was not able to understand the command properly, the experimenter would verbally give the command to the robot instead.

The test subjects

The five test subjects for the initial run were all AI students or former AI students at the University of Groningen. All of them also had at least some experience with robotics, although they had not worked with the verbal interaction part of this system. All of them were males. Their grasp of English (the language used in the interaction) ranged from decent to fluent.

The location

The experiment was conducted in the Cognitive Robotics Lab at the University of Groningen. The Nao was sitting in front of the user, who was holding or wearing the headset as they preferred. The experimenter was sitting to the side, starting the software, answering questions from the user and making sure the experiment proceeded correctly.

There were some other things going on in the lab during the experiments, but the noise did not rise above the level of normal conversation.

4.3 Results

The results for this experiment have been divided between the subjects the users were questioned about.

A summary can be found in the following table:

Command	- -	-	+/-	+	++
The robot understood the command correctly	1	3	4	7	10
The robot executed the command correctly	0	2	3	2	18
I understood what the robot was trying to do	0	0	4	6	15
The interaction with the robot felt natural	0	2	4	16	3
The robot was clear about what went wrong	2	5	0	0	2

Each row shows the results for one specific evaluation question the users were asked. The numbers in the columns following the evaluation questions show how often each evaluation was given for that question. For example, the users rated the naturalness of the interaction with the robot as ++ three times during this experiment.

4.3.1 Understanding the command

These results were generally positive. Most responses were either + or ++. The robot usually understood the command within the first three tries. However, two of the participants had some initial trouble in getting the robot to listen to them.

Notably, both of these participants were speaking slowly, with clear pauses between each word. When both participants tried speaking naturally, without pauses between each word, the performance of the speech recognition was similar to that with the other participants.

4.3.2 Executing the command correctly

All of the participants rated all of the commands where no error was encountered with + or ++.

If an error was encountered, most of the participants gave a score of - or –. However, in those cases the robot did try to execute the command correctly, which could be seen in the execution logs. It had been explained to the participants that attempting to do the correct thing but failing was an acceptable result for this statement and should not result in a lower rating here. However, they seemed to disregard this instruction.

4.3.3 Understanding the robot's actions

Subjects were generally positive about understanding the robot's actions when no errors occurred, with the occasional rating of +/-.

But when an error did occur, their understanding ranged from – to +/-, tending towards –.

4.3.4 Naturalness of the interaction

The naturalness was on average rated +. In most cases where the rating was lower than that, it was caused by the robot having difficulties in understanding the user's command.

4.3.5 Clarity of the failure reporting

For two of the cases where an error was reported, the rating was ++. In the other seven cases, it was either – or -. In general, the users considered the error reports unnatural.

4.4 Discussion

The main conclusion from this experiment was that the error reporting was clearly inadequate. Most of the time, users had no idea what had gone wrong. The repeated statements of failures when a lower-level behavior failing caused higher-level scripts to fail as well were also deemed confusing.

This would be the main thing to be fixed in the next part of the experiment.

An interesting piece of additional data encountered during this experiment was that two of the users tried talking to the robot slowly, with clear pauses between their words. When asked about it later, they claimed they did so to make things easier to understand for the system. However, the result of this unnatural speech pattern was that the robot was unable to understand them. This was because the speech recognition system was tuned for regular human speech, in which words flow into one another, without pauses between them. When the users were requested to speak naturally instead, the system had very little trouble understanding them.

This highlights a concern in the testing of human-robot interaction. Human users will sometimes make assumptions about what makes things easier for the robot, and those assumptions can easily be incorrect or even counter-productive. While a problem like this could theoretically be solved by properly instructing users about it, that might not be feasible for systems placed in a domestic environment, where it is impossible to instruct all possible users.

Another interesting feature of these results was that most of the participants seemed to misinterpret the statement "The robot tried executing the command correctly." as "The robot executed the command without errors." This is a simple mistake to make, and future participants would receive some additional instruction about this statement.

4.5 Improvements based on user feedback

Now, a set of improvements was implemented based on the feedback given during the first stage of the experiment. These improvements would then be tested using both new and repeat test subjects.

4.5.1 Failure reporting

As stated previously, the primary problem to be fixed was the unclear failure reporting.

The multiple failure reports had been confusing. The use of the internal names for scripts and behaviors was also making the reports more difficult to understand. For example, users would receive an error report like: ”Behavior goto has failed. Plan goto-get-object-goto has failed.” when the robot was trying to go to the kitchen to get the crackers before leaving the appartment (command 3), but was unable to reach the kitchen. While this kind of feedback could be useful for developers who know and understand the names used, non-expert users were just confused.

Therefore, a different way of reporting the failures was implemented. Instead of vocalizing errors at the logging step, the lowest-level core behaviors would vocalize what went wrong as soon as the error was noted. Higher-level parts of the system would not give further error messages to keep things simple.

The error messages themselves would also be different. Instead of simply stating that a certain action/behavior had failed, they could now give the specific reason given inside the behavior. For example, the behavior to go to a specified location would now state that ”I could not reach the kitchen.”, where kitchen would be the place it was currently trying to reach. It was hoped that statements like that would be easier to understand and more natural for the user.

4.5.2 Current actions

Additionally, participants sometimes seemed uncertain about what the robot was doing, even though this was not fully expressed in the scores given for the relevant statement.

In each of the lowest-level behaviors (except the one which has the robot introduce itself) the robot would already state it had completed that action. So after completing a command to go to the kitchen, it would state: ”I am now in the kitchen.”

But now, most lowest-level behaviors would also state their goal as they were started. For example, the behavior to go to a location would state ”I will now go to the kitchen.” when going to the location kitchen. This should also increase the user's understanding of the robot's actions, especially since they cannot see it do things physically.

4.6 Methods for the improved version

In order to test these modifications and make sure they are actually improvements, a second round of the experiment was run.

This second round was several weeks after the first round.

4.6.1 Participants

Four of the five participants in the first round participated again in the second. They received no additional instructions, but were obviously more familiar with the system in this round.

In addition to them, there were four new participants. All of them were also AI students at the University of Groningen. Their grasp of English also varied from decent to good. Three of them were males and the fourth was female. All of them were in their twenties.

Differences between the results of the two groups will be discussed below, where a relevant difference existed.

4.6.2 Other changes

The rest of the experiment was executed in the same way as during the first round.

4.7 Results

Again, the results have been divided by subject. The following table summarizes them:

Command	- -	-	+/-	+	++
The robot understood the command correctly	3	1	7	6	23
The robot executed the command correctly	2	3	7	1	28
I understood what the robot was trying to do	0	1	0	6	32
The interaction with the robot felt natural	1	1	6	16	16
The robot was clear about what went wrong	1	1	1	5	6

4.7.1 Understanding the command

Once again, these results varied considerably.

The repeat test subjects had little trouble with the speech component, although none of them were understood perfectly all the time. Most of the interactions were rated ++, with a lower rating ranging from - to + for the cases where recognition did not function perfectly.

On the other hand, some of the new participants had considerable trouble getting the robot to understand them. Some of them had the same problem as the first group of participants when they started. They were talking too slowly, which made things difficult for the speech recognition.

The speech recognition seemed to have trouble with the female participant's voice in particular.

Ratings for the new participants varied wildly. Most of the ratings were –, +/- and ++, in about equal numbers. A few - and + ratings were also given.

4.7.2 Executing the command correctly

The results for this subject were very similar to those in the first round.

Once again, most commands were rated ++.

But again, some subjects (both new and repeat) gave lower ratings when the system encountered an error during execution, even though they had been informed that encountering errors should not lower this rating if the agent tried to do the right thing but failed in the process.

4.7.3 Understanding the robot's actions

The vast majority of commands were rated as ++ on this subject.

A few subjects rated one or two commands as + instead and one participant rated a single command as -.

4.7.4 Naturalness of the interaction

The naturalness was rated as + or ++ for most commands by most participants, with the occasional +/-.

There were a few – and - ratings, all of them in cases where the participant was having trouble with the speech recognition. This included both repeat and new users.

4.7.5 Clarity of the failure reporting

Most of the failure reports were rated + or ++. However, a few were rated as +/-, - or –. All of those were cases where a more complex command failed in the first or second step, preventing the execution of the rest of the command.

4.8 Discussion

4.8.1 Clarity of the robot's actions

The first clear improvement was that the users were much more aware of what the robot was doing and trying to do. Scores were higher for both the new and repeat users. Since this is a critical part of working with a domestic service robot, it was critical that this facet of the interaction was improved.

However, the fact that users could not see what the robot was doing physically may make this reporting of what the robot is trying to do redundant or even annoying in the long term when implemented on a mobile platform. Determining if this is the case would require much long-term testing.

4.8.2 Failure reporting

The users also considered the error reporting clearer and more natural.

However, some users were confused when the robot did not try to execute the next steps in a command if a step before it failed.

To the robot's internal planning, it was clear that it could not perform those next parts. After all, it is not possible to locate and grasp an object if the robot is not in the room where the object is located.

The failure reporting did not inform the user of this, though. And it seemed that some users assumed the robot should still try those next steps.

Based on these results, it looks like the current level of failure reporting is sufficient for single tasks, but insufficient in cases where failure in part of a task prevents any following steps from being executed.

For those cases, it might be useful to at least state that the later parts of the command could not be executed because of this failure.

4.8.3 Speech recognition

Understanding the user's speech still seemed to be a major obstacle, especially in the case of inexperienced users.

Much of the trouble was caused by the quality of the speech recognition software, which simply could not handle some of the speech patterns encountered. The only real way to improve this would be to use better speech recognition software.

But there were some issues which could be prevented.

In some cases, speech recognition was impeded because the user was unclear about what the system was expecting the user to say.

For example, they would try repeating the command to be given while the robot was still waiting for a confirmation that it understood the command correctly. In such cases, the user simply lacked knowledge about the state the robot was in.

There could be several ways to deal with this problem.

One option would be to have the robot repeat its last statement at set intervals of time. This way, the user would be reminded of the input the robot is asking for.

Another option would be to show the robot's current state in some way. For example, the Nao's coloured lights could be used for this. In this case, one colour could be used when the robot is waiting for a command, another when it is waiting for a confirmation and a third when it is executing a command. There could also be a fourth colour for when the robot's emergency button is active.

Such an improvement would be used to supplement the verbal information the robot gives.

Chapter 5

Final evaluation

5.1 Results of the project

In the end, this project has added a considerable number of improvements to this robot control framework, bringing it that much closer to the point where it can be used for systems interacting with people in a real, non-experimental situation.

The added adaptability of the Interval Estimation learning will allow robots to adapt their behavior to fit their environment, greatly improving their general effectiveness.

The human-robot interaction capabilities are also at a level where users are generally pleased with the interaction. Only in cases where the robot had serious trouble recognizing their speech did they feel the interaction was unnatural.

Both of these factors are critical for robots used in an interactive household situation. Getting them at this decent level means that research can now shift its focus further towards those areas which are still clearly lacking in the system, such as object grasping.

5.2 Remaining questions

However, there are still some questions that remain about the research done in this project.

5.2.1 Interval Estimation

One of the biggest uncertainties lies in the effectiveness of the current version of Interval Estimation for more varied behavior components.

The current implementation of Interval Estimation uses completion times to generate its distributions and determine which version is more suitable. While this works well for many tasks, there are others where completion times are simply not suitable. Most notably, this includes tasks which are to be performed until the user commands the robot to stop, like following someone. In this case, completion times are not useful for determining the effectiveness of a version since the minimum and maximum times to complete the task will vary without a clear pattern.

In such a case, the only data available is that of success or failure. Fortunately, the Interval Estimation algorithm can work using a binomial distribution, but this would still have to be implemented in the system.

5.2.2 Interaction

There are always ways to further improve human-robot interaction.

The big question that remains about the interaction as it is now is if the speech interpretation based on the most likely heard sentence is flexible and efficient enough if the amount of recognizable speech is increased. To understand a range of commands fit for use in the real world, a considerable amount of processing and analysis is needed.

Many of the problems with the interaction occurred when the speech recognition returned grammatically valid but logically and contextually inconsistent sentences which would never be uttered by a human operator. Such unnatural interpretations of speech occur easily in software systems, but humans hardly ever have this problem. This makes the interaction seem unnatural.

Solving this problem would be a highly complex undertaking. Especially the usage of context-based information is a key part of human speech recognition, but it is virtually never used in speech recognition software since discerning the relevant context can itself be very difficult.

5.3 Potential future work

This project offered a variety of options for future research.

5.3.1 State analysis and prediction

The current logging system publishes the starting and stopping of various behaviors. These changes in running behaviors can be seen as changes in the robot's and the world's state.

With some additional work, the system could express this state in a format suitable for modern systems using formal logic.

This would make it possible to analyze the system's behavior and even generate new combinations of behaviors based on the state transition data. However, it is a distinct deviation from the current behavior-based system with its speech-based instruction. This means it would probably require quite a bit of new research.

5.3.2 Improved failure handling

Another thing that could be improved is the system's failure handling.

Currently, when a failure occurs and cannot be handled by trying a different behavior at any level, the system just stops where it is and reports the failure. One potential improvement is that the robot will try to look for the user and report the issue in person. After that, it might ask what it should do if the situation occurs again. That information can then be used to generate a new behavior to use if a similar failure occurs. This generation would work in the same way as the generation of behaviors through verbal instruction.

In this manner, the system will slowly become able to handle more and more potential failures.

5.3.3 Long-term Interval Estimation

Most research on using Interval Estimation so far has focused on more short-term testing.

To ensure the stability of the system and test the long-term adaptability of the algorithm, it will be necessary to place one or more robots using Interval Estimation to learn in a real-world environment for a long-term experiment.

The main issue with such an experiment is that most robots currently available are simply not suitable for long-term independent operation in the real world.

5.4 Final thoughts

All in all, this project has considerably improved the functionality of the system used. Most of these improvements could be implemented in any behavior-based robot control architecture, making it much more flexible and capable of working long-term in the real world.

Automatically learning which version of an action is most effective for this specific agent in its specific environment is something which has been done before. However, this is, as far as I know, the first time it has been implemented as a core part of a robot control architecture.

Evaluations of human-robot interaction in have primarily focused on systems which were designed with interaction as a primary concern. Any task execution based on this interaction was of a much lower priority in those studies. On the other hand, the interaction evaluated here is part of a system which was primarily designed to execute tasks, with interaction only as a tool for this execution. The quality of the interaction in this system was less of a concern than the quality of any task executions. This meant that the focus of the testing was also on the usefulness of the interaction in completing the tasks the user wants the agent to complete. This shift in focus is perhaps more important than the specific results here.

We are getting closer and closer to the day when having a single, multi-functional robot to assist you with general household tasks is commonplace. Once that day arrives, the elderly in particular will have a considerably improved standard of living.

We can only do our best to ensure that day comes before we are in need of this robotic assistance ourselves.

Bibliography

[1] T. Wisspeintner, T. van der Zant, L. Iocchi, and S. Schiffer, “Robocup@home: Results in benchmarking domestic service robots,” *Interaction Studies*, vol. 50, no. 3, p. 392, 2010.

[2] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach.* Pearson Education, 1995.

[3] R. E. Fikes and N. J. Nilsson, “Strips: A new approach to the application of theorem proving to problem solving,” *Artificial Intelligence*, vol. 2, no. 3–4, pp. 189 – 208, 1971.

[4] M. Ghallab, A. Howe, C. Knoblock, D. McDermott, A. Ram, M. Veloso, D. Weld, and D. Wilkins, “Pddl - the planning domain definition language,” Tech. Rep. CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, October 1998.

[5] A. L. Blum and M. L. Furst, “Fast planning through planning graph analysis,” *Artificial intelligence*, vol. 90, no. 1, pp. 281–300, 1997.

[6] D. H. Warren, *WARPLAN: A system for generating plans.* Department of Computational Logic, University of Edinburgh, 1974.

[7] R. Waldinger, *Achieving several goals simultaneously.* Sri International, 1975.

[8] D. V. McDermott, “A heuristic estimator for means-ends analysis in planning.,” in *AIPS*, vol. 96, pp. 142–149, 1996.

[9] J. Hoffmann, “Where’ignoring delete lists’ works: Local search topology in planning benchmarks.,” *J. Artif. Intell. Res.(JAIR)*, vol. 24, pp. 685–758, 2005.

[10] J. Hoffmann, “Ff: The fast-forward planning system,” *AI magazine*, vol. 22, no. 3, p. 57, 2001.

[11] J. Hoffmann and B. Nebel, “The ff planning system: Fast plan generation through heuristic search,” *Journal of Artificial Intelligence Research*, vol. 14, 2001.

[12] M. P. Georgeff, “Communication and interaction in multiagent planning,” in *Proceedings of the National Conference on Artificial Intelligence*, pp. 125–129, 1983.

[13] R. E. Fikes, P. E. Hart, and N. J. Nilsson, "Learning and executing generalized robot plans," *Artificial intelligence*, vol. 3, pp. 251–288, 1972.

[14] K. Currie and A. Tate, "O-plan: the open planning architecture," *Artificial intelligence*, vol. 52, no. 1, pp. 49–86, 1991.

[15] R. Brooks, "Intelligence without representation," *Artificial Intelligence*, vol. 47, pp. 139–159, 1986.

[16] R. Brooks, "A robust layered control system for a mobile robot," *Robotics and Automation, IEEE Journal of*, vol. 2, no. 1, pp. 14–23, 1986.

[17] P. A. Corning, "The re-emergence of "emergence": A venerable concept in search of a theory," *Complexity*, vol. 7, no. 6, pp. 18–30, 2002.

[18] V. Braitenberg, *Vehicles: Experiments in synthetic psychology*. MIT press, 1986.

[19] R. C. Arkin, *Behavior-based robotics [electronic resource]*. MIT press, 1998.

[20] M. Wooldridge, *An Introduction to Multi-agent Systems*. Chichester, England: John Wiley and Sons, 2002.

[21] E. Gat *et al.*, "On three-layer architectures," *Artificial intelligence and mobile robots*, pp. 195–210, 1998.

[22] K. Fischer, "How people talk with robots: Designing dialogue to reduce user uncertainty," *AI Magazine*, vol. 32, no. 4, pp. 31–38, 2011.

[23] M. Scheutz, R. Cantrell, and P. Schemerhorn, "Toward humanlike task-based dialogue processing for human robot interaction," *AI Magazine*, vol. 32, no. 4, pp. 77–84, 2011.

[24] J. Peltason and B. Wrede, "The curious robot as a case study for comparing dialogue systems," *AI Magazine*, vol. 32, no. 4, pp. 85–99, 2011.

[25] L. A. Pineda, L. Salinas, I. V. Meza, C. Rascon, and G. Fuentes, "Sitlog: A programming language for service robot tasks," *International Journal of Advanced Robotic Systems*, vol. 10, p. 538, 2013.

[26] B. Hickendorff, "Enhanced human-robot interaction by reason-based behavior adaptation," Master's thesis, University of Groningen, 2011.

[27] T. van der Zant, M. Wiering, and J. van Eijck, "On-line robot learning using the interval estimation algorithm," in *Proceedings of the 7th European Workshop on Reinforcement Learning*, pp. 11–12, 2005.