Grounding Oriented Design

Christopher Stanton

A Thesis submitted for the degree of Doctor of Philosophy

Faculty of Information Technology

University of Technology, Sydney

2007

Certificate of Authorship and Originality

I certify that the work in this thesis has not previously been submitted for a degree nor has it been submitted as part of requirements for a degree except as fully acknowledged within the text.

I also certify that the thesis has been written by me. Any help that I have received in my research work and the preparation of the thesis itself has been acknowledged. In addition, I certify that all information sources and literature used are indicated in the thesis.

Signature of Student

Acknowledgements

A huge “thankyou” goes to my supervisor, Professor Mary-Anne Williams. In late 2001, I was a disillusioned software engineer, bored with my chosen career path, drifting from one job to the next, searching for something more, and for some strange reason, I found myself spending my time at work daydreaming about artificial intelligence (perhaps because I wanted the machines I was programming to do the work for me). So, I made inquiries at my local university (Newcastle) about the possibility of undertaking a research honours degree in artificial intelligence. These inquiries led me to Mary-Anne, and as the cliche goes, the rest is history. Mary-Anne introduced me to robotic soccer, offered me the opportunity to undertake a PhD at UTS, entrusted with me an important leadership role (leading the software development for the UTS RoboCup team), allowed me the freedom to pursue the research areas that interested me, provided me with the opportunity to discuss the grounding problem with great minds such as John McCarthy and Peter Gärdenfors, and gave me the guidance to keep me pointed in the right direction. Not only was her guidance and support invaluable, but she has been a great mentor.

Another huge “thankyou” goes to my parents, Dr John Stanton and Dr Patricia Stanton. My parents, both academics, have obviously had an enormous influence on my being the person I am today. Their support, advice, proofreading and love, has been offered, as always, unconditionally.

Table of Contents

List of Figures

List of Tables

Abstract

The symbol grounding problem[67] is a longstanding, poorly understood issue that has interested philosophers, computer scientists, and cognitive scientists alike. The grounding problem, in its various guises, refers to the task of creating *meaningful* representations for artificial agents. After more than 15 years of widespread debate and circular introspection of the so-called symbol grounding problem we seem none-the-wiser as to what constitutes being meaningful, and indeed grounded, for an agent[128].

We argue, in the context of practical robotics, a grounded agent possesses a representation which faithfully reflects pertinent aspects of the world. In contrast, an ungrounded agent could be, for example, delusional or suffering from hallucinations ("false positives"), overly concerned with irrelevant things (e.g. the frame problem[93]), or incapable of reliably perceiving, recognising or anticipating relevant things in a timely manner ("false negatives"). While most grounding research concerns how to develop agents which can autonomously develop their own representations (i.e. *autonomous* grounding), the fact all robotic systems are grounded through human design on a case-by-case, ad-hoc basis has been overlooked. This thesis presents *Grounding Oriented Design* - a methodology for designing and grounding the "minds" of robotic agents. Grounding Oriented Design (or, alternatively *Go-Design*) is a vital first step towards the development of autonomous grounding capabilities through improved understanding of the processes by which human designers ground robot minds.

Grounding Oriented Design offers guidelines and processes for iteratively decomposing a robot control problem into a set of collaborating skills, together with a notation for representing and documenting skill designs. Grounding Oriented Design consists of two main phases: basic-design which involves constructing a skill-architecture, and a detailed-design in which a skill-architecture is used to design the agent's representation and decision-making processes. A groundedness framework[143] is used for describing and assessing the groundedness of either the complete system or of individual skills. Examples of the methodology's use and benefits are provided, while suggestions for future work are discussed.

Chapter 1

Introduction

This section describes:

- The author's motivation for writing this thesis.
- The research problem the thesis addresses.
- The need for a methodology to address the research problem.
- The scope of the research problem addressed by the thesis.
- The objectives of the thesis.
- The research contribution of the thesis.
- The scientific method employed.
- An outline of the remainder of the thesis.

1.1 Motivation

Motivation for this thesis is driven by two factors: firstly, my interest in the philosophical, theoretical and long-term issues arising from the grounding problem; and secondly, my experience as not only a professional software developer, but more importantly, my experience in both developing software for controlling mobile robots, and managing a team of programmers as the software development leader of the successful "UTS Unleashed!" robot soccer team[1]. In RoboCup, a world removed from the philosophical musings of Searle's[112] Chinese Room[2], the problems we face in grounding

[1]Information on UTS Unleashed! can be found at http://www.unleashed.it.uts.edu.au/, while information on the robot soccer competition (RoboCup) can be found at http://www.robocup.org

[2]For those not familiar with Searle's Chinese Room thought experiment, it is discussed in Section 2.1.2.

autonomous mobile robots are very real. Accurately perceiving the robot's simple "world" is an enormous challenge. Robots hallucinate (for example, seeing imaginary soccer balls), while sometimes failing to detect a ball which is in the robot's line of sight. Systems tend to be troublesome to debug (i.e. to fix), as perceptual input is uncontrolled, always changing, and difficult to exactly reproduce. Physical actions tend to be clumsy and lacking in dexterity in comparison to those found in the natural world, with the computer programs usually representing very little about the action and its consequences - rather, all we usually embed is the procedural knowledge of how to execute it. This is symptomatic of design in general - the programs know very little of what they are doing, for instance that "it" (the program and the hardware it is controlling) is even playing soccer. Important events to us, like scoring a goal, are often not even detected or represented by many systems (and, perhaps surprisingly, nor are they necessary to play soccer).

I am interested in artificial intelligence, cognitive science, robotics, and robot soccer because, in the longer term, I hope research in these fields will lead to the development of controllable, intelligent artifacts. Perhaps naively, I dream of a world in which artificially created machines do all the work for us, and we can sit around and relax, free to spend our time however we please. In this utopian world, the machines are called "agents", and if they are physically embodied "robotic agents". Agents operate on their own as if they were a surrogate self, continually making the "right" decisions and performing the "right" actions, even though they are operating in novel, complex and uncertain real-world situations. They work without human supervision, and only seek further instruction or help when it is absolutely necessary. For these agents of the future to become reality rather than fantasy, they will require a degree of artificial intelligence.

Intelligence, however, has proved very difficult for researchers to define. So far, "human-like" intelligence has also proved impossible to implement. For artificial agents, their intelligence ultimately rests with the algorithm (i.e. the program) that controls the agent. Such programs interpret the input from sensors (such as cameras, microphones, buttons, etc), and then (hopefully) activate the appropriate effectors (for example, motor positions, software commands, etc) which produce outcomes (e.g. behaviours) which conform to the designer's expectations. Unfortunately for agent developers, agents - being at least partially autonomous entities - need control programs that can respond to *future* environmental conditions, and thus there is always a degree of uncertainty regarding the exact environmental conditions the agent will experience. Herein lies a fundamental problem for agent developers, as developers must anticipate the agent's future environmental conditions, and design robust programs which endow the agent with all the "intelligence" to cope with novel experiences as they arise. The more uncertain the future environmental conditions, the more difficult the problem. For autonomous robots, operating in the freedom of the real world, the degree of uncertainty can be very high.

Our inability to develop agents which can adapt, as can many biological agents, to changes in the real world is illustrated by how many of the more famous "successes" of robotics and artificial intelligence have avoided, rather than deal with, autonomous perception and decision making in uncertain real-world environments. For example, industrial successes with robotics have generally been limited to factory automatons, which operate in highly controlled (and thus certain) environments, performing repetitive procedures with little or no sensing of the environment. Famous robots that have operated in real-world dynamic environments, such as the Mars Rover explorers[3], avoid the problem through sacrificing autonomy for teleoperation - in other words, humans interpret data streamed back to Earth by the robot, and then execute many of the difficult or "intelligent" decisions for the robot by remote-control. Lastly, arguably AI's most famous single achievement was when a supercomputer called Deep Blue[4] defeated the then reigning chess world champion Garry Kasparov. However, the chess program operated in a virtual world, had minimal learning capabilities, and perceptual tasks such as recognising pieces and their locations were handled by a human operator - the program's interface to the outside world. The only uncertainties for the chess program (or perhaps more importantly - for its programmers) are the opponent's future moves, and in the discrete, structured world of chess, the short-term possibilities and their consequences can be calculated.

To build robotic agents that operate in uncontrolled, real-world, every-day situations we need to develop agents that can adapt to changes in the environment in a manner similar to ourselves. In this thesis I will argue the ability to interpret new experiences *meaningfully* is crucial to the development of robust adaptive behaviour - i.e. to enable robotic agents to cope with novel situations for which have not been explicitly programmed. However, how do we create such a capability? How do we implement it in a computer program? It is an open problem.

1.2 The Research Problem: Grounding

The term "grounding", in the context of artificial intelligence, cognitive science, and communication, is related to the concepts of *meaning* and *understanding*. The term grounding has been applied loosely and generally to many different (but related) topics, such as how words get their meanings[72, 32, 60], how human participants in conversation reach shared meaning and understanding[44, 43], similarly how participants in human-computer interaction reach shared meaning and understanding [18, 27], how artificial agents can generate (or evolve) their own meaningful languages[120, 123, 118, 119], how the arbitrary symbols of a computational system become intrinsically meaningful[67, 124, 128], how agents relate beliefs to the external world[108], how to "anchor" internal representations of physical objects[49], and how designers can create meaningful ("physically grounded") behaviour

[3]http://marsrovers.jpl.nasa.gov/home/
[4]http://www.research.ibm.com/deepblue/

for robotic agents[21, 23, 24, 22].

Most of the diverse literature cited above has diverged mainly from two different usages of the term "grounding" within cognitive science, namely Harnad's "symbol grounding"[67], and Clark and Schaefer's[44] "common ground". Harnad's symbol grounding concerns how the symbols of a symbol system become intrinsically meaningful to the agent manipulating the symbols, with Harnad likening the problem to the task of trying to learn Chinese from a Chinese dictionary, but without understanding a single word of Chinese to begin with. The task appears impossible, as each meaningless Chinese word is defined with other meaningless Chinese symbols. Harnad concluded the meaning of symbols must be grounded in the agent's experience of the world. Over the last 15 years the symbol grounding problem has been cited by a large body of research which attempts to find computational mechanisms for modeling and explaining symbol meaning (often linguistic symbols) through lower-level non-linguistic sensorimotor representations, i.e. by mapping high-level "symbolic" representations to lower-level, "sub-symbolic" representations of the agent's experience in the world. On the other hand, Clark and others[44, 43] used grounding to describe the process of reaching a state of mutual understanding regarding what is "meant" during communication (to facilitate collective action)[5]. This version of grounding, originally defined as a model of communication between people[44], has been generalised to include different forms of electronic communication media[43], and more recently in the domain of human-computer-interaction[18, 95].

1.2.1 Grounding for Practical Robotics

Grounding, and its related issues of meaning and understanding, are deeply rooted philosophical problems, whose implications for robotics are not yet fully comprehended. While philosophical issues related to grounding are discussed in Chapter 2, our interest is in practical robotics, and in particular, developing control programs to allow robots to accomplish specific tasks. To avoid getting "bogged-down" in definitions of grounding that are based on meaning and understanding, grounding is viewed as:

> Grounding is the process of embedding an artifact in an environment to serve a particular purpose. In the case of robotic agents, it involves enabling that agent to be "in touch" (colloquially speaking) with the state, and nature, of the environment. Grounding is related to the process of perception, and involves correctly perceiving, conceiving, and responding to relevant aspects of the world. Grounding, however, is both task-specific and "body" specific - different agents, with different sensorimotor capabilities, performing different tasks in different environments, will need to perceive, conceive, and respond to

[5]For example, during a conversation listeners may nod their head to signify understanding, while speakers may repeat or rephrase their utterances.

different changes in the world with different levels of accuracy or acceptable error. Thus, we assume there can be a degree of error between an agent's beliefs about the world, and the state of that world (which may also be unknown to us). The term *groundedness* is used to describe how *well* an agent is grounded, with groundedness being graded and multi-dimensional.

1.2.2 Why is grounding important?

Meaning is important to us - it is clear that our thoughts are about something, i.e. we, the "thought users", attribute both meaning and content to our thoughts. The lack of understanding, meaning, and common-sense reasoning exhibited by computational programs has been one of strongest areas of criticism directed towards proponents of artificial intelligence [112, 55]. The symbol grounding problem has been described as one of the major outstanding problems facing artificial intelligence[56].

At a practical level, grounding effects all artificial systems. All artificial systems are grounded in some way, from pocket calculators to autonomous robots. However, both pocket calculators and autonomous robots are grounded by their designer - that is, we find meaning and structure in the world, and write programs to take advantage of it. Most of the intellectual work in AI is done not by programs but by the program's writers, and all the work involved in specifying the meanings of programs is done by people - not programs. Thus, robotic agents are typically grounded through their designer's understanding of the problem - i.e. the designer's understanding of the agent's capabilities, the environment in which the agent must operate, and the task the agent must perform - rather than through any agent-centric understanding of the environment, task, or themselves. The consequence of this approach is that the knowledge embedded in a program is the designer's solution to the current problem, and not the creative ability to solve the next problem that might arise. Due to the uniqueness of every robot-task-environment relationship, the implementation of robot control programs is a laborious, ad-hoc process, in which programs are hand-built and rely upon expert domain knowledge and heuristics. Consequently, program development is highly iterative and scaling up is difficult.

For robots to respond appropriately to novel and unforseen situations (situations for which they have not been explicitly programmed), novel situations need to be interpreted meaningfully by the agent - and not just by the designer *a priori*. However, even correctly classifying a situation, event or "thing" as being new or unknown is a difficult task in itself[80]. This "programmer's dilemma" - of how to program for an uncertain, changing future - has, over time, led to calls for new approaches to development, such as behaviour-based robotics and developmental robotics[6] - paradigms which place an emphasis on minimising innate knowledge, while maximizing learning and program plasticity

[6]These issues are discussed in Chapter 3.

during the robot's operation - the common aim being to write programs that are more adaptable to change in the world.

The grounding problem arises in any system which relies upon beliefs (however implicit or explicit) regarding the state, nature or behaviour of the world for the purposes of decision-making. Thus, there exists a need for a general solution to the grounding problem to replace the ad-hoc, case-by-case approaches which rely on domain knowledge and heuristics inflexibly embedded in the program. The lack of a general solution to the grounding problem is evident in our inability to develop robust robotic agents that can operate in dynamic, complex and uncertain environments (i.e. the real world). Instead agents are tailored for narrow, specific niche tasks in environments which are often tailored to assist them (e.g. markings are placed to assist in navigation, or lighting is controlled to assist with vision processing). Moreover, grounding is important because understanding this process will impact not only how we build robots, but also provide insight into how we find and use "meaning" in the world. A solution would entail a theory of meaning - how to produce it, how to use it, and what it is. Such a solution will have an impact in many disciplines - machine "understanding" (or semantics) is at the core of problems such as natural language processing, computational linguistics, information retrieval, and perception tasks such as computer vision.

1.3 The Need for a Methodology

The grounding problem has generated a large body of research (as we will see in Chapter 2). Most of this research is driven by the goal of developing autonomous grounding capabilities for agents - i.e. the ability to perceive, model and represent the world through interactive experience of it. Grounding is a practical problem with significant implications, yet it is rarely considered by robotics developers as a problem *per se*. The fact that all robotic systems are grounded through human design on a case-by-case, ad-hoc basis has been overlooked. While the ability to develop artificial systems that can ground themselves is a long-term goal of grounding research, we posit that a stepping-stone to such a development is an improved understanding of the processes by which human designers ground robot minds. Therefore, a methodology - being a systematic process - can be an initial step towards automating a general-purpose grounding process.

There are no methodologies concerned with grounding[7], despite the manual creation of representation being one of the largest costs associated with deploying robots[136]. As current grounding approaches tend to be ad-hoc, and reliant upon human knowledge, the success of each project is dependent upon the quality of the programmer and their knowledge of the domain. Methodologies, in

[7]There are grounding "frameworks", which are discussed in more detail in Chapter 3.

contrast, offer a systematic way of doing things. For example, while software design is a creative process, it is not devoid of structure, and software development methodologies are intended to facilitate systematic development of software, with the aims of minimising risk and improving product quality. While there are numerous software design methodologies (e.g. Agile[85], Extreme Programming[8]), there exist few specialist methodologies for designing software to control autonomous robots (except for [20, 136, 25]). Software for controlling autonomous robots is posed with unique difficulties (as opposed to "traditional" software), such as uncertain information (e.g. perception), uncontrolled and unpredictable environments, and the need for the software program to adapt to and respond appropriately to unforeseen circumstances. Thus, a grounding methodology, while providing insight into how the human mind grounds robotic systems, can also be of immediate benefit by improving the quality of current and newly developed robotic systems.

1.4 Scope

The implications of the grounding problem are wide-ranging - at one extreme they concern philosophical issues such as intentionality and consciousness[112], while at the other we have technical problems related to the classification and labeling of sensorimotor data[67, 49] which affect our ability to build artificial perception and language systems. While this thesis does discuss in detail these wide-ranging issues, the scope of the grounding methodology is software design for controlling autonomous mobile robots. The problem of building software for controlling autonomous mobile robots is treated as a technical, engineering problem. We assume software robotic agents are built for a purpose; i.e. to satisfy a set of requirements. We also assume quality and productivity are issues of concern (e.g. system verification, error minimisation and detection). The methodology presents a set of steps to guide the designers of "robot minds" through the problem decomposition process, together with a modeling notation for representing designs, with the aim of developing optimal solutions for software control problems. The methodology provides a conceptual means for modeling robot minds in terms of decisions and the knowledge required to make those decisions, with a level of detail that makes implementation in in a software programming language straightforward. Thus, the methodology is concerned purely with software - the methodology does not cover issues related to robot physical construction, such as hardware selection. Issues related to environmental tailoring (changing an agent's environment to help ground that agent) are considered.

[8]http://www.extremeprogramming.org/

1.5 Thesis Objectives

There appears to be no silver bullet for the grounding problem[128, 121], nor does this thesis provide one. The objectives of this thesis are:

1. To integrate, critique and relate existing grounding literature to the development of algorithms for controlling 'everyday', autonomous robots. The first objective of this thesis is to consider how the grounding problem effects real-world robotic agents, and why we need to solve this problem.

2. To develop a framework and methodology for grounding robotic agents. The methodology should provide a structured process for designing and representing both the decision-making processes and knowledge required for robot control problems, with the resultant designs being capable of comparison and straightforward translation into software solutions.

3. To explore how the grounding methodology and framework can be used for designing and comparing long-term solutions to the autonomous grounding problem (the problem of how to build artificial systems which can ground themselves).

1.6 Contribution

The major contributions of this thesis are:

1. A critique of existing grounding-related literature, in which implications of grounding with respect to the design of software for controlling autonomous robots is considered. The critique argues that there are two main grounding subproblems, namely the problem of relevance (selecting what to represent), and the problem of reference (maintaining that representation over time with respect to a changing world). It is also argued that grounding is graded and multidimensional, and that the dimensions (the different qualities of an agent's groundedness that are important) will vary with respect to each task, environment and agent relationship.

2. Grounding Oriented Design (Go-Design) - a grounding methodology to assist with designing and documenting software for controlling autonomous mobile robots. A diagrammatic notation is presented for modeling both the decision-making processes and the knowledge upon which those decisions rely. The notation allows us to model and document grounding "solutions", independent of an implementation language, but in a sufficiently detailed manner so that translation to an implementation language is a straightforward process. The diagrammatic modeling notation is not only capable of efficiently and expressively modeling robot

control problems, but also a wide variety of software problems in general. The benefits of the methodology include improved predictability in terms of deliverables and minimisation of project risk through the reduction of ad-hoc practices. An example of how to use the grounding methodology and framework is provided, and the benefits of using the methodology illustrated. Note - the methodology is not an algorithm for agent software design, and it cannot generate provably optimal agents.

3. Incorporation into the Go-Design methodology a framework for comparing and evaluating the *groundedness* - or quality - of representations. Groundedness, for many practical problems, can be measured in a straightforward manner. Just as the flights in an airline reservation database should correspond to real flights, a robot's beliefs about the state of the world can be verified. For example, consider, a soccer robot's representation of a soccer ball - properties of the agent's model such as the ball's distance and location relative to the robot can be empirically measured. Likewise, representing different aspects of the environment can facilitate different decisions, and consequently different behaviours. The grounding framework is used in conjunction with the grounding methodology to specify the required groundedness of systems, and for producing test-cases.

4. A discussion of new, practical and promising approaches for overcoming the grounding problem, including the use of ontologies and the semantic web[116].

1.7 Scientific Method

The methodology presented is essentially an engineering tool that draws on research in software engineering, robotics software development, and knowledge representation. It is an example in constructionism, a proof of concept, a synthetic methodology in which understanding is gained through building[101]. The methodology has been refined through stepwise improvements over a considerable period of time, during which time the methodology was used to not only design systems, but to also model a wide variety of existing complex robotic systems.

Unfortunately, comprehensively evaluating the "worth" of a methodology using comparative studies presents time and resource constraints. An ideal way to evaluate a new methodology would be to run numerous side-by-side trials with different experimental groups, such as developers using the new methodology (in this case Go-Design), developers using no structured methodology, and developers using other comparable, existing methodologies. This is obviously beyond the time and resources available in a PhD program, since the development and testing of the methodology itself has taken more than three years. So instead, this thesis will:

1. Identify the current gaps in software development methodologies for robotic systems.
2. Analyse the grounding problem and its impact on software development for robotic systems.
3. Present the methodology.
4. Provide practical examples of how it can be used to develop effective software solutions for complex robotic systems.
5. Compare the methodology to existing approaches and identify its advantages over current techniques.

1.8 Thesis Outline

- Chapter 1 is the current introductory chapter. It introduces the research problem, situates it with respect to existing research, provides a brief overview of our approach to the problem, and then details the remainder of the thesis.
- Chapter 2 includes a literature review of the grounding problem, beginning with Searle's Chinese Room argument[112] and Harnad's symbol grounding[67], but then detailing subsequent related research applications (e.g. language grounding and computer vision) and approaches (e.g. various hybrid approaches in which symbolic systems are integrated with "sub-symbolic" learning algorithms).
- Chapter 3 considers the practical implications of the grounding problem for the designers of robot minds - i.e. the "programmer's perspective" on grounding. Different aspects of the grounding problem are identified, such as grounding-by-design and autonomous grounding.
- Chapter 4 provides an overview and an introduction to Go-Design, and discusses the methodology's design considerations.
- Chapter 5 presents the first stage of Go-Design - context-level analysis, which concerns understanding the nature of the current robot control problem.
- Chapter 6 presents the first step of the methodology's design process, called "basic design". Basic design involves the iterative decomposition of a robot control program into a modular "skill architecture".
- Chapter 7 presents the second stage of Go-Design - "detailed-design". Detailed-design involves taking a skill architecture, and identifying the representation, decision-making processes and level of groundedness required to implement each skill.

- Chapter 8 provides a discussion and summary of the thesis's major findings, including limitations of the methodology, while presenting plans for future work.
- Lastly, Appendix A contains a "step-by-step" guide to Go-Design - that is, a summary of the important steps involved in Go-Design.

Chapter 2

Grounding: Approaches and Research Areas

The concept of "grounding" in cognitive science is perhaps best described by the philosophical arguments of Searle's (1980) "Chinese Room" and Harnard's (1990) "symbol grounding". Both are concerned with the nature of meaning and understanding, and whether machines will ever be capable of "human-like" understanding (whatever that may be). In this chapter I present an overview of Searle's and Harnad's work, as well as a survey of the subsequent literature which followed. I will argue that many different attempts to describe or overcome the so-called grounding problem, are actually attempts to describe or overcome distinct sub-problems of the grounding problem, and thus a coherent framework for describing how these sub-problems interact is lacking. In doing so, I will cover some of the "deep" problems, such as meaning, intentionality and understanding, while describing how grounding literature concerning robotics generally addresses specific, practical issues concerning problems of sensory interpretation, perception, or just a general (if albeit limited) "understanding" of the world or some aspect of it.

2.1 Grounding: What is it?

The term "grounding" in the cognitive science, artificial intelligence, and robotics literature is used in many different but related ways. For example, there is Harnad's symbol grounding[67], Brooks's physical grounding[21, 23, 24, 22], Vogt's[133] "physical symbol grounding", representation grounding[36], theory grounding[104], double grounding[99, 98], human grounding[81], "common ground" grounding[44, 43]), or grounding with respect to the evolution of languages (e.g. [117, 118, 123, 119, 122, 120]). When reading these different papers, you inevitably ask yourself a straightforward question - are these researchers talking about the *same* thing or *different* things? Are these apparent variations of the grounding problem actually the same problem in different guises? If not, what are the different problems? In this section we will survey the different interpretations of grounding[1].

2.1.1 Layman's grounding

In everyday English, to say someone, or something, is (or is not) grounded can have different meanings. It could mean that a plane can't take off, or that a child's parents have confined them to their bedroom. However, to say a person is not grounded could also mean that their understanding or beliefs about the world (or a particular topic) are incorrect, irrelevant, or even delusional. In contrast, a grounded person is indeed the opposite - for example, "the mechanic has a solid grounding with regards to truck engines" implies the mechanic has either experience, or a thorough understanding, of the mechanics of truck engines. It is this type of grounding - loosely described as "understanding" of the world - that we are concerned about in this thesis.

2.1.2 Searle's Chinese Room

Searle's[112] Chinese Room is a thought experiment which argues against the possibility of "strong AI". According to Searle, supporters of strong AI support the physical symbol system hypothesis (PSSH). The PSSH, formulated by Newell and Simon[97], states that "a physical symbol system has the necessary and sufficient means for intelligent action", implying that computers, when we provide them with the appropriate symbol-processing programs, will be capable of intelligent action. Proponents of strong AI support the idea that cognition is computation, mental states are computational states, and that a "mind" can be created with the "right" computations, i.e. the same computations that the human brain performs. Thus, according to supporters of strong AI, any machine performing the "right" computations will also have a mind.

In Searle's thought experiment, a version of a Turing Test[2] is offered, in which the reader is asked to consider two different perspectives. Central to this thought experiment is the Chinese room - a box-like room whose only interface to the outside world is through small slots through which pieces of paper can be passed to and from the world on the outside of the box. On one hand, we have a Chinese man outside the room, passing in notes written in Chinese and receiving sensible, handwritten responses written in Chinese passed back through the slots in the wall. From the Chinese man's perspective on the outside of the room, he is having a written conversation with

[1]For alternative summaries of grounding literature see [134, 128, 145, 124].

[2]The Turing Test[131] is a famous intelligence test for artificial intelligence. The test involves a "blind" interview in which the interviewer addresses questions both to a computer and to a human being. If, after a period of time, the interviewer is unable to distinguish between the human and the machine, the machine is considered "intelligent".

another Chinese person trapped inside the room. In contrast, the reality of the situation is that there is a man inside the room but he does not know anything at all about Chinese - not a single word. Instead, the man inside the room is able to follow a procedure (in his native language) in which he uses the incoming patterns of meaningless symbols (i.e. the Chinese) to find another set of meaningless symbols. He then writes the new symbols down and passes the note back out, creating the appearance of being able to understand Chinese, while the reality is quite the opposite.

Searle argued that the man inside the room is in fact like a computer, following a set of rules and procedures but not understanding or knowing what it is doing. Searle questioned whether a machine will ever be able to "understand" (and indeed "think") as does a human being, arguing that machines will never be capable of human-like intelligence because they lack *intentionality* and *understanding.* However, Searle did not conclude that the creation of intelligent machines is impossible - rather, that artificial intelligent machines will require the same "causal powers" possessed by human brains.

2.1.3 Harnad's Symbol Grounding

In 1990, Harnad[67] coined the term "symbol grounding" to describe a problem related to Searle's Chinese room[112]. Harnad likens the symbol grounding problem to trying to learn Chinese from a Chinese dictionary alone, where every word is defined in terms of other Chinese words. Thus, without any knowledge or experience of Chinese, as each Chinese word is defined in other *meaningless* words, the task is seemingly impossible. At some point, Harnad argues, the meaning of at least some words must be grounded in experience. Harnad defines the symbol grounding problem as:

> "How can the semantic interpretation of a formal symbol system be made intrinsic to the system, rather than just parasitic on the meanings in our heads? How can the meanings of the meaningless symbol tokens, manipulated solely on the basis of their (arbitrary) shapes, be grounded in anything but other meaningless symbols?"[67]

Harnad's concludes that at least some symbols of a symbol system must be directly grounded through experience, proposing a candidate solution in the same paper, in which neural networks are suggested as a means of learning the relationships between (arbitrary) symbols and the subsymbolic (and "invariant") sensory features to which they relate. Thus, Harnad attempts to solve the grounding problem by finding the patterns of sensory features that correlate to symbols, or vice versa - the apparent argument being that because the machine's symbols are "connected" to (or influenced by) sensory experience the symbols are now (somehow) intrinsically meaningful to the machine.

2.1.4 Brooks' Physical Grounding

"There is an alternative route to Artificial Intelligence that diverges from the directions pursued under that banner for the last thirty some years. The traditional approach has emphasized the abstract manipulation of symbols, whose grounding, in physical reality has rarely been achieved." (Brooks[21]).

Robotic agents require an appropriate description of how to act in the world - but not necessarily an explicit description of the world itself. Brooks[21] response to the symbol grounding problem (and other problems related to traditional AI) was to argue that traditional, explicit symbolic representation "just gets in the way". According to Brooks, "the world is its own best model ... the trick is to sense it appropriately and often enough". Instead of building top-down models of the world, Brooks believes that intelligent behaviour can *emerge* in a bottom-up direction from a collection of cooperating "behaviours", with each behaviour tightly coupled to sensors and effectors. Using a methodology to build, integrate and layer these behaviours, known as the "Subsumption Architecture"[3], Brooks described his approach as part of "Nouvelle AI" (or behaviour-based robotics), all of which is based upon the *physical grounding hypothesis.*

"Nouvelle AI is based on the physical grounding hypothesis. This hypothesis states that to build a system that is intelligent it is necessary to have its representations grounded in the physical world."

"Only through a physical grounding can any internal symbolic or other system find a place to bottom out, and give meaning to the processing going on within the system... The world grounds regress."[22].

Nouvelle AI emphasised that physically grounded systems require real robots, built bottom up, so that "high level abstractions have to be made concrete"[21]. The development process involves the hand-crafting of individual behaviours (i.e. functions, algorithms, etc), with some new, emergent behaviours arising from the interaction of behaviours. Brooks argues that because of the physical grounding hypothesis, "traditional symbolic representations" are no longer necessary, and the symbol grounding problem is avoided.

2.1.5 Representation Grounding

A symbol designates or denotes something else[97], while a "representation", by definition (i.e. to *re-present*), stands for something else (the referent). In defining the symbol grounding problem, Harnad[67] describes the problem in relation to "symbolic AI". However, what is a symbol?

[3]The Subsumption Architecture is discussed in more detail in Section 3.3.3.1

Steels[121] comments the use of the term "symbol" in artificial intelligence has probably created "the greatest terminological confusion in the history of science". Steels argues this confusion arises from differing uses (or meanings) of the term symbol by researchers with different backgrounds, i.e. philosophers, linguists and computer scientists use "symbol" in different ways. For example, a computer programming language is itself symbolic, yet when a neural network is implemented in a computer language using such symbols, the neural network is not considered by cognitive scientists or philosophers to be "symbolic" - rather, it is considered to be "subsymbolic". Thus, "symbols" in the context of symbol grounding literature usually refer to logical propositions that are used for reasoning about the world (e.g. "Sydney is in Australia", "Australia is in the southern hemisphere", so therefore "Sydney is in the southern hemisphere"). However, we take the view that a symbol is merely a representation (i.e. they stand for something else), and need not be a logical proposition. For example, other common examples of representation in robotics (and software in general) include variables, classes, databases, and so forth - i.e. any data structure, internal state or memory. Several authors[36, 90, 102, 145] have commented that the symbol grounding problem is not limited to "symbols", but to representation in general. For example, MacLennan[90] describes the grounding problem as "how do representations come to represent", while Pfeifer and Verschure[102] describe a "general grounding problem" which applies to knowledge "structures", rather than just "symbols".

2.1.5.1 Analog Computation

Harnad frequently makes the distinction between symbolic and analog computation (e.g. [67, 68]).

> "By 'computation' I mean symbolic computation: the manipulation of physical 'symbol tokens' on the basis of syntactic rules that operate only on the 'shapes' of the symbols (which are arbitrary in relation to what the symbols can be interpreted as meaning), as in a digital computer or its idealization, a Turing Machine manipulating, say, '0's and '1's. What I will say about symbolic computation does not apply to analog 'computation' or to analog systems in general, whose activity is best described as obeying differential equations rather than implementing symbol manipulations"[68]

The distinction between analog and digital is analogous to the distinction between continuous and discrete. Input is input is input, whether it be continuous or discrete. Consider the following practical robotics example: on the Sony AIBO robot some button sensors provide a continuous analog signal which correlates to the force applied to the button, yet other button sensors on the same robot only provide a discrete boolean state (i.e. "is pressed" versus "is not pressed"). For the cognitive agent controlling the AIBO robot, do the representations of analog button presses and digital button presses need to be grounded any differently? By raising the distinction of analog versus digital computation Harnad is in effect raising a separate issue: the role of discrete or continuous representational processes in human cognition. MacLennan[90] comments that "grounding is just as important an issue for continuous (analog) computation as for discrete (digital) computation.".

2.1.6 Autonomous Grounding

Many researchers believe grounding to be a technical problem which involves connecting symbols with experience (and vice-versa) *autonomously*. For example, Prem[103] argues that symbol grounding is "*automated* model construction". Likewise, Steels[121] argues if "someone claims that a robot can deal with grounded symbols we expect that this robot *autonomously* establishes the semiotic map that it is going to use to relate symbols with the world". For Taddeo and Floridi[128] symbol grounding is the problem of how an artificial agent can "*autonomously* elaborate its own semantics" through interacting with its environment[4].

2.1.7 Summary

In this section we have seen that the grounding problem concerns how to connect symbols and representations with the world. In the 1990s symbol grounding could be seen as an attempt to bridge (or connect) the fields of connectionism with traditional (symbolic) AI. However, there are a range of different definitions and interpretations of grounding - some which emphasise that the grounding process must be autonomous, those that emphasise intrinsic meaning (or first-hand semantics), and those that focus on connecting symbols to categorisations of sensorimotor experience.

At its most philosophical, grounding concerns "deep" questions such as the meaning of meaning, and whether machines will ever be capable of "meaningful thought" or consciousness in a manner similar to human cognition (e.g. as in the Chinese room[112]). One aspect of this problem is how to make symbols meaningful to the symbol user (i.e. a computer program) and not just the designer - a common view being that this can somehow be achieved (at least in part) by connecting symbolic representations with "subsymbolic" representations of experience (e.g. iconic and categorical representations). Thus, the symbol grounding problem according to most technical research, concerns how to create and "connect" a "symbolic" representation with categorical representations of sensorimotor experience, with connectionism being just one option for categorising and classifying experience.

In the next section we will consider *meaning* - a term widely used in defining grounding, but rarely defined or elaborated upon.

[4]My italics.

2.2 Meaning

"The symbol grounding problem concerns how the meanings of the symbols in a system can be grounded (in something other than just more ungrounded symbols) so they can have meaning independently of any external interpreter" (Harnad, 1993[68]).

"Grounding does not equal meaning, however, and does not solve any philosophical problems" (Harnad, 1993[70]).

Harnad's symbol grounding assumes that for a symbol system to escape the Chinese room argument[112], some of the symbols of that system must be grounded (i.e. have meaning) in something other than more ungrounded, meaningless symbols. Harnad concludes that some symbols must be grounded in subsymbolic, categorical perceptions of reality, offering neural networks as a means for learning such relationships. In this section we examine the impact of grounding on theories of meaning, and the implications for autonomous robotics.

2.2.1 Grounding and Meaning

"To ask how a symbol is grounded is to ask how it becomes meaningful" (Glenberg *et al.* [59]).

"The meaning of a representation can be nothing but a representation. In fact, it is nothing but the representation itself conceived as stripped of irrelevant clothing. But this clothing never can be completely stripped off: it is only changed for something more diaphanous. So there is an infinite regression here"(Charles Peirce[100]).

Grounding has become synonymous with meaning. However, meaning is a poorly understand concept that has intrigued philosophers for centuries, as have other fundamental issues related to cognition and consciousness. What are meanings? Where do they exist? Is grounding *really* a process of meaning giving?

2.2.1.1 Internalism vs Externalism

Searle's Chinese Room is an argument against "strong AI" - proponents of which hold that cognition is computation, i.e. a mind is the result of the right "program". Searle argues that cognition must involve more (i.e. semantics) than simply computation (i.e. syntax), by asking us to consider that given the right program, Searle himself could appear to be understanding Chinese, but the reality is in fact the opposite. The Chinese he produces is never translated to English, and thus Searle has no understanding of what he is doing. Searle's view of meaning is that of semantic internalism - that meanings exist "in the head" or mind of the meaning user.

In contrast, semantic externalism holds that meanings exist in the world. Putnam's[105] "Twin Earth" is a famous thought experiment supporting semantic externalism. The twin earth thought experiment asks us to consider a mirrored, twin earth somewhere in the universe where everything is exactly the same, except the chemical composition of water. On each world there are two identical beings, both who possess a concept about "water", but because the chemical composition of water is different on the two earths, Putnam argued the meanings of "water" are different, and thus meanings "ain't in the head".

2.2.2 Intrinsic Semantics

> "It is, for example, rather obvious that your thoughts are in fact intrinsic to yourself, whereas the operation and internal representations of a pocket calculator are extrinsic, ungrounded and meaningless to the calculator itself, i.e. their meaning is parasitic on their interpretation through an external observer/user."[145]

Harnad[67] asks how can the meaning of symbols be not "parasitic on the meanings in our heads"? For Harnad, the meaning of a symbol system requires more than relations between symbols - a symbol system is meaningless unless at least some symbols are grounded (related to or connected to) in something "other" than more meaningless symbols. Harnad describes grounding as a problem of creating "intrinsic meaning", or what has been called "first-hand semantics"[145]. Similarly, Bickhard[11] describes the problem of "designer semantics" - that the representations of any computer system (and thus robotic agent) is only meaningful to the designer of the system, and not the system itself - and the existence of this relationship (the encoding between a representation and referent by the designer) is not even made aware to the computer system.

2.2.3 A Theory of Reference

> "Symbol grounding is a new name for an older problem - the problem of providing a theory of reference for atomic formulae of a system of internal representation"[42]

Harnad's appeal to the Chinese Room and the notion of intrinsic semantics concern philosophical issues such as intentionality, consciousness and meaning. Harnad's practical solutions (connecting symbols with sensorimotor experience of the real-world entity referred to by the symbol), however, attack a very different problem - a theory of reference[42]. A theory of reference (or alternatively "correlational semantics"[103]) involves relating internal representations with external entities (e.g.

somehow connecting a symbol "John" with the real-world "John") - in other words, understanding how the atomic units of a language come to have meaning.

A large proportion of grounding-related research treats meaning as a problem of reference. For example, symbol grounding has been described as establishing the "direct correspondence between internal symbolic data and external real world entities"[4]; the problem of how "symbols should acqure their meaning from reality"[133], or the association of a symbol "with a pattern of sensory data that is perceived when the entity that the symbol denotes is seen, or tasted etc"[92]. Anchoring[49], a variation of the grounding problem, embraces the problem of reference - anchoring involves "maintaining the correspondence between symbols and sensor data that refer to the same physical objects"[51].

2.2.4 More than a theory of reference?

Harnad, however, acknowledges that referents aren't the sole source of meaning for symbols. For example, Harnard argues:

> "We know since Frege that the thing that a word refers to (its referent) is not the same as its meaning. This is most clearly illustrated using the proper names of concrete individuals (but it is also true of names of kinds of things and of abstract properties): (1) "Tony Blair," (2) "the UK's current prime minister," and (3) "Cheri Blair's husband" all have the same referent, but not the same meaning."[72]

Harnad's example illustrates the richness of meaning - meaning for a human being appears to involve more than simply relating internal symbols with external entities. One aspect of meaning used in Harnad's example is emotional or connotative meaning. Roy[108, 109] identifies three important aspects of meaning in language - emotional, connotative meaning (e.g. "my father gave me that cup - it has great meaning for me"), functional meaning (e.g. "this coffee is cold" can imply "get a hot coffee"), and referential meaning (as in a theory of reference, e.g. "I meant that one"). Cohen *et al.*[46] argue a representation is meaningful if it not only indicates a condition (i.e. a state) of the world, but if that indicator serves a functional purpose, such as informing action. Steels[121] makes the distinction between "m-representations" for human representations (e.g. thoughts, pictures, etc) and "c-representations" for computer-based representations to highlight the gap between the richness of human meaning, and the reality of representation use in computer-based systems (e.g. variables).

If we consider linguistic meanings of meaning, referential meaning (e.g. "I *meant* that one!") is just one component. Another aspect is *understanding* (e.g., as in "do you know what I mean?"). Thus, meaning is often defined circularly as how an event, action, word, etc is *understood*, and that conversely, to understand something is to know the *meaning* of it. For example, Barsalou[10] describes meanings as "people's understandings of words and other linguistic expressions". Likewise, it is this "understanding-based" notion of meaning that is appealed to by Searle[112] in his Chinese room argument - the man inside the room does not understand the Chinese symbols, and thus they are meaningless to him. Other linguistic usages of meaning include to imply *consequence* or causation (e.g. "that alarm means trouble", "friction means heat", or "lower costs mean lower prices"); *relevance* - the worth, value or significance of something to the something else (e.g. "the critic's opinions meant nothing to the author" or "her boyfriend meant a lot to her"); and lastly, *intention* - we use the term "meaning" with regard to discussing intent, design or purpose (e.g. "I meant 8am - not 8pm!", "I meant to go swimming this morning, but I overslept", or "that building that is meant for storage").

2.2.5 Summary

The grounding problem is related to meaning, but meaning is a concept which has troubled philosophers throughout history. Thus, trying to define grounding in terms of meaning is (perhaps) somewhat meaningless. However, two distinct aspects of meaning are appealed to in grounding literature - firstly, the idea that the grounding problem is the problem of creating intrinsic, first-hand semantics (or meaning) for an artificial computational system; and secondly, referential meaning - that the grounding problem can (at least partly) be solved by somehow connecting (in the right way) symbols with sensorimotor data. In the next section we consider in detail the different approaches to grounding, and consider the aspects of meaning they address. We will see that most grounding-related research, despite appealing to the problem of intrinsic meaning, is empirical and constructionist and attempts to solve the problem of reference.

2.3 Approaches to Grounding

A large body of multi-disciplinary research has been generated by the grounding problem. While grounding-related research varies in implementation and application, most approaches have focused on ascribing meaning through categorical perception. These approaches assume grounding can be achieved by linking "symbolic" representations to sensorimotor, "subsymbolic" representations that are "invariantly" correlated (or are "causally" related) with the real-world phenomena being represented. In other words, I know what a pizza means because I know what it tastes like, smells like, looks like, feels like, and so forth - and therefore, if we could do the same for a robotic system (using cameras, taste sensors, etc), the robotic system's meaning of pizza would also be grounded[92].

The underlying analogy (and assumption) behind these approaches is that the output data produced by a robot's sensors is somehow comparable to the sensations (and emotions) we feel (such as taste, touch, happiness, etc) - a form of "robot functionalism"[71]. What usually differs between approaches are the methods used for categorisation and classification. Neural nets have been suggested as one means of acquiring such "meanings" (e.g. [67, 106]), as have conceptual spaces[39], or perceptual features within a logical framework[49]. An inherent assumption in most approaches to grounding is that intrinsic semantics or meaning can be obtained through categorising experience. In this section we examine the different approaches to grounding artificial systems.

2.3.1 Harnad's Approach

In defining the symbol grounding problem, Harnad also offered a candidate solution. Harnad proposed that neural networks could be used to connect symbolic representations to the world, suggesting the use of connectionism for learning perceptual categories. Harnad makes the distinction between different types of representation. With symbolic representations, the form of the representation and form of the represented entity are largely unrelated or arbitrary (e.g. the numeral "2" and the concept it represents). In contrast, the structure of a subsymbolic representation is invariantly (causally) related to the entity it depicts (e.g. as in a map). Harnad's proposal involved three stages:

1. *Iconisation*: The process of relating sensory input to subsymbolic, "iconic" representations. Iconic representations are subsymbolic representations "which are analogs of the proximal sensory projections of distal objects and events". In other words, "icons" are representations which structurally resemble the thing they represent, e.g. in the way a map has a structural similarity to the part of the world it represents. Harnad uses the following example to illustrate - "in the case of horses (and vision), they would be analogs of the many shapes that horses cast on our retinas"[67]. Therefore, iconic representations have features, from which the next step - categorical perception - can learn the "invariant" features, so as to discriminate and identify categories of things in the world. Thus, iconic representations are essentially a sensorimotor stream, faithfully preserving the true "shape" of the world that is experienced by the agent.

2. *Categorisation.* The process categorising inputs based upon their similarities and differences. This step is performed by a neural network. Categorical representations are learned and innate feature detectors that pick out the invariant features of object and event categories from their iconic representations.

3. *Identification.* This process uses symbolic, or arbitrary, representations. Identification is the process of assigning a unique response (that is, a name) to a class of inputs, treating them as

equivalent or invariant in some respect. For Harnard, symbolic representation involves symbol strings describing category membership.

Harnad[67] also goes on to define a process by which new, higher-level symbols can be performed through the use of elementary (or directly) grounded symbols (i.e. symbols that are immediately grounded in sensorimotor experience), a process later described as "symbolic theft"[32]. For example, the concept of a "zebra" could be grounded (without ever experiencing one) by combining the concepts of "horse" and "stripes", as long as the symbols "horse" and "stripes" are grounded through subsymbolic representations (but more about this later in Section 2.3.3).

2.3.1.1 Neural Networks

Harnad's initial approach was a catalyst for a body of grounding-related research in which neural networks[5] are used. For example, Davidsson[53] argued for the role of machine learning as a "general" solution for symbol grounding (like Harnad[67]) suggesting that neural networks could enable agents to learn representations from sensorimotor experience). Some other examples of the use of neural networks for grounding-related problems include Sales and Evans[111], who ground linguistic symbols using neural networks. Spiegel and McClaren[115] use machine learning to ground the concepts of "odd" and "even". Riga *et al.*[106] use a hybrid model, consisting of both a self-organising map for classifying sensory information, together with a supervised neural network for category acquisition. Cangelosi[28] presents a framework for modeling language in neural networks and agent simulations, while in other research, Cangelosi *et al.*[33] use neural networks with autonomous robots to learn linguistic descriptions of actions and objects.

2.3.2 Categorical Perception

> "Neural nets may be one way to ground the names of concrete objects and events in the capacity to categorize them (by learning the invariants in their sensorimotor projections)... If neural nets turn out to be unable to do this job, other pattern learning mechanisms might still succeed"(Harnad[70])

A common theme in grounding literature is the meaning of representations is (at least partly) due to them being "connected" (in the right way) to the *categorisation* of sensorimotor experience[73]. Categorisation appears to be a key aspect of cognition - people form categories of many different kinds every day; examples include "cars", "people", "foods you refrigerate", "things to take to university" and so on. There are even categories of categories - for example, Barsalou[8] defines three types of

[5]The term "neural networks" (or connectionism) refer to neuron-based models of the human brain capable which are capable of learning.

categories that occur in everyday cognition - *ad-hoc*, *explicit definition*, and *no explicit definition*. Examples of ad-hoc categories are "birthday presents for your mother", or "things to take on a camping trip". Explicit definition categories have precise rules for inclusion. For instance, a triangle is a closed figure having three straight lines for sides and three angles whose sum is 180 degrees. In contrast, no explicit definition categories, as their name suggests, have no known explicit rule for category membership. Instead, members of this category have some associated properties that are generally true, but no universally so, of their items. An example of a no explicit definition category could be "sport". For instance, while sport generally implies physical activity and competition, many people would argue over whether activities such as chess, snooker or synchronised swimming should be classified as sport.

2.3.2.1 Machine Learning

Harnad[70] suggests that neural networks are just one type of machine learning algorithm that may capable of learning patterns in sensorimotor data, and thus forming categories of "experience". Other learning techniques that may be capable of the "how" of categorisation include self-organising maps[83], genetic algorithms[94], and geometric-based techniques such as conceptual spaces[57]. In the following subsections these machine learning techniques are briefly discussed. Those readers familiar with such techniques may wish to skip to Section 2.3.3.

2.3.2.2 Self-Organising Maps

Self-organising maps[83] (SOMs) involve a mapping of a high-dimensional input vector to a cell in a low-dimensional matrix based upon similarity measures of the input vectors with a model vector associated with each cell, thus the resultant matrix is organised (i.e. categorised) based upon the similarity (or conversely differences) of input vectors. Examples of SOMs use in grounding-related research include Blank *et al.*[13] who suggest how SOMs can be used for the life-long learning of an artificial agent, and Riga *et al.*[106] who use a self-organising map for classifying sensory information.

2.3.2.3 Genetic/Evolutionary Algorithms

Genetic algorithms are a type of learning algorithm inspired by evolutionary and genetic processes, using techniques such as mutation, inheritance, selection, and recombination (or crossover)[94]. Such algorithms typically involve the production of new generations of individuals within a population by combining or mutating the attributes of "fit" individuals, thus (hopefully) moving towards an optimal solution. In grounding research which utilises genetic algorithms, Swarup *et al.*[127] present a system in which agents learn ontologies to solve multiple tasks by using genetic algorithms to improve a particular representation structure known as "frequent subgraphs". The authors claim their approach is more flexible than compared to the use of neural networks. Other researchers have used genetic algorithms to train neural networks. For example, Law[86] has conducted research to develop grounded behaviours in which genetic algorithms evolved neural networks to produce wall following behaviour, while Nakisa and Plunkett[96] use genetic algorithms to train neural networks to learn representations for recognising speech sounds.

2.3.2.4 Conceptual Spaces

Gardendfors' "conceptual spaces"[57] is a theoretical framework for modeling concepts and, in particular, provides methods for modeling categorisation and similarity. Conceptual spaces model concepts using spatial and geometrical representations of cognitive dimensions such as colour, shape, pitch, and so forth. For example, sound perception could be represented by a two dimensional space, using the dimensions of pitch and volume. Points in a space can represent instances, while regions can represent categories. Common clustering techniques can be used to form and modify categories, while different distance metrics can be used for modelling similarity. As a conceptual space represents qualities of the environment independently of any linguistic formalism, it can be thought of as occurring prior to symbolic representation. Gärdenfors argues that symbolic, associationist, and conceptual levels of representation are all required for the construction of artificial agents, with conceptual spaces offering a bridge between the symbolic and connectionist modes of representation. This intermediate representation is likened to a bridge between subconceptual knowledge (where knowledge is yet to be categorised) and symbolically organised knowledge.

Chella *et al.*[38, 40, 39, 37] have applied conceptual spaces to the grounding problem[6]. For example, conceptual spaces have been used as a means of enabling robotic agents to understand visual scenes [38, 39], or to explore the "learning by imitation" development paradigm[37] (i.e. robots can learn new actions or movements by observing a teacher performing the action), which is achieved by modeling an action space. Lastly, Roy[107] suggests how conceptual spaces can provide a means to model context sensitivity in natural language processing.

2.3.3 Symbolic Theft and Sensorimotor Toil

An obvious criticism of the hypothesis that meaning is related to categorical perception is how do we explain for entities that can never be perceived because they are imagined[42], detached[58], or never existed? Harnad's reply[70] to such criticism focuses on a "peekaboo unicorn", a horse with a horn which vanishes without a trace before any entity with a grounding capability can ever perceive or

[6]Note - much of this work falls under the category of "anchoring" - a specialisation of the grounding problem which we will discuss in Section 2.4.3.

detect it (through either their senses or the use of other measuring instruments). Thus, unverifiable in principle, Harnad argues a concept of the peekaboo unicorn can be grounded, as long as (at least some) symbols used to describe it are grounded. This process of giving a "non-perceivable" concept such as the "peekaboo unicorn" meaning has been described as the *symbolic theft*[30, 32, 31] of meaning from *elementary symbols*[67, 70].

Given the role that categorical perception plays in cognition, it is a common argument that symbolic representations must be grounded bottom-up using categorical representations (for example [67, 70, 32, 31, 72]). By building categories that filter features of iconic representations, within category differences compress, while between-category similarity distances expand, allowing for reliable classification of category membership. The use of categories restricts the grounding problem to a set of elementary symbols, where each of these elementary symbols has a corresponding real-world object, event or relation[31]. The process involved in grounding elementary symbols in sensorimotor experience is described as *sensorimotor toil*[30, 32, 31, 35].

Cangelosi and Harnard[32] extend Harnard's original candidate grounding solution by introducing a new tier composed of complex symbols, formed from the symbolic theft of meaning from elementary symbols (e.g. the term "zebra" could be described in terms of "horse + stripes", with the terms "horse" and "stripes" being directly grounded, elementary symbols). Thus, with Cangelosi and Harnad's approach there are now four stages or representation processing:

1. Sensory data is converted to iconic representations.
2. Categorical representations are formed from iconic representations.
3. Elementary symbols are "connected" (or directly grounded) to categorical representations.
4. Lastly, complex symbols are formed from manipulation of other elementary or complex symbols.

While the theory of symbolic theft provides a plausible explanation for the grounding of abstract concepts, many questions are raised. For example, at what point is a symbol system grounded? What balance between directly-grounded elementary symbols and complex symbols (formed through symbolic theft) must there be? If we liken it to Harnad's[67] problem of trying to learning Chinese from a Chinese dictionary alone, is our understanding of Chinese grounded if we know a few words of Chinese to begin with, and thus use only a few Chinese words to ground all the new Chinese we learn?

2.3.4 Hybrid Systems

Harnad[70] uses the "peekaboo unicorn" example as evidence as to why a neural network alone is not likely to be enough for grounding, as he argues there has to be a symbolic level at which the higher-order categories formed out of propositional strings are represented. Therefore, Harnad argues that a grounding process must require both a symbolic component, and a component capable of learning the categorical representations of sensory invariants. In the relatively short history of AI two competing paradigms - connectionism and symbolic AI (symbolicism) - have both jostled for attention and argued over which is the more appropriate paradigm for mind-modeling and implementing intelligence[91]. Hybrid systems are strategies which combine neural networks with symbolic models such as expert systems, decision-trees, or case-based reasoning[126][7]. The motivation for the hybridisation of connectionism and symbolic AI is based on the assumption that while the technologies are different, they are also complementary. Symbolic AI is more transparent[8], easily semantically interpreted, and provides a high-level of problem abstraction, thus simplifying the process of encoding expert knowledge and decision-making rules. On the other hand, connectionist models show advantages with regard to learning capabilities[63].

Harnad's[67] hybrid system approach to the grounding problem is not alone. Cangeosi and others[35, 33] use a supervised neural network together with a symbolic system to enable robots to learn the names of actions through demonstration. Davidsson[53] uses neural networks to connect computer vision representations with sensory data. Greco *et al.*[65] use neural nets to categorize and name images of shapes, such as circles, ellipses, and squares. Sun[125] has developed CLARION, a cognitive architecture which focuses on bottom-up learning coupled to symbolic reasoning.

2.3.5 Cognitivism versus Behaviourism

As Harnad's[67] proposal that symbols must be grounded through experience was gaining momentum, concurrently in the 1990s the notions of embodiment, agent-environment interaction and situatedness began to receive more attention [1, 2, 21, 23, 24, 22, 6]. Appealing to notions of embodiment, situatedness and agent-environment interaction is a common argument used to deflect awkward questions related to how symbols and representations acquire meaning[54, 124, 41, 92, 5, 113, 15]. Proponents of embodiment agree that the symbols of a disembodied symbol system lack intentionality, with the meaningfulness of symbol systems derived through the human designer. However, proponents of emobodiment argue an agent can be grounded through sensory-motor *interaction* with the environment - as opposed to *passively* experiencing it.

Ziemke[145], in a literature review, describes approaches to grounding as subscribing to either the

[7]In contrast, *unified* strategies aim at developing symbolic capabilities using neural networks alone[126].

[8]A quality helpful with debugging (program fixing).

"cognitivist" or "enaction" paradigms of cognitive science. Cognitivists are described as supporting the slogan "cognition is computation", while "enactivists" emphasise embodiment, action, and distributed intelligence. Cognitivists attack the grounding problem in a way similar to that suggested by Harnad[67], by trying to establish "causal" connections between internal representations and external entities or object categories through "invariants" in sensorimotor data. Ziemke[145] states that while the cognitivist approach supports the idea of interaction with the environment (in principle), much of this research only attempts to ground representations in *sensory* invariants alone, thus neglecting much of what it means to be embodied. In contrast, the "enactivism" paradigm stresses "siuatedness" and "embeddedness", and views grounding not purely in a symbolic-sense, but also in a behavioural-sense, e.g. as in the physical grounding hypothesis[21]. That is, as enactivists are committed to agent-environment interaction, the grounding problem becomes a problem of finding the right "function" for that agent, task and environment. Consequently, enactivists view grounding as a system-level process - that is, grounding an agent involves finding the "right" control program or function that produces the "right" (i.e. "meaningful") *behaviours*, whereas cognitivists treat grounding as problem of making *representations*[9] meaningful. As Ziemke[145] points out, if we adopt the enactivist/behaviourist view of grounding, where does grounding start and end?

2.3.6 Developmental, Learning Systems

Another common way to claim that symbols are grounded, or more precisely intrinsically meaningful to a robot, is to say that a robot *learned* the meaning of symbols or representation through experience of (or interaction with) the world[13, 14, 45, 138]. A relatively new paradigm, developmental robotics[13, 35, 140, 138, 139] (or cognitive developmental robotics), is an approach to developing intelligence which emphasises lifelong learning and the modeling of human cognitive development. The aim of developmental robotics is to create artificial intelligences which exhibit autonomous mental development, in the same way human cognitive and behavioural development occurs through infancy to adulthood. Much of the practical motivations behind developmental robotics is to develop robots capable of learning new tasks that a human programmer does not anticipate at the time of programming.

One of the leading proponents of developmental robotics is Weng[138, 139, 140]. Weng argues autonomous mental development can relieve programmers from the burden of designing a robot's representation and control structures. One of Weng's robotic projects is called SAIL (Self-organizing Autonomous Incremental Learner). The goal of the SAIL developmental robot is to automatically generate representations and architectures. SAIL is capable of exploring the world on its own, building representations and categorisations by automatically deriving discriminating features, while

[9]Especially "symbolic" representations.

SAIL's learning process can be guided by human instruction and reinforcement by pressing a "good" button or "bad" button on the robot. Other examples of developmental robotics include the "robot baby" project[46, 45], which is an effort to model cognitive development on a robot platform, and that of Blank *et al.*[12, 13, 14] whose goal is to create robots that can discover concepts autonomously.

2.3.7 Summary

In this section different approaches to the grounding problem have been presented. All are concerned with how to make representation meaningful, though what constitutes representation and meaning is different under each approach. Hybrid systems ascribe meaning to a high-level symbolic reasoning system by connecting the symbol system to machine learning algorithms capable of forming categorical (subsymbolic) representations of sensorimotor data. In such systems, the process of symbolic theft can enable the construction of new concepts, including those that are abstract or imagined. Behaviourist approaches treat the grounding problem as a practical problem of finding the right "function" to control a robot's behaviour, and thus the representation being grounded is the entire control program itself. Lastly, developmental robotics embraces a life-long learning approach to robot development, and proponents of this approach argue that if a robot learns its own representations then such representations are grounded.

In the next section we will turn our attention to specific grounding areas, such as natural language processing, computer vision, information retrieval, and the semantic web. It is argued the grounding problem is a wide-ranging problem for which a general solution may have far-reaching consequences.

2.4 Grounding Research Areas

Grounding is a far-reaching, multi-disciplinary problem. In this section we consider different research and application areas relevant to the grounding problem, such as computer vision, language, the semantic web, information retrieval, and the general problem of robotic system development. We begin by discussing language.

2.4.1 Language

There is a large body of grounding-related research concerning language. Most grounding-related research in computational linguistics falls into the following categories:

- Understanding natural language - the task of creating machines that can understand human languages, either written or spoken.

- Evolutionary computational linguistics - an area of research which uses computation linguistics to not only help improve our understanding of the origins of language, but to also see how collections of robotic agents can develop their own languages.
- Psychological experiments with human subjects, which examine the role of perception, action and embodiment in grounding language.
- The process of establishing "common ground" (i.e. shared or mutual understanding) during communication between agents.
- So called "non-grounded" approaches[109], in which linguistic meaning is modeled as structural relations between symbols.

In the following subsections we will discuss each research area.

2.4.1.1 Natural Language Processing

Natural language processing (NLP) is an area of research devoted to developing techniques which allow machines to "understand" different forms of natural[10] language, such as spoken or typed utterances, documents, and so forth[79]. Due to NLP's concern with understanding the meaning of language, it can be argued that all aspects of NLP are affected by the grounding problem. In the longer term, solutions to NLP may be of benefit in the following areas:

- Man-machine communication and interaction. Rather than having to use graphical or text-based interfaces (e.g. complex commands or programming languages), a person may be able to simply tell the machine what they want it to do.
- Knowledge acquisition. If machines could understand text and speech, they could read books and newspapers, or even listen to the radio.
- Information retrieval, which involves finding relevant articles from a large database. Thus if a machine truly "understands" not only the user's query, but also the content of the documents in the database, all relevant articles should be found.
- The facilitation of automatic translation, enabling speakers of different languages to communicate seamlessly without the need for a human translator.

There are two main approaches to natural language processing - grounded approaches and "un-grounded" approaches[110, 109, 134]. Grounded approaches model meaning by connecting language

[10]"Natural" languages are those that people speak (e.g. English, Chinese or Swahili), as opposed to "artificial" languages, such as programming languages or logic.

with perception and action, while so-called ungrounded approaches model meaning as structural relations between symbols (i.e. as per a dictionary). Regardless of whether the system is grounded or ungrounded, there are two distinct processes in language processing:

1. Processing (or translating) a text-based representation of a sentence (or sentences) into a representation which conveys the meaning of the text. This task requires syntactic parsing to determine the structure of the sentences being analysed, after which semantic analysis is performed.

2. Processing spoken language, which requires the conversion of spoken language into a text-based representation which can then be processed as per Step 1. Thus, this step requires additional knowledge about phonology and signal processing, and is faced with the the further difficulties of coping with the ambiguities that arise in speech.

Grounding-related approaches to NLP generally focus on connecting the NLP system of an embodied agent (i.e. a robot, simulated or physically embodied) to the world through the sensory and perceptual data of that robot - the assumption being that without such a connection to the world, a “deep” understanding of language by an artificial system is not possible. For example, Roy[108] describes “ungrounded” approaches to NLP as being trapped in “sensory deprivation tanks”, where all meanings bottom out in “symbolic descriptions of the world as conceived by human designers”. Roy and his associates at the Cognitive Machines research project at MIT[11] have been developing grounded approaches to NLP, with their long-term aim being the development of conversational robots. Much of their work uses machine learning techniques to connect and learn the relationships between symbols and sensorimotor data. For example, Gorniak and Roy[64] use a probabilistic model to disambiguate multiple hypotheses of perceived speech (but in a contrived micro world). The model incorporates a Hidden Markov Model[12] based speech recogniser (for perceiving utterances), together with a probabilistic grammar parser (which determines the most likely utterances based upon grammatical rules).

2.4.1.2 Computational Language Acquisition and Evolution

Computational language acquisition and evolution is concerned with using computational algorithms to understand the origins of language - that is, how languages have evolved over time among collections of agents (including people). The term “social symbol grounding” has been coined to describe the collective negotiation that occurs in mulit-agent systems that develop a shared lexicon[29, 135].

[11]http://www.media.mit.edu/cogmac/

[12]A Hidden Markov Model is a statistical model in which the system being modeled has unknown parameters, and the challenge is to determine the hidden parameters from the observable parameters. The extracted model parameters can then be used to perform further analysis, for example for pattern recognition applications.

The objective of research into computational approaches to language evolution (also called "evolutionary computational linguistics") is to define computational models of how communities of agents (either simulated or robotic) can develop (i.e. evolve) shared grounded communication systems. Proponents of such research believe that communication abilities should be learned autonomously, rather than programmed by a human designer. For example, Steels[120] argues that an improved understanding of evolutionary computational linguistics has potential benefits in the domains of both human-to-robot and robot-to-robot communication, while also providing insight to those disciplines concerned with the origins of language and communication.

Generally, studies in evolutionary computational linguistics involve a multi-agent system which learns a communication system (of reduced complexity compared to natural language) which allows the system to refer to common environmental entities or phenomena using mutually understood signs or symbols (for literature overviews see [19, 34, 82, 134]), and thus are subject to the grounding problem. The computational techniques used in language evolution studies, like other grounding research, rely on methods of categorisation (such as pattern recognition and neural networks) to allow the agents or robots to differentiate and recognise different experiences. While neural networks are the most common approach and have proven relatively successful, it has been noted that the use of neural networks makes analysis of the category membership rules more difficult and less transparent[134].

A common technique to facilitate learning in this area of research is to use language or discrimination games - a famous example of which are the "Talking Heads" experiments (e.g. [123, 119, 122]). The Talking Heads experiments involved two cameras interacting in a simplified visual environment, which consisted of coloured shapes on a white board. In these experiments, the agents develop a shared lexicon for the entities in the environment (i.e. the coloured shapes) through the use of language games. The language games involve two agents associating arbitrarily constructed labels with particular environmental entities by playing variants of a "guessing-game" (i.e. "tell me the name of this object") with each other. Extensions of the talking heads experiment included the use of simulated or real robotic agents and the nature of the sensorimotor experiences to be categorised[7, 132, 134]. With these "embodied" experiments, the agents aim (through a series of discrimination games) to distinctly categorise sensorimotor experiences, allowing agents to construct repertoires of categories from scratch. While the Talking Heads experiments (and their variations) have shown that it is possible for agents to construct their own symbols from perceptual input, the experimental scenarios are highly structured and removed from the environments in which a mobile robot may expect to operate (such as the real world).

2.4.1.3 Psychological Experiments

There have been a number of psychological-based empirical investigations which have shown a strong interdependence between language development and embodiment factors, such as the relationship between language grounding and perception[9, 52], and the relationship between language grounding and action[60]. Many of these experiments are based upon asking subjects to perform language tasks regarding perception and action, in which priming effects (i.e. quicker response times) are uncovered when both the language phrases presented in the task and the bodily experience required to complete the task are "consistent". For example, Glenberg and Kaschak[60] asked participants to judge the sensibility of sentences such as "You gave Andy the pizza", "Andy gave you the pizza", or "open the drawer". The subject's response was recorded by pressing a "yes" button - however, the location of the "yes" button would be varied so as to either require a movement toward the subject's body or away from the subject's body. Responses were faster when the action was consistent with the action implied by the sentence. Glenberg *et al.*[59] argue that such priming effects are evidence that language is grounded outside of the language system through perception and action. Glenberg and Robertson[61, 62] propose the "Indexical Hypothesis" (IH), which states that meaning is based on action and affordances, and furthermore, that language is made meaningful by cognitively simulating the actions implied by sentences. According to the IH, three processes transform words and syntax into an action-based meaning:

1. Words and phrases are indexed or mapped to non-arbitrary perceptual symbols[9], which "are based on the brain states underlying the perception of the referent"[60].
2. Affordances are derived from the perceptual symbols (however, Glenberg and Robertson's[61, 62] account of how such affordances is derived is not clear).
3. Lastly, the "sensibility" of the cognitive simulations stimulated by language are constructed in terms of action from the syntactic constructions of sentences.

Thus, in short, the IH proposes that language is made meaningful by cognitively simulating the actions implied by sentences.

2.4.1.4 Establishing "Common Ground" during Communication

Clark and Schaefer[44] refer to a very different notion of "grounding" - finding common ground in communication and discourse. An example of finding common ground in language use is how speakers acknowledge and repeat or rephrase speech to provide feedback to each other. Speakers look for signs of understanding from their listeners, often repeating themselves if listeners do not provide adequate feedback of attention. Poor grounding results in communication failure, and in order to avoid miscommunication participants in dialogue continuously attempt to align their mutual knowledge. While initially concerned with human communication, research in "common ground" grounding has been extended to include computational theories[130] and the domain of human-computer interaction[95]. For robots to interact with us in this world we will need a shared grounding - for example, if I ask a robot to get a specific object, the robot's representation should be concerned with the same real-world entity as myself. Thus, as research in general-purpose robotics progresses, the relevance of "common ground" grounding will become increasingly relevant.

2.4.2 "Dictionary Grounding"

In contrast to grounding through embodiment and experience, the opposing hypothesis is that a symbol's meaning is in fact defined by other symbols - as per a dictionary. While such approaches may seem counter-intuitive in light of Searle's Chinese Room[112] and Harnad's symbol grounding[67] arguments (and have also been labeled as "ungrounded"[110, 109, 134]), there are many examples of this approach. Perhaps the most well known is CYC[87], an ongoing project which represents semantic knowledge in a massive logical database, that aims to (hopefully) enable commonsense reasoning through the sheer bulk of represented knowledge. Collins and Loftus[47] propose conceptual information arises from the pattern of relations among nodes in a network, with meaning arising through the pattern of interconnections between nodes. For example, every node corresponds to an undefined word, and the set of nodes to which a particular node is connected corresponds to the words in the dictionary definition. Others, such as HAL[26] (Hyperspace Analogue to Language) and LSA[84] (Latent Semantic Analysis) take advantage of high-dimensional vectorial algebraic models to model meaning based upon word co-occurrences in massive amounts of data. For example, with HAL the meaning of a word is represented as a vector in a multi-dimensional space based upon 140,000 word-word correlations. Similarly, with LSA the meaning of a word is its vector representation in a space with hundreds of dimensions. For a comparison of grounded versus ungrounded approaches see[59].

2.4.3 Anchoring

A relatively recently identified special case (or subproblem) of the symbol grounding problem is anchoring[49]. Anchoring involves maintaining the link between an agent's internal symbols and the *physical objects* represented by the symbols.

> "We call anchoring the process of creating and maintaining the correspondence between symbols and sensor data that refer to the same physical objects. The anchoring problem is the problem of how to perform anchoring in an artificial system"[51].

Any agent which functions in a dynamic, physical environment, and has a symbolic reasoning component, must have a solution to the anchoring problem. Whereas symbol grounding is concerned with giving meaning to all types of abstract symbols, anchoring restricts the symbol grounding problem to the technical problem of maintaining the relationship between symbols and the percepts that refer to the corresponding external *physical* objects[49]. For example, the anchoring of a household robot could involve linking sensorimotor representations of household objects to the internal symbolic representations of those objects. However, the anchoring problem is subject to the criticism that by restricting the grounding problem to a specific subset of entities (i.e. physical objects), an incomplete theory or solution will be achieved[12].

2.4.4 Vision

Grounding research in the area of vision is concerned with the semantics of computer images, such as real-time vision for robotic systems or content-based image retrieval (the task of categorising and searching through large collections of images). Examples of work in this area include Davidsson[53] who uses neural networks to learn object models for identifying and categorising objects. Hudelot *et al.*[75] view the grounding problem, in the context of computer vision, as how to provide a mapping between numerical image data (i.e. statistical features) of processed images and representations of semantic concepts. Two approaches are presented - a neural network based approach and an *a priori* knowledge based approach. Both approaches involve connecting low-level image data with semantic representations through the use of visual concept ontologies. Chella *et al.*[38] suggest using conceptual spaces[57] for computer vision representations. Conceptual spaces are geometric representations (see Section 2.3.2.4) which Chella *et al.* use as an intermediate layer of representation between raw image data and symbolic representations. Solid block-like shapes are represented using a particular mathematical technique known as superquadrics, and complex objects are then represented by combinations of superquadrics (e.g. a hammer may be composed of two solid shapes - one for the handle, and another for the hammer's head). A similarity metric is then used for categorising, identifying and comparing different objects. In other research, Chella *et al.*[39] suggest how conceptual spaces modeled with superquadrics can be used with situation calculus[93] (a long-standing formal logic for representing action and change) for interpreting motion in dynamic scenes.

2.4.4.1 Pastra's Double Grounding

Human communication involves constant integration of visual and linguistic information. Pastra[99, 98] discusses the need for language-vision integration in robotics, and the role that grounding plays in this process. Pastra coins the term "double-grounding" to describe the process and theory behind how visual representations ground language in physical aspects of the world, while linguistic representations ground vision in mental aspects of the world. Her aim is to endow agents "with the intrinsic intentionality required for exemplifying intelligence in vision-language multimodal situations", claiming that her approach "actually endows an agent with the intrinsic intentionality required for exemplifying intelligence in vision-language multimodal situations"[98]. One of Pastra's systems, called Vlema (Vision-Language intEgration MechAnism), uses an inference mechanism for associating its visual and linguistic representations, producing textual descriptions of building interiors.

2.4.4.2 Image Retrieval and the Semantic Gap

Recent advances in computing, communication, and data storage have led to an increasing number of large digital libraries of documents (both text-based and with multi-media content) publicly available on the Internet. Information retrieval is the science of searching within large collections of information for relevant text, documents, and/or images. Within the realm of content-based image retrieval, the "semantic gap" problem[75, 144] is related to the grounding problem. The semantic gap problem describes the problem of retrieving images that match a user's search query. The semantic gap problem reflects the discrepancy between the relatively limited descriptive power of low-level imagery features and the richness of user semantics.

2.4.5 Action and Behaviour

While both humans and animals are capable of autonomously acquiring skills which require high degrees of dexterity, the same is not true for robots. Mobility tasks such as walking and running pose serious challenges for legged robots, while the manipulation of objects (e.g. even kicking, throwing or catching a ball) is even more difficult. Perhaps more importantly, in most cases the robots have little understanding of their own actions or their consequences. In comparison to the large body of grounding research focusing on language and perception, there is relatively little work concerning the grounding of action and behaviour. Some examples include Wermter[141] who uses neural networks and imitation learning to learn how to perform and recognise three behaviours - "go", "pick", and "lift"; Law[86] uses neural networks to evolve wall following behaviours; Coradeschi and Saffioti[50] focus on anchoring instructions within a logical framework such as telling the robot to "go near" a particular object; and Chella *et al.*[40] present a framework for describing actions using conceptual spaces[57].

2.5 Summary and Conclusion

In this chapter we have discussed a variety of topics, including:

- Harnad's influential symbol grounding problem[67], which asks how can the meanings of symbols be made intrinsic to the symbol system, rather than being interpreted by the human users of the symbol system. Harnad's solution to this problem is to connect symbols to subsymbolic sensorimotor representations through the use of neural networks.
- Numerous researchers[36, 90, 102, 145, 121] have commented that the symbol grounding problem is not restricted to logic-based systems, and instead is relevant to all types of representational systems.
- Brooks' physical grounding[21] treats the grounding problem from a behaviourist perspective, and is concerned with the engineering of grounded behaviours through a tight coupling of sensing and acting (rather than being concerned only with the grounding of a symbolic reasoning component).
- There are numerous approaches to modeling meaning which ignore the grounding problem entirely, instead modeling the meaning of symbols through their relationship to other symbols, with no symbols grounded in experience of the "real-world". The most famous example of this approach is CYC[87].
- Grounding is a multidisciplinary problem, with areas of grounding research including psychology, natural language processing, evolutionary and computational linguistics, computer vision, information retrieval, and developmental robotics.

In conclusion, while grounding to most researchers is concerned with establishing intrinsic meaning in artificial systems, approaches to solving this problem address a different problem - i.e. the problem of establishing a theory of reference between "symbolic" representations used for decision-making and the "subsymbolic" sensorimotor representations (i.e. sensorimotor data) that relate to the real-world entities being represented. Thus, grounding is commonly treated as a problem of categorical perception, in which machine learning techniques (such as neural networks) are used to learn the relevant categories of sensorimotor experience, and these categories are then in turn "connected" to high-level representations used for decision-making.

Next, we consider the implications of grounding for practical robotics.

Chapter 3

Grounding: A Programmer's Perspective

Let's briefly recap. In the previous chapter we were presented with a range of views on grounding, concerning not only how to approach it, but also differing on what it exactly is. While all approaches are concerned with meaning (and most approaches with meaningful representation), interpretations of the nature of both representation and meaning vary under each approach (as we would expect with the meaning of meaning being such a long-standing philosophical problem). Approaches which ascribe to Harnad's symbol grounding[67] aim to develop intentional systems with intrinsic meaning, but rather than modeling such notions of meaning through self-awareness or introspection, meaning is modeled as a problem of reference - that is, connecting representation to sensorimotor data. Alternatively, physical grounding[21] views grounding as a holistic problem of embedding a robot in an environment. Even Harnad, as far back as 1993, has stated that symbol grounding "is beginning to mean too many things to too many people"[70].

In this section we consider the grounding problem in the context of *practical* robotics - i.e. what it is, and why it is important with respect to building robotic systems that satisfy a set of requirements or serve a particular purpose[1]. In particular, a definition of grounding is offered that defines the problem as it is treated for the remainder of this dissertation, and a justification of that definition will be argued. Specific subproblems of the grounding problem will be identified, providing light on why the grounding problem is so difficult. Lastly, the chapter will conclude by discussing the need for systematic method in grounding robotic agents, including a discussion of relevant development methodologies. Grounding Oriented Design (Go-Design) - a methodology for designing and grounding robotic agents - will then be introduced in the following chapters.

[1]As opposed to systems which are developed purely for research or as cognitive models.

3.1 A Working Definition

"It is (and will always remain) a logical possibility that even the kind of grounded system that is the ultimate goal of my approach - a system capable of passing the 'Total Turing Test,' ... i.e., one whose linguistic ... and robotic capacity is totally indistinguishable from our own - could fail to have any intrinsic internal meanings"(Harnad[70])

"All models are wrong; some of them are useful."[17]

Before we discuss the relevance of the aforementioned quotes, recall that in Chapter 1 the following definition of grounding was offered:

Grounding is the process of embedding an artifact in an environment to serve a particular purpose. In the case of robotic agents, it involves enabling that agent to be "in touch" (colloquially speaking) with the state, and nature, of the environment. Grounding is related to the process of perception, and involves correctly perceiving, conceiving, and responding to relevant aspects of the world. Grounding, however, is both task-specific and "body" specific - different agents, with different sensorimotor capabilities, performing different tasks in different environments, will need to perceive, conceive, and respond to different changes in the world with different levels of accuracy or acceptable error. Thus, we assume there can be a degree of error between an agent's beliefs about the world, and the state of that world (which may also be unknown to us). The term *groundedness* is used to describe how *well* an agent is grounded, with groundedness being graded and multi-dimensional.

Note, there is no use of the term "meaning" in this definition. While grounding is related to meaning, meaning is a concept which has troubled philosophers throughout history. Researchers have been unable to agree on a definition of meaning, and theories of how to implement a model of meaning are even less clear. As such, in the context of practical robotics, the utility of discussing grounding implementations in relation to meaning is questionable. It is even debateable whether a robot will ever be capable of being a meaning "user" (i.e. capable of experiencing meaning intrinsically) - Searle's[112] argument might be right. Also, we may not need robots to be meaning users, self-aware or have intentionality to be useful. As Harnad[70] points out, we many never be able to answer this question - even if we manage to develop artificial intelligences that equal or surpass our own.

3.1.1 The Process of Embedding

Grounding research is motivated by the difficulties of building autonomous agents capable of perceiving and acting intelligently in dynamic and uncertain environments. While many researchers argue that for an artificial agent to be grounded it must autonomously learn the meaning of representations (e.g. [67, 69, 103, 59, 128, 110, 109, 134, 121]), the fact is, *all* artificial agents are embedded by design in their environments. That is, designers use their knowledge of the problem to design appropriate control mechanisms, regardless of the balance between knowledge acquired through learning versus innate knowledge endowed to the robot's control program *a priori*. So, is grounding the process of writing control programs? No, but it is involved in this process. More specifically, *we ground robots by understanding the world for them.* The process of embedding an artificial agent in an environment involves identifying regularity and structure within the world that can be used for decision-making. For the designers of robot control programs, it can be as simple as making decisions regarding "what" to perceive and represent. For example, the designer of a control program for an autonomous vehicle may decide to represent and perceive other cars, traffic lights, pedestrians and so forth. Likewise, with regard to implementing a representation design, developers may make decisions such as identifying a sensor state (e.g. a sensor value or pattern of values) which correlates to something in the real world, or in the case of a learning system, develop a (biased) learning algorithm to learn such a relationship. Unfortunately, we often encode little of this knowledge of how we understand the world into the program, and a "code-and-fix" iterative development paradigm results.

3.1.2 Groundedness

Anyone who has ever had "hands-on" development experience in robotics knows a robot's model of the world will invariably have a degree of error, and the nature of that error can vary. For example, a robot soccer player may miscalculate the distance of the soccer ball from the robot, or alternatively fail to "see" the ball entirely. The term *groundedness*[143] refers to the quality of an agent's grounding - i.e. how *well* the agent is grounded. Systems from airline reservation databases to autonomous mobile robots rely on grounded representations. For example, an airline reservation system must manage information about flights and passengers in a way that corresponds to real flights and real passengers. Similarly, an autonomous mobile robot that navigates a physical space will be more effective in achieving its objectives if its internal representations of physical barriers correspond to real physical barriers in its environment. Despite the varied approaches to grounding, little attention has been paid to measuring and assessing the performance of different theoretical approaches and practical implementations (except for [143]). There are many unanswered questions:

When is an artificial agent grounded?[2] Can one agent be more grounded than another?

The prevailing view regarding a system's groundedness is that for a system to be grounded it must *learn* its own meaningful representations or symbols[67, 69, 103, 59, 128, 110, 109, 134, 121]. Our view is that in assessing the quality of an agent's beliefs it is unimportant[3] whether such beliefs are embedded in the program by the programmer, or are whether they are learned by a learning program (in which case the learning algorithm and its biases are also embedded by the programmer). Rather, what is important is the quality of the agent's representation with respect to the agent's intended purpose. To put it simply, *an agent is grounded when it knows what it needs to know, to do what we need it to do.* In other words, an agent is grounded when its knowledge and understanding allows for sound decision-making and appropriate action, therefore making it possible for an agent to accomplish its intended purpose. However, as groundedness is graded and multidimensional, the point at which a robot's grounding is "satisfactory" is not always clear. For example, consider a soccer playing robot and its representation of the location of the soccer ball. The representation of the ball's location may be in error by the smallest of distances (e.g. 1 millimetre), a large distance (e.g. 1 metre), or somewhere in between. Thus, when considering if a representation is grounded, our judgements must be made with respect to task requirements and task performance. That is, designers must specify and understand what constitutes a grounded representation for each grounding problem.

3.2 The Difficulties of Grounding

"The standard reply of the symbolist ... is that the meaning of the symbols comes from connecting the symbol system to the world 'in the right way.' But it seems apparent that the problem of connecting up with the world in the right way is virtually coextensive with the problem of cognition itself."[67]

Due to the ease and effortlessness of which we "make sense" of the world, developers often tend to underestimate the complexities and difficulties in building autonomous agents. The grounding problem is a difficult problem, and has been compared in difficulty to the entire problem of artificial intelligence[67, 12]. In this section we detail specific problems related to grounding. In particular, two main grounding subproblems will be presented: the problem of *relevance*, and the problem of *reference*. The problems of relevance and reference will then be illustrated by comparing the difficulties in writing traditional software (for which we use the example of writing a program to operate a pocket calculator) with the difficulties of writing programs for controlling autonomous

[2]For brief discussions of this issue see [92, 129].

[3]Unless it is specifically relevant to the project.

mobile robots.

3.2.1 Representation Design

Designing and implementing the innate representation for an agent is an unavoidable problem of agent construction. While there has been debate over the need for explicit representations of the world[23], all autonomous agents require at least a description of how to act in the world. The choice of representation governs how both the agent and designer must "think" about the world, with different representations often producing different outcomes - a phenomenon known as "representation dependence"[66]. Representation design contains two central, but related, problems (or "pitfalls"[102]): the problem of *relevance*, which concerns identifying *what* to represent; and the problem of *reference* which concerns how to *maintain* that representation with respect to a changing world.

3.2.1.1 The Relevance Problem

> "For one doesn't want to re-plan in the face of *every* change, only those which are *relevant*, that is, which are likely to affect the achievability of the goal. Thus, for a heavy robot moving across a room the location and dynamics of big, solid objects is likely relevant, but the speed and direction of the draft from the open window is not. Unless the task is to carry a stack of papers. Likewise, the broken and deeply pitted floor tiles make no difference to a robot with big spongy wheels, but might matter to one with different means for locomotion. In general, what counts as a relevant fact worth noticing - say, whether something falls in the general class of obstacle or not - will depend both on the capacities of the agent and the task to be performed. This is obvious enough, to be sure, but turns out to be notoriously difficult to implement in a representational system of any size"[5].

We do not sense, perceive or represent the *entire* environment due to the sheer magnitude and complexity of the reality in which we exist. The problem of relevance concerns *what* to represent[5]. The relevance problem is related to the long-standing frame problem[93], which concerns the problem of how to represent the effects of action without having to explicitly represent the large (possibly infinite) number of intuitively obvious non-effects. Relevance is governed by the goals and objectives of the system. That is, representations serve a purpose - it doesn't make sense to represent unless one knows what the representation affords. Currently, most decisions concerning relevance are made by the designer *a priori*, rather than by an algorithm at run-time.

3.2.1.2 The Reference Problem

The relevance problem concerns what to represent; the problem of *reference* concerns how to maintain the correspondence between representations and the entities they refer to. The problem of reference is related to the problem of perception, as it involves relating representations to sensorimotor data. The problem of reference is made difficult by a changing world - how do we create a representation which reflects in a timely manner the relevant aspects of a dynamic world? The difficulty of this task is compounded by the quality of artificial sensors, as robot sensors often contain a high-degree of noise, and therefore also uncertainty. There are a number of specific errors of perception that can arise when tackling the problem of reference:

- "False positives" and "false negatives"[42]. False positives describe the situation in which the robot effectively hallucinates, falsely perceiving the presence of an entity. In contrast, false negatives describe the situation in which the robot fails to perceive the presence of an entity in a timely manner.

- "Perceptual aliasing"[142] describes the situation when an agent's internal representation confounds external world states due to partial observability of the world. For example, there may be multiple, distinct situations which require separate responses - however, the perceptual input for these distinct situations may appear similar (or even identical).

- Errors of precision and quality (i.e. groundedness). For example, a robot soccer player correctly detects the presence of the soccer ball, but misperceives the distance or location of the ball.

These errors of perception can easily arise as no two percept images of the same entity will rarely (if ever) be identical. For example, in computer vision the same object can appear differently due to changing lighting conditions, as a result of changing viewpoints, occlusion, or from being only in partial view, and an object may even change its appearance over time. Compounding this problem, two different objects may appear almost identical, and in this case, it presents the problem of how does an agent discriminate?

3.2.2 "Traditional" Software vs Robotics Software

Recall that in Section 3.1 grounding was defined as the process of embedding an *artifact*. Thus, from the perspective of a programmer, what is the difference between grounding a pocket calculator and grounding an autonomous robot? A pocket calculator has both sensors (a keypad) and effectors (a screen to display numbers), just as a robot has sensors and effectors (albeit with different capabilities). Certain events need to be perceived (the pressing of buttons), and "behaviour" must be generated (the correct calculations displayed). However, developing the software to operate a pocket calculator (and "traditional"[4] software in general) is seemingly much simpler than developing the software to control an autonomous mobile robot. In the following subsections we consider why this is so.

3.2.2.1 Perception

With regards to perception, both the problems of relevance and reference are easily overcome when writing the software for operating a pocket calculator. Let's begin by considering the problem of reference. Firstly, the programmer of a pocket calculator can assume that the buttons of a pocket calculator work correctly. Likewise, when developing traditional software the programmer assumes input devices such as keyboards or mice are reliable. In contrast, the sensors of an autonomous mobile robot (such as cameras, distance sensors, microphones, etc.) have a degree of noise. That is, even in a static, constant environment a sensor on a mobile robot may produce different readings. Secondly, the mapping between sensor events and real-world events is straightforward with traditional software. Consider the pocket calculator - if a "button press" event is detected on the "9" button, the programmer can safely assume that in the real-world the "9" button has been pressed by the user[5]. In contrast, consider an artificial vision system - the robot programmer must contend with a real-time stream of images (e.g. more than 25 images per second), where each image may contain thousands and thousands of bytes of data, and it is likely no two images are ever identical. To make matters even more difficult, many different real-world entities and events may need to be discriminated from such a stream of data, whereas with the pocket calculator there is a straightforward 1:1 mapping from each sensor (a button) to a real-world event (the button press).

This brings us to the problem of relevance. For the programmer of the pocket calculator, the real-world events that are required to be detected are not only *finite*, but more importantly *a priori* knowledge of these events is *complete*. There are a finite set of digits in the decimal number system, and a finite set of operations that can be performed on them. However, for the programmer of an autonomous mobile robot, the real-world events or entities that may need to be detected may not be completely known *a priori*. Thus, the robot programmer may be required to develop a program that can recognise new, important events or entities without knowing what those entities might be. In the case of the pocket calculator, it is analogous to having to write a program which can deal with the advent of a new numerical operator or number system, but without manually changing the programming of the calculator.

Thus, generally speaking, the complexity and uncertainty of input streams for robotics software far outweighs that of most forms of traditional software. Compounding this problem is the fact that

[4]e.g. software that runs on personal computers with monitors and keyboards, etc.
[5]Even though it may have been unintentionally bumped or pressed.

the robot programmer often has incomplete knowledge of the events and entities that may need to perceived in the future.

3.2.2.2 Action

For the programmer of a calculator, using the calculator's effectors is a simple task - a matter of calling a procedure (e.g. to display the number "100" a parameterised command such as "`print-screen(100)`" might be called). With traditional software, there are rich libraries of pre-built, reliable effector commands for controlling devices such as monitors, printers, and so forth. In contrast, for robots, many actions and behaviours need to be written from the ground up, and are difficult, complex problems in themselves. For example, developing gaits for legged robots is a research area in itself, in which researchers devise algorithms to control many motors (often three or more per leg) at the same time, over time, to produce walking behaviours. Thus, representation design for robotic agents must map to capabilities that the robot can perform, and often at the beginning of a robotics project the behavioural capabilities of a robot are uncertain or unknown.

Just as the sensors of traditional software are reliable, so are their effectors. The pocket calculator programmer can assume that a command such as "`print-screen(100)`" is *deterministic* - or simply put, the programmer can assume it will work. The same is not true for many robotic effectors, actions and behaviours. Whereas the performance of output devices of traditional software are not affected by the environment[6], the performance of many robotic effectors are directly related to the state of the environment. As such, robotic actions and behaviours can be non-deterministic - i.e. they can fail or have unintended consequences. Thus, while separation of input and output is clear-cut in traditional software, many robotic actions require perception to guide them. For example, autonomous robots may be required to act upon dynamic objects in dynamic environments (e.g. kicking a ball, cleaning up trash, etc.). Also, as actions may fail, the robot programmer is posed with problems such as how does the robot "know" an action has failed, how does it know the cause of that failure, or alternatively, how does a robot even know that an action has been successful - and is such knowledge even *relevant* or necessary?

So, we've come full circle - action for robotic agents, just like perception, is affected by the problem of relevance (i.e. what does the robot need to know about action?) and the problem of reference (i.e. how does the robot maintain its representation about action with the performance of that action in reality?). Unfortunately, this not only brings all the difficulties associated with perception (as discussed in Section 3.2.2.1), but is made more difficult as perception must be *integrated* with action.

[6]Of course, devices such as monitors and printers are physical devices and therefore invariably fail over time due to wear-and-tear - however, the programmer can *assume* they will work correctly.

3.2.2.3 Requirements and Specification

When writing the software to operate a pocket calculator, the programmer has clearly defined functional requirements - not only does everyone know how a calculator should behave, the functional requirements for a calculator can be easily formalised and specified. Moreover, the functional specifications are *static* - they do not change with time. Similarly, with traditional software, requirements can be specified in numerous ways - for example, by using graphical screen layouts, measures of performance such as speed of execution, or action flows (e.g. "when the user clicks the 'print button' the report should be printed"), and so forth. Again, functional specifications tend to be relatively static, and in the world of commercial software development, changes to specifications are only made with discrete revisions that are "signed off" by both the developer and the customer.

As a robot is a physical being (i.e. not simply a disembodied software agent), the nature of requirements for robots differs from traditional software in that many (but not all) requirements are specified in terms of human-like behaviours (e.g. deliver the package, detect the intruder, etc). The engineering of robot minds is complicated by the fact that specifying requirements (i.e. "what must the system do?") involves specifying a design (i.e. "how can it be achieved?"). For example, imagine building a household robot, for which has the requirement "the robot should be able to get you a drink from the fridge". Consider the countless ambiguities and decisions that may arise in performing such a (relatively) simple task. Should the robot be able to get you *any* type of drink from the fridge? Should it be able to pour a drink from a container into a glass? What should the robot do if it can't find you the drink you want? Or if it spills the drink? Or there is more than one drink to choose from? And so on, and so on, and so on. Thus, as robots need to be told explicitly how to do absolutely everything, answering the question of requirements (i.e. "what must the system do?") requires the design of a potential solution ("i.e. how should the robot do it?"). Taken to an extreme, a *complete* functional specification would encompass *every* specific behavioural detail for *every* relevant situation the robot may *ever* encounter in the future. Making matters worse[7] is the fact that what is the "right" action from one person's perspective may be the "wrong" action from another person's point-of-view - we all have different ideas of what constitutes "commonsense". Thus, another member of the household may want the household robot to behave differently in the situations we have just described. Therefore, because autonomous mobile robots are expected to operate in dynamic, uncontrolled environments, and as many of the the capabilities of a robot are not realised at the beginning of a project, generating performance specifications and requirements is not a straightforward task. Moreover, generating functionally complete specifications and requirements may be impossible, or at the least, very difficult.

[7]Yes, it gets worse.

3.2.2.4 Testing and Debugging

Testing software involves comparing software against the software's requirements or specification, while "debugging" refers to the process or locating and removing faults in a computer program. With our pocket calculator, systematic testing and debugging can be easily undertaken for two main reasons: firstly, the pocket calculator has a precise set of requirements - thus erroneous behaviour is easily detected; secondly, when faults are detected by the tester, as all input into the system (i.e. button presses) can be controlled and replicated by the tester, faults can be reproduced and isolated.

Unfortunately, it is rare for these conditions to be true for autonomous mobile robots. As highlighted in the previous section (3.2.2.3), specifying precise requirements is often difficult for robotic projects. Also, as autonomous mobile robots operate in dynamic environments, testing and debugging is difficult due to the tester's lack of control over many real-world environmental conditions. As such, debugging (or understanding "why" an autonomous agent behaved in particular manner) is difficult because one can never be exactly sure which particular aspect or state of the environment triggered the errant behaviour. Conversely, the situations in which an agent is tested will never be exactly the same as the future conditions in which the agent will function. Our ability to debug robotic systems can also be restricted by our (i.e. human) perceptual systems - that is, the features of the environment the robotic agent's sensors respond to may be incapable of being perceived by our perceptual systems. So, while the agent's control algorithm may work successfully in one particular environment it may fail in another, even though the environments are perceptually indistinguishable to the human tester. The developer's problem is exacerbated by the fact that observable behaviour may give no clue as to the internal dynamics producing the behaviour. For instance, a robot moving in what appears to be a straight line path can be the result of the control program oscillating between movements to the left and right. Lastly, the problem of testing and debugging robotic agents is exacerbated when we move to multi-agent systems.

3.2.3 Summary

Designing and grounding representations has two key problems - the problem of relevance, which involves deciding what to represent; and the problem of reference, which involves maintaining the correspondence between representations and their referents. As we saw in Chapter 2, most grounding approaches focus on the problem of reference, and there is little (if any) grounding research devoted to the problem of relevance, with most decisions regarding relevance (and to a lesser extent reference) made by designers, not programs. Both the problem of relevance and the problem reference affect the grounding of perception and action for robotic agents, and both problems are made difficult by a complex, unpredictable, and changing world. Perception is a difficult task due not only to the high degree of complexity in sensory streams, but also because the perceptual signatures of relevant entities change with the world, while different entities may have similar perceptual signatures. With "traditional" software there is a clear separation of input and output, while action is deterministic. In contrast, with robotic agents, physical action is non-deterministic and requires perceptual guidance, and thus decisions must be made regarding the relevant aspects of action that should be represented, and how those representations should be maintained over time. Also, as robots are expected to perform physical action, specifying requirements for robotic projects includes specifying requirements in terms of the "correct" decisions that should be made, which creates the problem of separating requirements from design. As a complete set of requirements for a robotic agent would necessarily entail a design of how the the agent should act in all future situations, specifications for robotic agents will be incomplete. As a consequence, when testing robotic agents there is often no precise benchmarks or expectations to compare the system's performance against. Testing is also made difficult because replicating faults is difficult due to the uncontrolled, dynamic nature of the real-world.

In summary, as programmers of robotic software must plan for an unpredictable future, robotic development tends to be a highly iterative, "code-and-fix" paradigm. To develop truly autonomous robots we need to solve the grounding problem - i.e. develop a systematic method for grounding robotic agents, freeing developers from the task of revising a program every time an unanticipated aspect of the environment negatively affects the robotic system's performance. Finally, this brings us to our next topic of discussion - the need for a general, systematic method of grounding robotic agents.

3.3 Towards a General Grounding Solution

Ensuring the groundedness of agents' representations is imperative for successful autonomous decision-making, with the manual creation and maintenance of representation being one of the largest costs associated with deploying robots. Any artificial agent operating in a changing environment is required to respond appropriately to that environment, and thus must have a "solution" to its particular grounding problem. However, when grounding agents, decisions of relevance (i.e. what to represent) and reference (i.e. how to maintain that representation over time) are, on the whole, made manually by designers, rather than by programs - i.e. robotic agents are grounded by *design*. As such, current approaches to grounding robotic agents are ad-hoc - we make the decisions regarding relevance and reference on a case-by-case, system-by-system, task-by-task basis, but we embed little (if any) of the knowledge of how we find structure and meaning in the world. Therefore, such solutions are usually restricted to the particular domain for which the agent operates. As our robots can not autonomously ground themselves, robotic systems tend to brittle in the face of change, and a highly iterative "code-and-fix" development paradigm results. Thus, a general solution to the grounding problem is required.

In this section we consider methodologies which have been developed to aid the development of control programs for autonomous mobile robots. A methodology for grounding robotic agents is needed to provide a means for the systematic grounding of autonomous robots, while in the longer term, assist the development of autonomous grounding capabilities through an improved understanding of the processes by which human designers ground robot minds.

3.3.1 Software Development Methodologies

As most control programs for a robot will be implemented in software, software development methodologies are relevant to the design and implementation of software-based robot control programs. While software design is a creative process, it is not devoid of structure, and software development methodologies are intended to facilitate systematic development of software, aiming to minimise risk and improve product quality. There are numerous software design methodologies - e.g. procedural programming, structured programming, declarative programming, object-oriented programming, design patterns, application frameworks and component-ware[78]. A methodology for software development defines the abstractions used to model software. For example, procedural programming uses procedures, object-oriented programming uses objects, and so forth. Recent trends in software development methodologies have shifted towards methodologies that focus on flexibility, adaptability and change, such as Agile[85] development and Extreme Programming[8]. While software development methodologies can benefit software projects, software for controlling autonomous robots is posed with unique design and implementation difficulties (as discussed in Section 3.2), most of which arise from the nature of a robot's embodiment, such as uncertainty, complexity, uncontrolled and unpredictable environments, the problems of perception and action, and the need for the software program (i.e. agent or agents) to adapt to and respond appropriately to unforeseen circumstances.

3.3.2 Agent Oriented Software Engineering

> "My guess is that agent-based computing will be what object-oriented programming was in the 1980s. Everybody will be in favour of it. Every manufacturer will promote his product as supporting it. Every manager will pay lip service to it. Every programmer will practice it (differently). And no one will know just what it is."[76]

Agent-oriented software engineering is relatively new approach to software engineering in which the notions of an agent and agency serve as the core unit of computational abstraction[78, 137].

[8] http://www.extremeprogramming.org/

Jennings and Wooldridge[78] argue that the difference between agent-oriented software engineering and object-oriented software engineering is that an object is more passive than an agent. For example, objects encapsulate identity ("who"), state ("what") and passive behaviour ("how, if invoked"); whereas agents allow for describing a richer freedom of behaviour and interaction (e.g. "when", "why", "with whom", "whether at all", etc.)[137]. While, by definition, an agent is an encapsulated system, agent-oriented software engineering focuses decomposing a problem into a set (often a large set) of interacting (and sometimes competing) agents - that is, a multi-agent system (often called MAS).

Agent-oriented software engineering has two major drawbacks. Firstly, the patterns and the outcomes of the interactions are inherently unpredictable; and secondly, predicting the behaviour of the overall system based on its constituent components is extremely difficult due to the possibility of unanticipated emergent behaviour[77]. Also, while robotic agents are a type of agent - agent oriented software engineering generally focuses on software, rather than the embodied nature of robots.

3.3.3 Robotics Development Methodologies

We now turn our attention to development methodologies that specifically target *robotic* agents.

3.3.3.1 Brooks' Subsumption Architecture

Brooks' subsumption architecture[20, 21, 23, 24] is a long-standing, behaviour-based approach to designing and implementing control programs for autonomous mobile robots. A subsumption architecture is a way of decomposing complex, intelligent behaviour into simpler cooperating behaviour modules, which are organised in layers. Each layer implements a particular "goal"[9] or objective of the agent, and higher layers are increasingly more abstract. That is, lower levels more closely resemble reflexes (e.g. an "obstacle avoidance" behaviour), while higher levels pursue longer-term objectives (e.g. "explore room"). The subsumption architecture earns its name as each layer can subsume (i.e. overrule or inhibit) underlying layers. Each behaviour is individually hand-built, tested and physically grounded (see Section 2.1.4) through a tight coupling of perception and action. Figure 3.1 displays how a subsumption architecture may be conceptually organised. Figure 3.2 shows a schematic diagram for a robot capable of "hallway following". The "hallway following" robot has three layers, with the lowest layer performing obstacle avoidance, the middle layer performing a "wandering" behaviour, and the highest layer responsible for hallway following.

The main benefits of the subsumption architecture arise from the modularity of design, which allows individual behaviours to be developed, implemented and tested in isolation. Brooks' advocates

[9]Note, the term "goal" is used loosely, as Brooks has argued strongly against the need for explicit representation[23].

task-specific perception and action, which helps reduce the complexity of each behaviour. The main disadvantages of the subsumption architecture arise as a consequence of the modular, layered design. For complex systems, coordinating the subsumption between layers becomes a difficult task, as action-selection is controlled through a high-distributed process of inhibition and suppression. In contrast, it is generally a conceptually simpler task for an engineer to describe a behaviour in terms of a sequence of events, as this is a characteristic of our own planning processes[25].

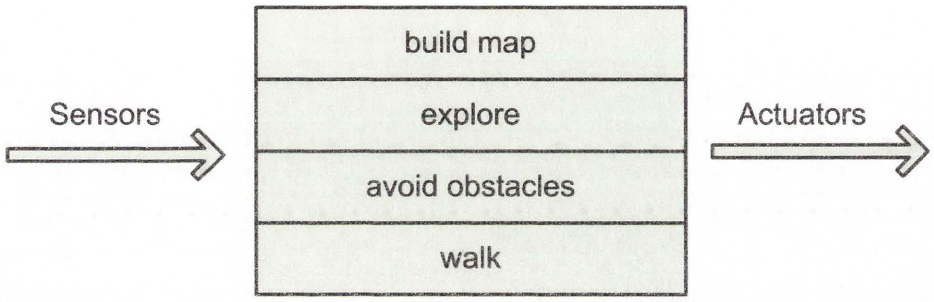


Figure 3.1: A representation of a possible subsumption architecture. Each layer represents an encapsulated behaviour (with its own sensing and acting), with higher levels possessing the ability to subsume lower-level behaviours.

3.3.3.2 Wasson's Representation Design Methodology

Wasson[136] offers a methodology for designing representation systems for reactive robots operating in dynamic and uncertain environments. The methodology guides the designer through the analysis of the robot's task, capabilities and environment, with the aim of answering the questions of what to represent, how to structure that representation and how to keep that representation consistent with a changing environment.

The first step in Wasson's methodology is involves creating a hierarchical decomposition of the agent's task. Wasson asks the designer to perform this process by identifying sequential subtasks and parallel (concurrent) subtasks. An example decomposition diagram is displayed in Figure 3.3. The next step is to identify control flows between tasks, and this is represented through the use of "subtask flow diagrams", as illustrated in Figure 3.4. Once subtask decomposition process is completed, the process of designing representation begins. This involves, for each subtask, identifying "task roles" - i.e. entities in the environment that play a significant role in each task. The designer is then asked to consider issues such as relevant attributes of the environmental entity, and how frequently the task role information should be verified, and the required precision (i.e. the groundedness) of the task role information.

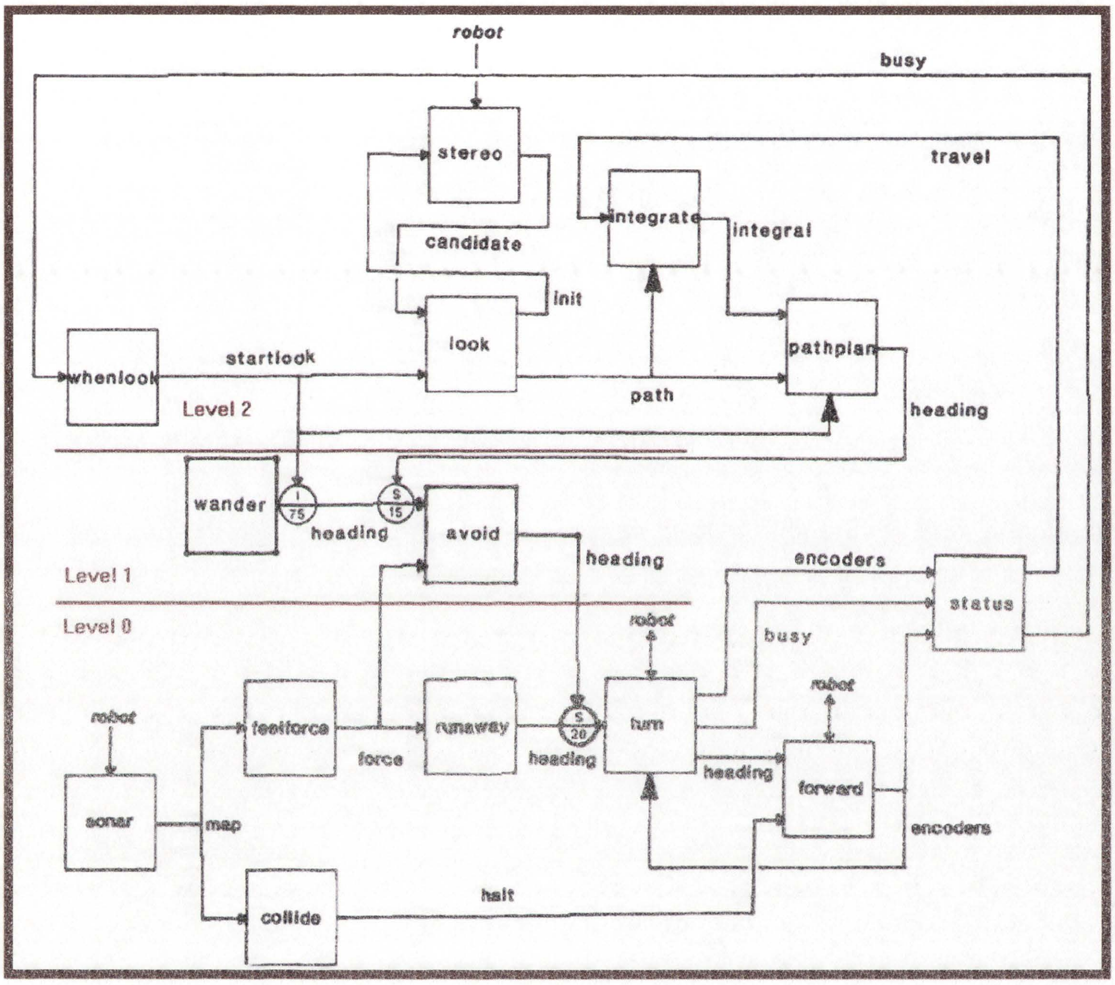


Figure 3.2: A schematic diagram of a subsumption architecture robot capable of "hallway following"[20].

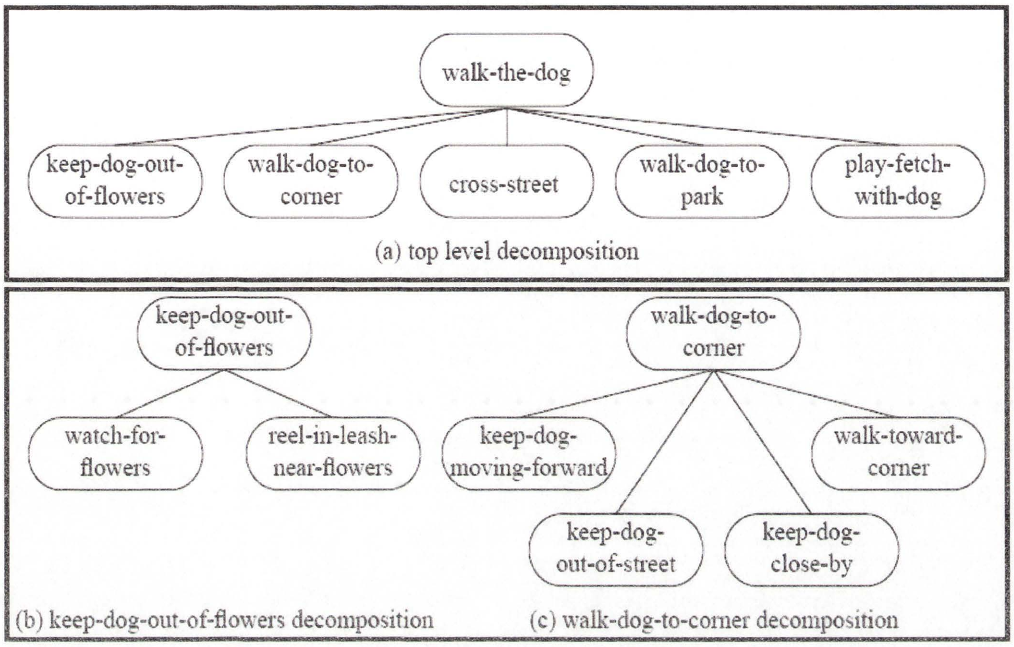


Figure 3.3: A decomposition diagram displaying a partial decomposition of the task "walk-the-dog"[136].

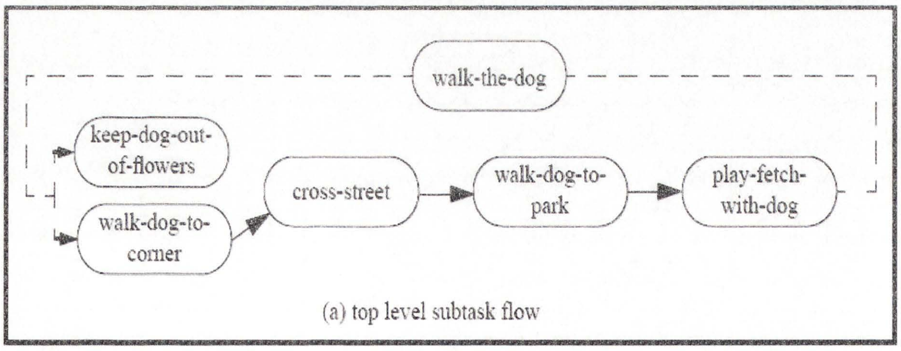


Figure 3.4: A flow diagram for the task "walk-the-dog"[136].

3.3.3.3 Real-time Control Systems Architecture(RCS)

Albus *et al.* have developed RCS[3, 4] (Real-time Control Systems Architecture) - a methodology and architecture suitable for software-intensive, real-time control problems, such as those posed by autonomous mobile robots. Albus boldly claims that RCS is designed to enable any desired level of intelligence, up to and including a human level of intelligence. Inspired by a theory of cerebellar function, RCS models the brain as a hierarchy of goal-directed sensory-interactive intelligent control processes, with each control process being implementation independent (i.e. each control process could theoretically be implemented by neural nets, finite state automata, production rules, etc). RCS uses a methodology to iteratively partition system tasks into control nodes, with each control node sharing a generic node model. Each control node contains a process for behaviour generation, world modeling, sensory processing, and value judgment, together with a knowledge database. The placement of a control node in an RCS architecture hierarchy indicates the the scope and time span of the node, with higher level nodes broader in their planning scope. An example RCS architecture is shown in Figure 3.5.

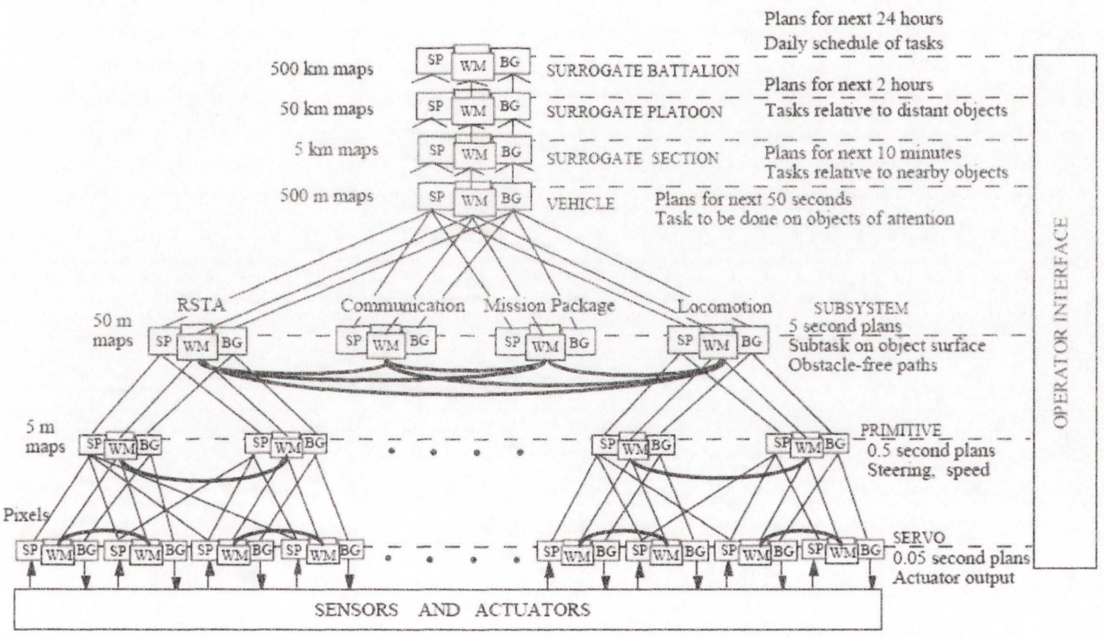


Figure 3.5: An example of an RCS architecture for controlling an autonomous vehicle. Boxes marked "SP" perform sensory processing, boxes marked "WM" perform world modeling, and boxes marked "BG" generate behaviours. Diagram taken from[4].

The RCS design methodology consists of a six step process:

1. An intensive analysis of the domain is conducted using domain experts and other available resources, with the procedural knowledge required to perform the problem task represented in a hierarchical task decomposition tree. In other words, the task is divided into subtasks, and each subtask divided into subtasks, and so forth, with each subtask represented as a command.

2. A hierarchical structure of organisational units that will execute the commands defined in Step (1) is identified. The roles and responsibilities of each organisational unit are specified.

3. The processing that is triggered within each organisational unit upon receipt of an input command is identified. Thus, *how* each unit will perform its task is identified, and this solution is represented as a finite state automaton. Also, the commands that need to be specified to lower-level control nodes in the tree hierarchy are identified.

4. For each state transition identified in Step (3), the situation in which control should pass from one state to another is analysed to reveal their dependencies on the world and task states.

5. The relevant world states identified in Step (4) are labeled, and and any relevant attributes of these world states are also identified.

6. Lastly, the required *groundedness* of the world states is identified, or as Albus calls it "the resolutions at which the relevant objects and entities must be measured and recognized by the sensory processing component" are identified, forming a specification for the sensor system.

Lastly, Albus argues that "all symbols in the RCS world model have been grounded to objects and states in the real world"[4]. Albus views grounding as a process of establishing correspondence between internal representations and the sensorimotor data which indicates the referents. The process of establishing this correspondence in RCS uses "context-sensitive gestalt grouping hypotheses" to connect data structures which represent entities and events to patterns of signals from sensors. RCS uses a predictive model in which model-based expectations are constantly compared to sensory-based observations.

Unfortunately, as of 2005, there "remain many features of the 4D/RCS reference model architecture that have not yet been fully implemented in any application"[4], therefore leaving some doubt as to the practicality of the RCS architecture. Also, while the RCS architecture appears highly elaborate, it remains unclear as to what level of granularity (i.e. the tree hierarchy depth and the time-scope of each node) should be used for different tasks. The RCS architecture results in highly complex hierarchical designs with no simple notation for representing them (as evidenced by Figure 3.5). Lastly, on a positive note, RCS is the only general purpose robotics methodology (besides Go-Design), which treats the symbol grounding problem as a problem *per se*.

3.3.3.4 Behaviour Oriented Design

Behavior-Oriented Design[25] (BOD) is a methodology for engineering behaviour-based agents (embodied or software-based) with multiple, potentially conflicting, goals or tasks. BOD is a behaviour-based approach to intelligence, and is influenced by the Brooks' subsumption architecture[20, 21, 23, 24]. BOD system architectures consist of a library of behaviours (called behaviour modules), with each behaviour capable of encapsulating its own perception, action, learning, and representation needs. Action-selection (i.e. behaviour selection) in BOD relies upon "reactive planning", and provides (both in design and code) a specific data structure called POSH (Parallel-rooted, Ordered, Slip-stack Hierarchical Reactive Plans) to perform this. With POSH, reactive plans can be thought of as a hierarchical, prioritised sequence of behaviours that should be executed in specific circumstances (contexts), with the designer's task being to specify when and how each behaviour is expressed. A simple example of a reactive plan is displayed in Figure 3.6.

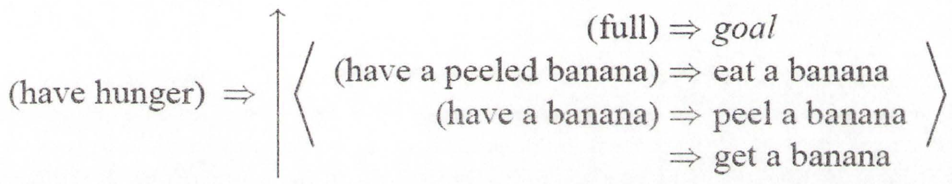

(have hunger) ⇒
(full) ⇒ goal
(have a peeled banana) ⇒ eat a banana
(have a banana) ⇒ peel a banana
⇒ get a banana

Figure 3.6: A "basic reactive plan" for a hungry monkey (p.30 [25]). Statements in brackets, e.g. "(have hunger)", are conditions upon which behaviours should be triggered. Thus, in this diagram if the monkey is hungry, it should get a banana, peel a banana, and then eat a banana.

The BOD methodology provides an iterative design process for decomposing the agent control problem into modular prioritised behaviours, with the decomposition process focusing on identifying "whats","whens", and "hows", with Bryson stating that "designing the intelligence for such an agent requires three things: determining what to do when, and how to do it" (p.23). Behaviours constitute "how", "when" is a problem of action-selection, and "whats" being the level of abstraction and granularity at which a problem is decomposed. Bryson's notation for representing behaviours is displayed in Figure 3.7.

The BOD methodology consists of two main phases - creating a specification, and implementing the specification. A system specification consists of:

1. A high level description of what the agent does.

2. A representation of the actions the agent will perform in terms of reactive plans.

3. The list of whats (including questions/senses) that occur in reactive plans.

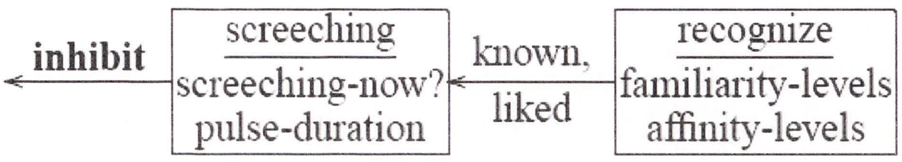


Figure 3.7: A behaviour diagram from BOD (p.27) which represents a monkey that screeches depending upon who it recognises. Behaviours are represented by rectangular boxes and internal state is represented underneath the behaviour name.

4. A list of behaviours that form the behaviour library.
5. A prioritised list of goals.

The implementation phase consists of the following steps:

1. Choose a piece of the specification to work on.
2. Implement the behaviours for that piece of the specification.
3. Revise the specification.
4. Go back to Step (1), and repeat until the system is built.

While BOD offers a simple conceptual framework for decomposing intelligence (in terms of "whats", "whens", and "hows"), it is very focused on specifically using the reactive planning technique for action-selection, thus making it difficult to use in situations where reactive planning is not appropriate. The notation for representing behavioural designs is at a conceptually high-level, thus leaving room for the same behaviour design to be implemented in many different ways. During design, perception, action, and internal state are bundled together into behaviours with no clear distinction. Representation is treated only as internal state, and mechanisms for maintenance or assessing the quality of that representation are ignored. It also does not provide any structured mechanisms for assessing system performance.

3.3.3.5 Roy's Grounding Framework

Roy[108] offers a grounding framework for grounding language in the world for robotic agents. Roy argues that current approaches to designing language processing systems are missing the "critical connection" of grounding, with grounding defined as the ability to "use words to refer to entities in the world". Roy uses "schema diagrams" to design systems, with schema diagrams consisting a "structured network of beliefs connected by projections" of sensorimotor experience. There are a number of types of projections including:

- *sensory projections*, which are "here-and-now" interpretations of sensory data;
- *transformer projections*, which map "from one analog domain to another". In other words, an interpretation of a sensory projection, and thus can be likened to a perception;
- *categoriser projections*, which "map analog domains onto discrete domains";
- *action projections*, which result in a binary, success or fail, outcome.

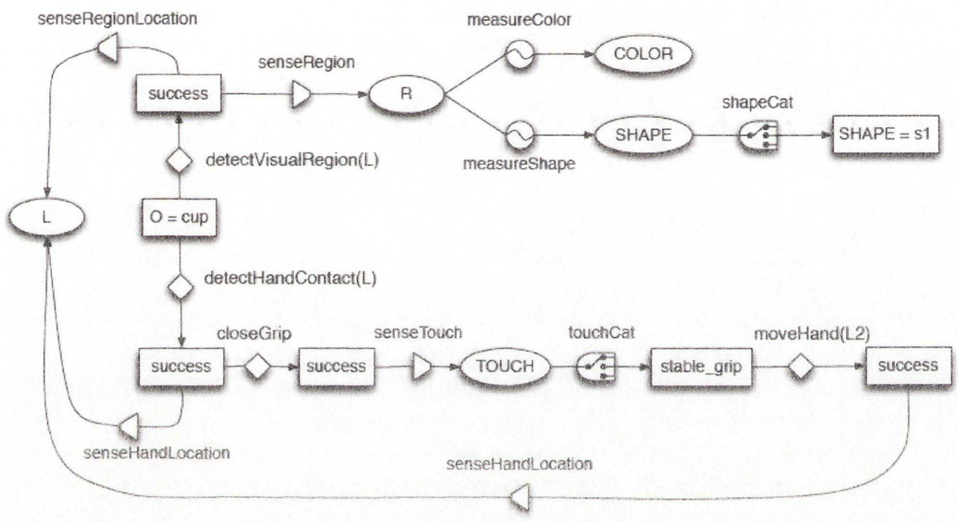


Figure 3.8: An example of a "schema for a tangible (touchable, graspable, moveable, visible) object such as a cup", taken from Roy's Grounding Framework[108]. Legend: "Analog beliefs" are represented by ovals, "categorical beliefs" are represented by rectangles, "sensor projections" by triangles, and "action projections" by diamonds.

An example schema diagram is displayed in Figure 3.8. The main limitations of Roy's grounding framework arise from its focus - its main concern is schematically representing the relationships between words representing language, and the relationship between those words and the sensorimotor experience of the agent. For example, the design is at a conceptually high-level - the diagram in Figure 3.8 could be implemented in many different ways. Where does flow-of-control start and end? No process for designing a schema is offered, as are processes for verifying a design lacking. For example, Roy states action projections result in a binary outcome (success or fail), yet in the example provided (Figure 3.8) failure conditions are not represented. What happens if any of the success conditions are not achieved? How are success conditions evaluated?

3.3.3.6 XABSL

Extensible Agent Behavior Specification Language (XABSL) is a tool for engineering the behavior of autonomous agents in complex and dynamic environments[89]. It is based on hierarchies of finite state machines for action selection. A platform-independent execution engine makes the language applicable on any robotic platform and together with a variety of visualization, editing and debugging tools, XABSL is a convenient system for the development of complex behaviors. XABSL's strength lies with finite state machine management, as both the development tools and programming language make this task relatively simple. The language has been successfully applied on many robotic platforms, mainly in the domain of RoboCup robot soccer.

3.4 Summary and Conclusion

In this chapter we considered the grounding problem from the perspective of designing and implementing software-based control programs for autonomous robots. Grounding was defined as the task-specific process of embedding an artifact in an environment, while the term *groundedness* was used to describe the quality of an agent's grounding, with groundedness being graded and multi-dimensional. Simply put, it was argued that a robot is grounded when it knows what it needs to know, to do what we intend it to do.

Currently, agents are grounded-by-design - i.e. we ground robots by understanding the world for them. In other words, we identify structure and consistency in the world that can be used for decision-making. For example, we make decisions regarding what to represent (the problem of relevance), and we also make decisions regarding how to maintain that representation over time (the problem of reference). We compared the difficulties of grounding traditional software (for which we used the example of a pocket calculator) with the difficulties of grounding control programs for autonomous mobile robots, concluding that the problem of grounding in the context of robotics is a much more difficult problem due to programmers having to anticipate and find structure in a changing, complex, uncertain, and unpredictable world.

Due to the difficult nature of the grounding problem, designers to tend ground robotic agents on a case-by-case, task-by-task basis, but without embedding any of our grounding "know-how" into the robot's control program. Consequently, this results in brittle systems which require frequent program revision with each unanticipated change in the world, as evidenced by the highly iterative, code-and-fix nature of the robotics development paradigm. Thus, a general, systematic solution to the grounding problem is required.

A number of existing robotics design and development methodologies were examined. However, none are specifically designed for grounding robotic agents. For example, Roy's grounding framework only considered schematic representations for grounding language, while Albus's RCS provided a model of how to ground, but many features of the architecture have never been implemented (and the architecture is highly complex). In the following chapters we will consider Grounding Oriented Design (Go-Design) - a methodology specifically designed with the grounding problem in mind.

Chapter 4

Grounding Oriented Design: Introduction and Overview

"Grounding Oriented Design" (or *Go-Design* for short) is a methodology for designing and grounding the "minds" of robotic agents. In this chapter we begin by providing an overview of the methodology, beginning with Go-Design's objectives and design considerations (i.e. explanations are offered as to why the methodology has been designed the way it has). Subsequent chapters will present the details of Go-Design.

4.1 Motivation and Objectives

Grounding is the process of embedding an agent in an environment to perform a task. Or more colloquially, an agent is grounded when it knows what it needs to know, to do what it we need it to do. Currently, the majority of the grounding process is performed by program designers, and not autonomously by the programs we design. That is, designers find meaning, structure, and patterns in the world that can be used in the design of robot control programs, but due to our poor understanding of this process, little (if any) of our knowledge of "how to ground" is encoded in programs. As a consequence, the systems we build tend to be brittle in the face of unanticipated[1] changes in the environment or task, and as such, robotics development is a highly iterative code-and-fix paradigm, in which developing systems capable of "scaling up" to human levels of intelligence has (so far) proven unattainable.

The development of Go-Design was motivated by the need to solve the grounding problem. While Go-Design focuses on grounding by design, rather than autonomous grounding (i.e. programs that ground themselves), developing and refining a grounding methodology provides a small first

[1]Unanticipated from the perspective of the designer.

step towards understanding the difficult problem of how we ground, with a longer-term view of automating this process. Secondly, a grounding methodology can help with the "here-and-now" problems related to the development of control programs for autonomous robots. As discussed in Chapter 3, designing software for robotic agents is posed with unique problems, such as selecting the relevant aspects of the environment to represent, and designing algorithms that can maintain the correspondence between representations and their referents in an accurate and timely manner. These problems are extremely difficult as the programmer must make design decisions *a priori* for agents that operate in a changing, complex, and unpredictable world. Importantly, the quality of an agent's grounding will impact the quality of an agent's decision-making, as robotic agents make decisions which rely upon the perceived state of the world. Thus, the groundedness of agents is imperative for designing robust and reliable systems. When developing Go-Design, two main objectives were formulated:

1. To build a methodology which would improve the groundedness of systems built using the methodology; i.e. using the methodology should result in better grounded systems than if no methodology was used.
2. To build a methodology which, in the longer-term, can improve our understanding of the grounding process, and that in time can be extended to automate the grounding process.

4.2 Methodology Scope

Both objectives concern improving the groundedness of systems - they differ only in scope. The scope of the methodology for this dissertation is to assist the designers of robot minds with the process of grounding-by-design. However, the methodology is intended to provide a base from which insights into how we ground can be gleaned, with the longer-term view of developing systems capable of autonomous grounding.

4.3 Design Considerations

How can we build a methodology to improve the groundedness of systems? In light of previous discussions of what constitutes grounding and how we perform this process (see Chapters 1, 2, and 3) the methodology was developed with a number of considerations in mind, which will now be discussed, in turn, in the following subsections.

4.3.1 Grounded Designers

Designing grounded robots requires grounded designers. As grounding is task-specific, the first step of Go-Design is understanding the nature of the current problem - a process called *context-level analysis*. This process is similar to the requirements elicitation and requirements specification processes that occur when building traditional software. However, Go-Design's context-level analysis is specifically tailored for the development of control programs for autonomous robots.

4.3.2 A Software Problem

Robot control programs will usually be implemented in software, and therefore Go-Design treats the grounding problem as a software development problem. The designs produced by Go-Design are sufficiently detailed (pseudo-code is required) that translation from a grounding design to a software implementation is a straightforward process. Go-Design provides a single diagramming notation which captures the key aspects of both software design (such as hierarchical, modular, layered designs, flow-of-control between modules, and pseudo-code) and grounding design (such as representation, referents, perception, behaviour, and decision-making).

4.3.3 The Relevance Problem

One of the main grounding problems is the problem of relevance - i.e. choosing what to represent. Thus, Go-Design provides a set of structured steps to assist the designer in identifying relevant entities that should be represented. This process involves firstly understanding the requirements and nature of the task-at-hand (a process we call *context-level analysis*), and then identifying the knowledge required to achieve the task.

4.3.4 Problem Decomposition and Decision-Making

Knowing what needs to be represented requires understanding the subtleties of what the agent must do. Therefore, to identify the relevant entities that should be represented the designer must identify the decisions the agent must make to respond appropriately to changes in the environment. Go-Design involves iteratively decomposing the problem task into subproblems until decisions are identified which map to actions and behaviours the agent is capable of performing.

4.3.5 The Reference Problem

Grounding involves maintaining representations with respect to a changing world. Therefore, Go-Design forces designers to identify and define (in terms of decision-making processes) the perceptual mechanisms responsible for maintaining the correspondence between representations and their referents, while also explicitly identifying the decision-making processes which *rely* upon those perceptions.

4.3.6 Groundedness

When is the quality of an agent's grounding "good enough"? How do we "debug" (i.e. find and fix faults in) ungrounded robots? A designer should be able easily explain *why* a robot is behaving in a particular manner - building a grounded robot requires a grounded designer. To help ensure a high quality of grounding, Go-Design guides designers through a process of identifying groundedness requirements for representations based upon the consequences of decision-making processes which rely upon those representations. Also, Go-Design creates transparent, easily understood designs in which the decisions that are dependent upon particular representations can be easily traced and identified, as well as offering structured processes for testing robotic systems.

4.3.7 Design Considerations - Summary

When grounding robotic systems the designer is faced with multiple, inter-related problems. Understanding representational relevance requires decomposing the problem into a set of decisions, while devising means to maintain the reference of relevant representations requires more decisions to interpret sensorimotor data. Lastly, when evaluating the performance of a system the developer must consider whether errors are due to poor decision-making logic, due to representations which misrepresent the state of the world, or because the system lacks the decision-making processes and representations to respond to relevant real-world events. For each of these problems, Go-Design provides a set of structured processes to assist the designer. Next, a brief overview of these processes is presented.

4.4 Methodology Overview

Due to our interest in grounding, Go-Design focuses on understanding (both of the developer and robot) by providing a simple means for modeling knowledge and decision-making in robotic systems. The design process has two main components:

- Processes, guidelines and techniques for designing a robot's mind in terms of units of encapsulated abilities which we call *skills.*
- A modeling notation for representing a mind's design through *skill diagrams.*

Thus, the main unit of abstraction employed by Go-Design is the *skill.* Skills are encapsulated abilities, which when working collaboratively, can accomplish the problem task. Skills can be anything - the ability to see, to throw and catch a ball, or to even "think". We could have called "skills" many other names, such as "abilities", "capabilities" or "things a robot can do". Skills are similar to the behaviours of a behaviour-based system, but differ in that skills need not be reactive (as usually the case with behaviour-based systems), and need not be physically based (e.g. the ability to plan can be a skill, or the ability to perform arithmetic can be a skill).

Go-Design identifies four types of skill: *actions*, *decisions*, *perceptions* and *behaviours.* Actions have no perception, and they can be assumed to be deterministic - i.e. the designer can *assume* they will "work" (but of course they might not, due to wear-and-tear, and so forth). Examples of actions include telling a motor on a robot to move to a particular position, or to set a pixel's colour on a monitor. While actions may lack perception, they can be decomposed. For example, an action such as "walk one step" could be constructed, without perception, as a sequence of motor positions. Decisions are a type of skill, which as their name suggests, are responsible for making choices. Decisions, in software, are implemented as testable conditions (e.g. "if" statements), which based upon the result of that testable condition, choose a particular outcome. In Go-Design, decisions can result in two types of outcome - they can either choose what to do now (i.e. immediately alter the flow-of-control through the program), or they can modify the agent's knowledge (which may affect flow-of-control at a later point in time). For example, a decision which affects flow-of-control could be "if the floor is dirty then clean it", whereas a decision which affects knowledge may be "if there is food on the floor then the floor is dirty". Determining whether the floor is dirty is an example of a perception. Thus, perceptions are a type of decision regarding the state of the world. Perceptions ultimately rely on interpreting sensor data (and other knowledge) being about something in the world. Note, there can be layers of dependency with perceptions - that is, we create can new perceptions from existing perceptions, and errors of perception can propagate through this chain of interpretation. Lastly, behaviours are skills which *integrate* other skills. For example, the ability to "clean the floor" would require perception of dirt on the floor, decisions regarding such as where and when to clean, other behaviours such as the ability to use a vacuum cleaner or mop, and the physical actions to control the robot's effectors.

Specifying how skills collaborate is achieved through the use of *flows.* As we have already discussed, there are two types of flows - *control-flows* and *knowledge-flows.* Control-flows are represented by *skill-transitions.* Skill-transitions are conditions when flow-of-control should be passed from one skill to another. Or in other words, skill-transitions can specify when a skill should stop, and when another should start. Thus, identifying skill-transitions is a problem of action-selection[2], i.e. the problem of selecting *when* to do *what*. In contrast, knowledge-flows specify the modification and sharing of knowledge (i.e. internal state and representation) throughout the system. In Go-Design, knowledge is represented by *concepts*, *percepts*, and *memories*. Concepts are akin to the classes of object-oriented programming - empty data structures, that when populated (or instantiated in the case of object-oriented programming) are represented as a percept or a memory. Percepts are produced by perceptions, and represent the robot's beliefs about the state of the world. Memories are internal state - variables with values. Memories can be about anything - "what am I doing now?", "what day is it?", or even procedural memory of *how* to do something. Thus, our concepts define the structure of both our memories and our beliefs (percepts).

Go-Design uses a process of iterative decomposition to identify the set of skills and their transitions required to perform the problem task. The problem task is called the *context-level skill*. This process of iterative decomposition involves decomposing the context-level skill into a set of sub-skills, and then decomposing each sub-skill into another set of sub-skills, until ultimately we have a set of skills which map to the body's capabilities. There are two main phases to the decomposition process - *basic-design* and *detailed-design*. Basic-design involves constructing a skill architecture - i.e. a hierarchical decomposition over the context-level skill into sub-skills. However, basic-design does not involve identifying skill types, knowledge requirements, or required levels of groundedness - this is the task of detailed-design.

4.5 Summary

In this chapter a brief introduction to Go-Design has been presented. We have discussed the motivations driving Go-Design, and the main units of abstraction, such the particular types of skills (actions, decisions, perceptions, and behaviours), flows (control-flows and knowledge-flows), and knowledge representation (concepts, percepts and memories). We have briefly discussed how Go-Design employes a two stage process of iterative decomposition, which involves basic-design which produces a skill architecture, and then detailed-design which identifies skill types, knowledge requirements, and groundedness requirements. However, we have not discussed the diagramming notation (this is introduced in subsequent chapters), nor the process of *context-level analysis* - the process of understanding the requirements of the problem task. It is context-level analysis to which we now turn our attention.

[2]Note, the term "action-selection" does not refer to action skill-types, but rather the general problem *when* to do *what*.

Chapter 5

Grounding Oriented Design: Context-Level Analysis

Go-Design is a methodology for designing and grounding the "minds" of robotic agents - a vital step towards the *engineering* of robotic agents. However, building grounded systems requires grounded designers. As Go-Design is task-specific (i.e. it is concerned with designing a robot's "mind" to solve a particular problem), the first step of Go-Design is understanding the nature of the current problem. In this chapter we begin with the first step of the methodology - a process called *context-level analysis*[1].

5.1 The Context-Level Skill

As the main units of abstraction in Go-Design are *skills*, the problem task and its requirements are represented by the *context-level skill.* The term "context-level" is used to describe the mind's required capabilities, i.e. the entire behavioural and functional requirements (and hopefully capabilities) of the system. The context-level skill can be likened to the context-level of a data-flow-diagram or the root of a tree, as when the design process begins, the context-level skill is the starting point of an iterative process of decomposition.

Consider designing a program to allow a robot to play soccer. We represent skills diagrammatically using a rectangular box as illustrated in Figure 5.1. Note, the "0" written above `Play-Soccer` indicates that it is the context-level skill - a diagramming notation borrowed from data flow diagrams (DFDs)[2]. However, our skill diagram in Figure 5.1 is quite barren - in fact, it is quite *meaningless.* What does "`Play-Soccer`" *mean*? In the absence of further elaboration, all meaning is derived

[1]Experienced developers of robotic systems may wish to skip this chapter, or return to this chapter after reading subsequent chapters which cover the detail of Go-Design.

[2]Data Flow Diagrams are a common way of representing the flow of data through an information processing system.

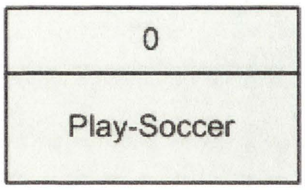


Figure 5.1: A context-level skill diagram for `Play-Soccer`

through your (i.e. the reader's) linguistic understanding of the phrase "play soccer", and the knowledge that a robot will be performing this task - but what kind of robot? In what environment? Why do we want a robot to play soccer? And what style of soccer? How will the robot kick the ball? And so forth... Thus, defining what we mean by `Play-Soccer` requires further elaboration.

Context-level analysis involves:

1. **Identifying the *objectives* of the system.** *Why* is the robotic system being built? Understanding the purpose of the system will help with both defining requirements and understanding the intention behind ambiguous requirements.

2. **Understanding the *current* capabilities of the system**, including the capabilities provided by the robot's existing hardware, such as sensors and effectors, and the existing software capabilities which exploit that hardware.

3. **Understanding the *required* capabilities of the system**, i.e. *what* the robot(s) must do, *when* must it be done, and *where* they must do it (i.e. the environmental operating conditions).

4. **Understanding the *potential* capabilities of the system.** In other words, what are realistic (and feasible) expectations concerning what can be achieved, given the resource and technological constraints relevant to the project? With robotics projects being an emerging and developing research field, answering such questions is often not straightforward, and requires understanding the nature of the robot or robots, e.g. sensory capabilities, effector capabilities, computational capabilities (e.g. processing speed, available memory), and so forth.

5. **Determining what *needs* to be, and *can* be, built to achieve the requirements**, i.e. identifying the gap between existing skills and required skills, as well as any development tools that may need construction. This phase of analysis may cause an adjustment of the project's requirements based upon feasibility considerations arising from the project's constraints.

In the following sections we present a set of guidelines, specifically tailored to the development of control programs for autonomous robots, which can assist the designer in understanding the problem task.

5.2 Objectives

"Objectives" concern *why* the robot's mind is being developed. Reasons can vary - e.g. is the system being built to better understand a topic being researched? Or is it being built to automate a process to save time and money? The main objective in identifying objectives is so that the designer has an understanding of what the project's sponsor would want in uncertain or ambiguously defined situations (ambiguous in terms of lack of requirement specification). There are two main steps:

1. **Identify the context-level objectives.** Context level objectives should be from the perspective of the project's sponsor - what is the project's sponsor hoping to achieve, and why is the system being built?

2. **Prioritise multiple objectives in order of importance.** This process involves identifying "trade-offs" or possible conflicts between different objectives, and then placing a value on their relative importance. For example, safety during the robot's operation versus the speed of the robot's operation.

5.3 Constraints

Constraints specify *how* requirements must be achieved, and thus must identified before design and development commences. Constraints will affect the feasibility of any potential solution. There are two main types of constraints that should be considered:

1. **What are the project's *resource* constraints?** Resources include:

   - Time, i.e. deadlines. When do requirements need to be completed by?
   - Manpower. The number of developers, their level of skill, and their ability to work cooperatively will influence what can be achieved.
   - The robot's computational resources. Algorithms in robotics need to be capable of real-time processing.
   - The robot's sensory and effector capabilities will govern what can be perceived and the physical behaviours that can be performed.

2. **What are the project's *implementation* constraints?** Implementation constraints dictate how problems are solved. In other words, do any particular implementation techniques (e.g. algorithms) need to be used? For example, due to a particular research interest a Kalman filter may need to be used instead of a particle filter for localisation[3].

[3]Kalman filters and particle filters are two common techniques used for robot localisation.

5.4 Current Capabilities

The fact we need to build something indicates we lack particular capabilities. We address the question of "what do we build?" by comparing the existing capabilities of the system with the required capabilities of the system. The robot's current capabilities are dependent upon both the robot's hardware, and the existing software for utilising that hardware.

5.4.1 The Robot(s)

Robots have capabilities and limits based upon their hardware, such as physical, sensory, communication and processing limits. For example:

- **What sensors does it have? What are their physical characteristics? What aspects of the world can they allow us to perceive?** For example, cameras will have characteristics such as field-of-view, distance sensors will have a limited range, and so forth. While many characteristics will be contained in the robot's manuals and supplied documentation, many must be discovered through experimentation. For example, during our experiences the with infrared distance sensor on the Sony AIBO, we mistakenly assumed we could use it to judge the distance to the soccer ball, only to discover the distance sensor was highly unreliable with moving and curved objects. Thus, the quality and nature of the robot's sensors will affect the difficulty of the perception problems faced by the robot.

- **What effectors does it have? What are their physical characteristics?** Robots have physical limits, e.g. how much they can carry, how quickly they can move, etc.

- **What are the processing capabilities of the robot? Does the processing take place on-board or off-board?** On-board processing will usually be limited, whereas off-board processing will be less restrictive. Awareness of the robot(s)' computational resources (memory and processing speed) is required for establishing the feasibility of different algorithmic solutions.

- **What is the robot's battery life?** This may affect the robot's design and suitability for the task, i.e. the need to frequently recharge the robot's battery may not be well suited to some tasks.

- **What communication capabilities does the robot possess?** For example, does the robot posses serial communication, removable flash drives, wireless TCP/IP, and so forth. The ability to communicate with the robot may affect how program revisions are uploaded, how the robot is programmed, tested and debugged, how communication between robots in multi-robot systems is performed, and the external resources that can be accessed (such as the internet, the semantic web, and so forth).

5.4.2 Software

Hardware is of little use without software to control it. The robot's current software will govern the starting point for tackling the context-level problem. For example:

- **How is the robot programmed?** Does it have an operating system? In what language can it be programmed? Does it have an API[4]? Does it have a development environment?
- **What development and debugging tools are available?** Software tools play a significant role in the development of robotic software. For example, representations for sensory translation are representations which allow the designer to better understand the robot's sensory experience. For example, rather than displaying the robot's camera image in terms of raw data (a very large sequence of integers), it is more *meaningful* to convert the raw data to an RGB image, as seen in Figure 5.2. Such tools provide essential infrastructure for robotic development - if they do not exist they will need to be constructed.
- **What existing perceptive capabilities does the robot have? What perceptive skills can be reused from previous projects?** Sensations describe the raw data from robot's sensors, whereas perceptions are interpretations of that data as being about something, e.g. the numeric value attributed to a pixel representing a colour. Intelligent behaviour requires understanding "what is going on in the world" - behaviours require perceptions to guide them.
- **For each perceptual skill, how reliable are they?** What is the nature of their error? How is their performance affected by or related to environmental influences? How can they be tested? Do visualisations need to be developed (e.g. writing software to stream the robot's vision so it can be displayed on a computer monitor). What are the consequences of error? For example, what are the consequences of the robot experiencing false positives (hallucinations) or false negatives (failure to detect the presence of an entity)? What are acceptable levels of performance?
- **What existing effector and behaviour capabilities does the robot have?** What physical actions can the robot perform? What "routines" can be used? For example, to make a robot walk do we need to write a locomotion module or can we simply call a command called "walk"? A skill design must map to actions the robot can already do, or that can be developed.

[4]API - Application Programming Interface - a list of commands that can be used by the programmer.

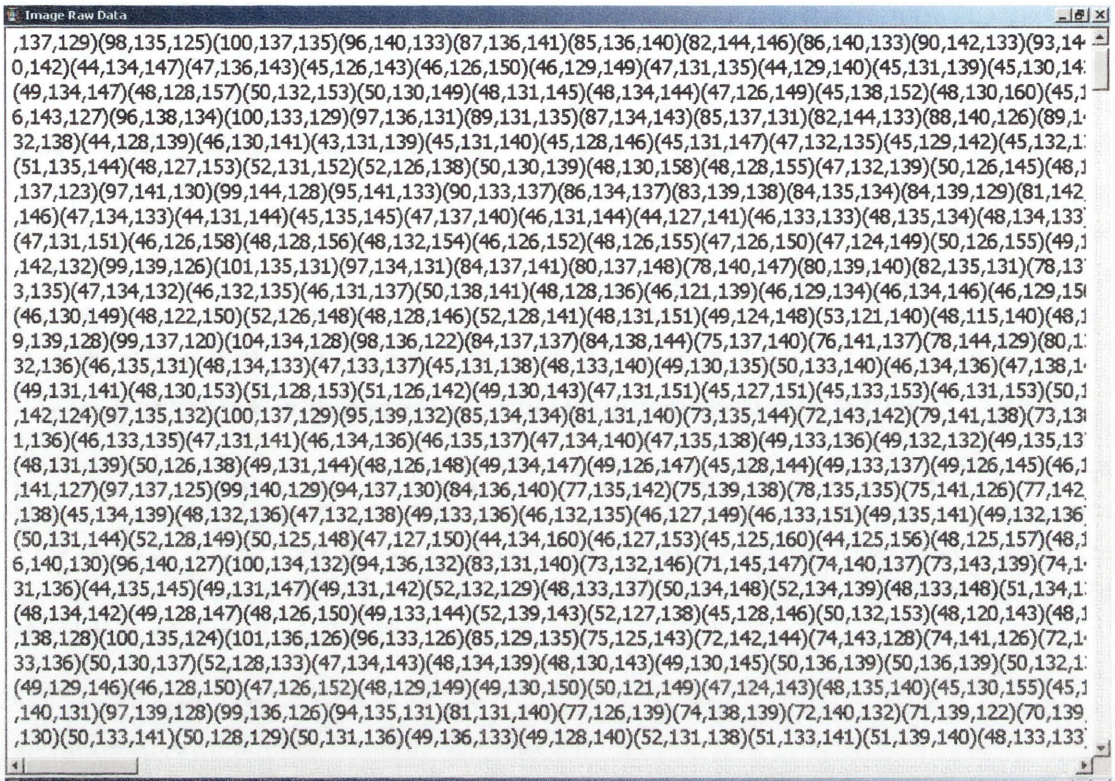
- **How reliable and accurate are the existing effector and behavioural capabilities?** For example, consider an inverse kinematics walking engine - does a command which instructs the robot move 1cm forward *really* move the robot *exactly* 1cm forward? If not, what are the consequences of such error and how can they be overcome?
- **Identify the capabilities of any existing communication software.** For example, are there FTP servers? Is it a case of socket programming, using reliable TCP/IP streams, or some other higher-level communication protocol? Communication capabilities are required not only for multi-agent systems, but for programming and testing.
- **What preexisting decision-making and planning processes does the robot have?** e.g. a reasoning engine, a skill-architecture, etc.
- **What other resources can assist with the development process?** For example, third party tools, newsgroups, and access to domain experts.

5.5 Required Capabilities

In the previous sections we considered the project's objectives (i.e. *why* we are building the robot mind), as well as the existing capabilities of the robotic system. The next step is to consider the requirements of the robot's mind - in other words, *what* the robot's mind needs to allow the robot to do. As requirements are essentially "things" the robot should do, they form a specification for how the robot should operate. When the design process begins (by decomposing the context-level skill into sub-skills), each sub-skill has its own (sub)set of the context-level requirements that contribute to achieving the context-level requirements. The context-level skill's requirements serve as a specification for the system - every skill developed should be done so to (at least partially) satisfy a requirement. Eliciting the complete requirements for the project is thus imperative for designing the system. We call the process of identifying what we are building *requirements understanding.* Later, a *requirements checklist* is presented to assist the developer in identifying requirements.

5.5.1 Separating Requirements and Design

Context-level requirements are requirements that are identified before design begins, i.e. they are usually stipulated by the project's sponsor. In other words, the project's sponsor will specify functional goals for the system (i.e. *what* the robot should do), and the job of the designer is to specify *how* the robot will achieve those functional goals. However (as discussed in Section 3.2.2.3), the engineering of robot minds is complicated by the fact that specifying requirements (i.e. "what must



```
,137,129)(98,135,125)(100,137,135)(96,140,133)(87,136,141)(85,136,140)(82,144,146)(86,140,133)(90,142,133)(93,14
0,142)(44,134,147)(47,136,143)(45,126,143)(46,126,150)(46,129,149)(47,131,135)(44,129,140)(45,131,139)(45,130,14
(49,134,147)(48,128,157)(50,132,153)(50,130,149)(48,131,145)(48,134,144)(47,126,149)(45,138,152)(48,130,160)(45,1
6,143,127)(96,138,134)(100,133,129)(97,136,131)(89,131,135)(87,134,143)(85,137,131)(82,144,133)(88,140,126)(89,1
32,138)(44,128,139)(46,130,141)(43,131,139)(45,131,140)(45,128,146)(45,131,147)(47,132,135)(45,129,142)(45,132,1
(51,135,144)(48,127,153)(52,131,152)(52,126,138)(50,130,139)(48,130,158)(48,128,155)(47,132,139)(50,126,145)(48,1
,137,123)(97,141,130)(99,144,128)(95,141,133)(90,133,137)(86,134,137)(83,139,138)(84,135,134)(84,139,129)(81,142
,146)(47,134,133)(44,131,144)(45,135,145)(47,137,140)(46,131,144)(44,127,141)(46,133,133)(48,135,134)(48,134,133
(47,131,151)(46,126,158)(48,128,156)(48,132,154)(46,126,152)(48,126,155)(47,126,150)(47,124,149)(50,126,155)(49,1
,142,132)(99,139,126)(101,135,131)(97,134,131)(84,137,141)(80,137,148)(78,140,147)(80,139,140)(82,135,131)(78,13
3,135)(47,134,132)(46,132,135)(46,131,137)(50,138,141)(48,128,136)(46,121,139)(46,129,134)(46,134,146)(46,129,15
(46,130,149)(48,122,150)(52,126,148)(48,128,146)(52,128,141)(48,131,151)(49,124,148)(53,121,140)(48,115,140)(48,1
9,139,128)(99,137,120)(104,134,128)(98,136,122)(84,137,137)(84,138,144)(75,137,140)(76,141,137)(78,144,129)(80,1
32,136)(46,135,131)(48,134,133)(47,133,137)(45,131,138)(48,133,140)(49,130,135)(50,133,140)(46,134,136)(47,138,1
(49,131,141)(48,130,153)(51,128,153)(51,126,142)(49,130,143)(47,131,151)(45,127,151)(45,133,153)(46,131,153)(50,1
,142,124)(97,135,132)(100,137,129)(95,139,132)(85,134,134)(81,131,140)(73,135,144)(72,143,142)(79,141,138)(73,13
1,136)(46,133,135)(47,131,141)(46,134,136)(46,135,137)(47,134,140)(47,135,138)(49,133,136)(49,132,132)(49,135,13
(48,131,139)(50,126,138)(49,131,144)(48,126,148)(49,134,147)(49,126,147)(45,128,144)(49,133,137)(49,126,145)(46,1
,141,127)(97,137,125)(99,140,129)(94,137,130)(84,136,140)(77,135,142)(75,139,138)(78,135,135)(75,141,126)(77,142
,138)(45,134,139)(48,132,136)(47,132,138)(49,133,136)(46,132,135)(46,127,149)(46,133,151)(49,135,141)(49,132,136
(50,131,144)(52,128,149)(50,125,148)(47,127,150)(44,134,160)(46,127,153)(45,125,160)(44,125,156)(48,125,157)(48,1
6,140,130)(96,140,127)(100,134,132)(94,136,132)(83,131,140)(73,132,146)(71,145,147)(74,140,137)(73,143,139)(74,1
31,136)(44,135,145)(49,131,147)(49,131,142)(52,132,129)(48,133,137)(50,134,148)(52,134,139)(48,133,148)(51,134,1
(48,134,142)(49,128,147)(48,126,150)(49,133,144)(52,139,143)(52,127,138)(45,128,146)(50,132,153)(48,120,143)(48,1
,138,128)(100,135,124)(101,136,126)(96,133,126)(85,129,135)(75,125,143)(72,142,144)(74,143,128)(74,141,126)(72,1
33,136)(50,130,137)(52,128,133)(47,134,143)(48,134,139)(48,130,143)(49,130,145)(50,136,139)(50,136,139)(50,132,1
(49,129,146)(46,128,150)(47,126,152)(48,129,149)(49,130,150)(50,121,149)(47,124,143)(48,135,140)(45,130,155)(45,1
,140,131)(97,139,128)(99,136,126)(94,135,131)(81,131,140)(77,126,139)(74,138,139)(72,140,132)(71,139,122)(70,139
,130)(50,133,141)(50,128,129)(50,131,136)(49,136,133)(49,128,140)(52,131,138)(51,133,141)(51,139,140)(48,133,133
```

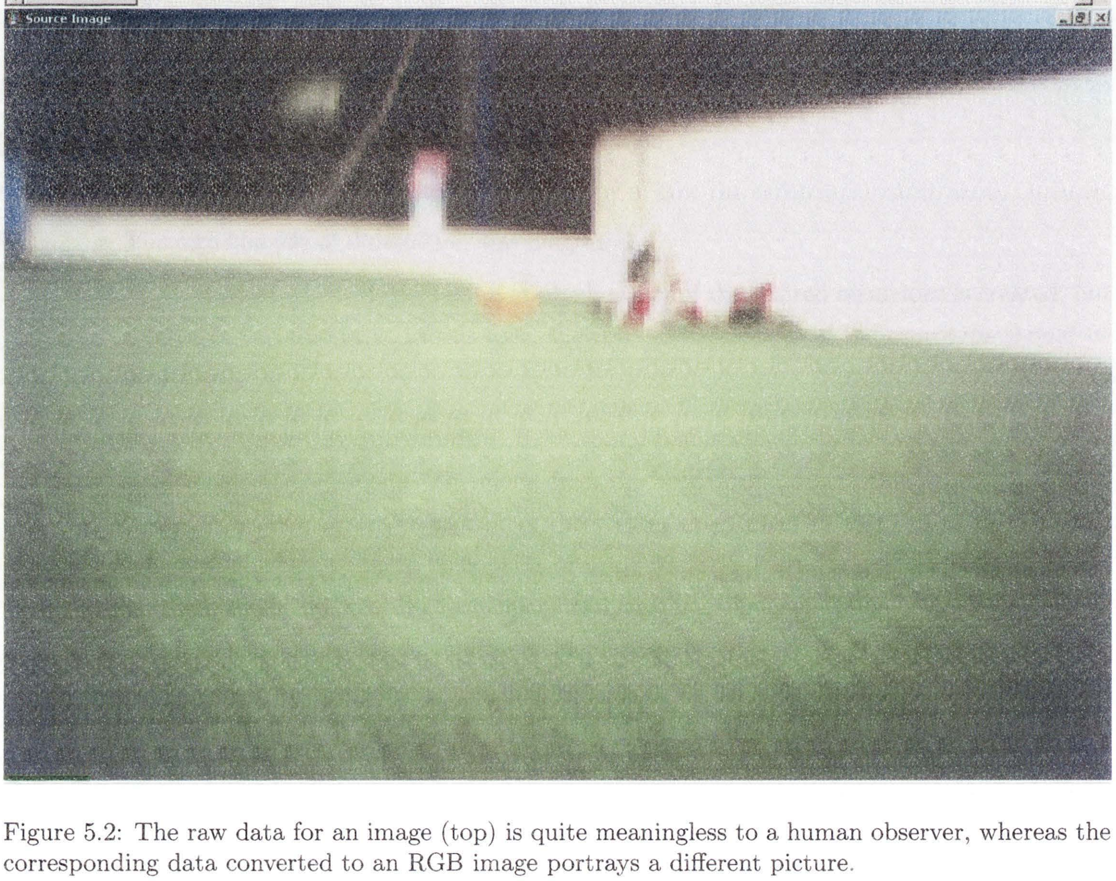


Figure 5.2: The raw data for an image (top) is quite meaningless to a human observer, whereas the corresponding data converted to an RGB image portrays a different picture.

the system do?") involves specifying a design (i.e. "how can it be achieved?"). This problem arises because robots need to be explicitly told how to do absolutely everything. As answering the question of requirements (i.e. "what must the system do?") requires the design of a potential solution (with a complete solution encompassing every specific behavioural detail for every relevant situation the robot may encounter in the future) an inevitable consequence of this phenomenon is that the design process will elicit new, and redefine existing, requirements. Therefore, approaches to the software development life-cycle which involve a discrete, staged separation of requirements analysis and design (e.g. as in the "waterfall" method[5]) are not appropriate when developing robot minds.

5.5.2 Eliciting Requirements - The Requirements Checklist

Eliciting requirements is an ongoing process - design helps elicit requirements. Go-Design provides a set of prompts, called the *requirements checklist*, to assist the designer in eliciting the requirements of the project. Not all items of the checklist will be relevant to any particular project, nor is the list exhaustive. When defining particular requirements the designer should attempt to quantify (if possible) a performance benchmark.

The requirements checklist is as follows (note - a concise version of this checklist is contained in the Go-Design Step-By-Step Guide in Appendix A):

1. **Describe the behaviour of the robotic system in as much detail as practically feasible.** System behaviour can be described in a number of ways, such as:
   - By writing a narrative, i.e. a textual description.
   - By creating a simulation, i.e. a virtual model of how the embodied system should behave.
   - Through the use of illustration and diagrams.
   - By developing a prototype, i.e. an approximation of the desired behaviour is created, but lacking in certain areas. In this case, the specific deficiencies of the prototype should be identified.
2. **Specify behavioural requirements**. Behavioural requirements are measurable behavioural (physical) performance targets. Therefore, from the behavioural description produced in Step (1), identify specific measurable aspects of the system's performance that should be achieved. For example, "the robot should be capable of using its robotic arm to grasp soft-drink cans", or "the robot should be capable of moving to and from the rooms of a single story house, safely avoiding obstacles and collisions, upon verbal command".

[5]The waterfall approach is a long-standing, common approach to the development life-cycle in which there are discrete, sequential development stages, e.g. requirements analysis, design, implementation, testing, and maintenance.

3. **Identify perceptual requirements**. Perceptual requirements are performance targets regarding the detection of particular events and/or aspects of the world. The behavioural description and behavioural requirements produced in previous steps will contain elements of the environment that must be perceived for successful task completion. Where possible, identify the sensors which can be used for the particular perception problem, and acceptable levels of error, such as acceptable levels of false positives and false negatives.

   - What *physical objects* need to be perceived? e.g. a soccer playing robot will be required to perceive the soccer ball;
   - What *attributes of those physical objects* need to be perceived? e.g. the location of the soccer ball;
   - What *events* need to be perceived? Events are relevant and significant changes in the state of the environment that should be detected for the purposes of decision-making. For example, in robotic soccer events may include a goal being scored, losing possession of the ball, and so forth.

4. **Identify locomotion requirements (movement)**. Locomotion requirements arise from behavioural requirements. e.g. how is the robot required to move around the environment? For example:

   - *Directional requirements.* What are the directional requirements? Does the robot only have to move backwards and forwards? Turning? Strafing? etc.
   - *Speed requirements.* How quickly should each type of movement be capable of being executed? e.g. straight-line speed? turning and rotational speed?
   - *Stability requirements.* How stable does the robot need to be while moving?
   - *Terrain.* Is the robot's movement requirements related to different types of terrain? What is the nature of the terrain the robot is required to move over? (or move through, in the case of autonomous ships, planes, helicopters, submarines, etc).
   - Obstacle avoidance and path planning requirements. What hazards in the environment may need to be avoided?

5. **Identify any special actions or movements.** For example, the ability to move an object, or the ability to get up after a fall (in the case of legged robots).

6. **Identify Communication Requirements**. Identify how, and what, the robot(s) communicate with:

- *Developers.* Developers need to "communicate" with their robots for the purposes of uploading program revisions, teleoperation, debugging, and testing, e.g. "the data link needs to support a sufficient rate to send command sequences, and to get data and telemetry back".
- *Users.* For example, a "security robot" needs to report the presence of intruders.
- *Other robots.* In a multi-robot system, do the robots need to communicate, and if so, what do they need to communicate and how will they transmit such information?

7. **Identify Environment Requirements**. *Where* is the robot or robots expected to perform their tasks, and *when* they perform those tasks in what state will the environment be? Understanding the nature of the environment allows the developer the best chance of anticipating the future environmental conditions in which the robot(s) will operate. The main problem areas are:

    - Identifying *change.* That is, what dynamic aspects of the environment might impact upon the performance of the system. For example:
        - Some physical objects may move location, effecting locomotion and path planning (e.g. obstacle avoidance).
        - Some physical objects may change appearance or shape, possibly effecting perception and action.
        - Lighting may change, affecting vision systems.
        - Background noise may change, affecting auditory perception.
    - Identifying aspects of the environment that can be *controlled* and/or *modified* to assist the robot(s). For example, adding fixed landmarks to assist with localisation.
    - Identifying *physical hazards* to the welfare of the robot(s) - e.g. a staircase poses a hazard for a wheeled robot.
    - Identifying *physical boundaries and limits* - are there limitations as to where the robot should be allowed to go?

8. **Identify development or debugging tools which need to be built.** For example:

    - *Does a programming environment or programming language need to be created?* Are the current means for programming the robot efficient and expressive enough? Can software be run off-board, offering the benefits of established development environments, such as integrated debuggers? For example, with the Sony AIBO, programs are written in C++. However, making program modifications requires rebooting of the robot - a process which can take several minutes. Consequently, to improve programming efficiency, many RoboCup teams have developed run-time interpreters for scripting languages such as Perl and Python, thus enabling programs to be modified without wasting the time required to reboot the robot.

- *What representations for sensory and perceptual translation need to be built?* For example, tools for streaming vision data.
- *What representations for state representation need to be built?* For example, visualisations of the robot's perceived location and orientation, or display of the robot's current plan.

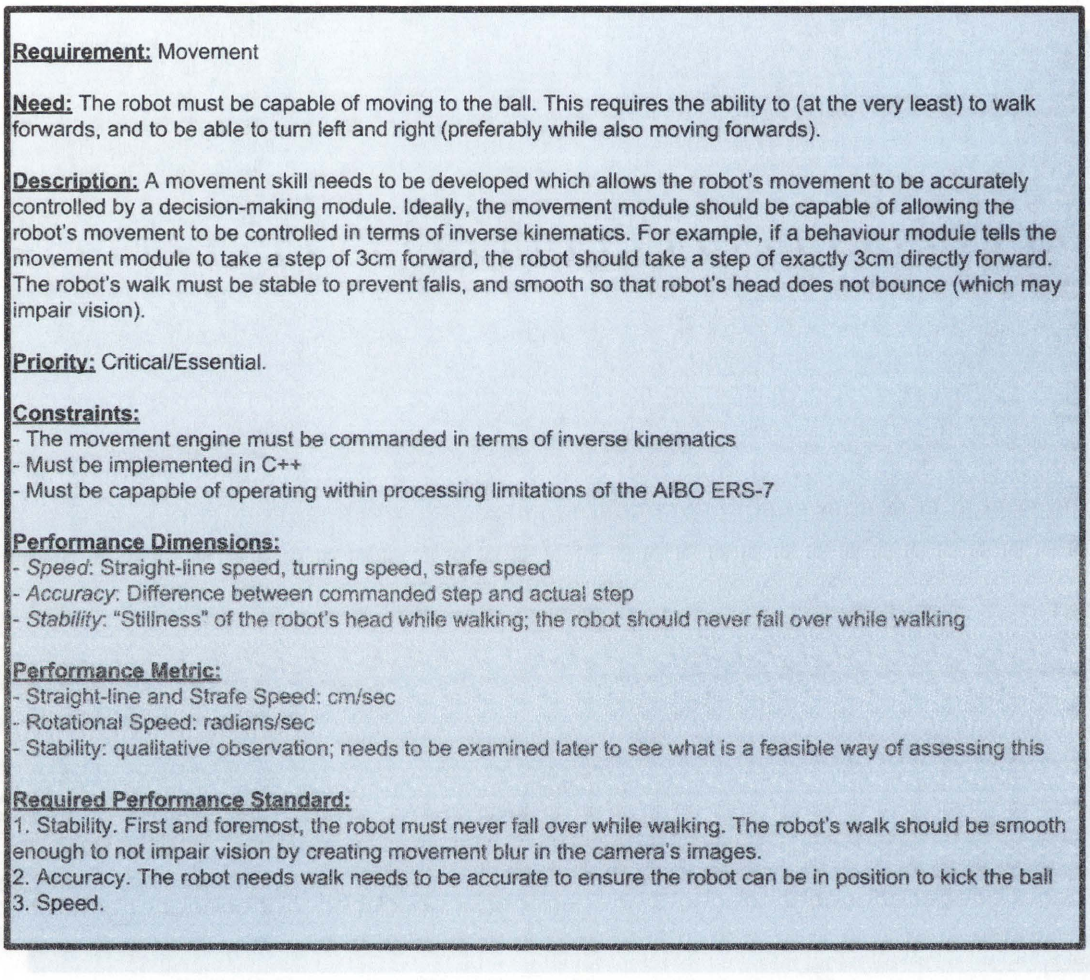
9. **Are there any *future* requirements?** In other words, requirements which are not part of the project's current scope, but should be allowed for in terms of building an extensible design.

Note - there are many other ways of categorising requirements, e.g. learning requirements, safety requirements, conceptual requirements, decision-making requirements, and so on. The categories presented in this checklist are provided as prompts and are not exhaustive nor complete.

5.5.3 Requirement Templates

Go-Design employs *requirement templates* for documenting requirements. An example *requirement template* is shown in 5.3. For each requirement the designer should specify:

1. ***What* the robot should do**;
2. ***Why* the robot should do it**, i.e. *the need*;
3. **The nature of any *constraints* on how the requirement should be achieved**, e.g. "the robot must be able to localise, but the localisation algorithm must use a Particle Filter";
4. **The *importance* or *priority* of the requirement**, e.g. System Critical/Essential, Desirable, Luxury, etc.
5. **Performance dimensions and metric**, i.e. how attempts to satisfy the requirement can be measured; and lastly,
6. **The required performance standard** in terms of measurable performance dimensions.



Requirement: Movement

Need: The robot must be capable of moving to the ball. This requires the ability to (at the very least) to walk forwards, and to be able to turn left and right (preferably while also moving forwards).

Description: A movement skill needs to be developed which allows the robot's movement to be accurately controlled by a decision-making module. Ideally, the movement module should be capable of allowing the robot's movement to be controlled in terms of inverse kinematics. For example, if a behaviour module tells the movement module to take a step of 3cm forward, the robot should take a step of exactly 3cm directly forward. The robot's walk must be stable to prevent falls, and smooth so that robot's head does not bounce (which may impair vision).

Priority: Critical/Essential.

Constraints:
- The movement engine must be commanded in terms of inverse kinematics
- Must be implemented in C++
- Must be capapble of operating within processing limitations of the AIBO ERS-7

Performance Dimensions:
- *Speed*: Straight-line speed, turning speed, strafe speed
- *Accuracy*: Difference between commanded step and actual step
- *Stability*: "Stillness" of the robot's head while walking; the robot should never fall over while walking

Performance Metric:
- Straight-line and Strafe Speed: cm/sec
- Rotational Speed: radians/sec
- Stability: qualitative observation; needs to be examined later to see what is a feasible way of assessing this

Required Performance Standard:
1. Stability. First and foremost, the robot must never fall over while walking. The robot's walk should be smooth enough to not impair vision by creating movement blur in the camera's images.
2. Accuracy. The robot needs walk needs to be accurate to ensure the robot can be in position to kick the ball
3. Speed.

Figure 5.3: Requirements for `Movement`

5.6 Conclusion

This section presented techniques to help the designer understand the problem at hand. In the following sections we turn our attention to structured processes for designing solutions to robot control problems. Note that this process is ongoing throughout the life of the project. Understanding requirements requires design, which can lead to reconsideration of requirements.

Chapter 6

Grounding Oriented Design: Part I - Basic-Design

In the previous chapter we focused on context-level analysis - i.e. developing an in-depth understanding of the problem we are trying to solve. We now turn our attention to the first step of the design process - designing a skill-based architecture to solve the required task. In subsequent chapters we will then demonstrate how the skill-based architecture can be used to construct a detailed-design which focuses not only on skills, but on decision-making and knowledge representation.

6.1 Introduction

Go-Design is a methodology for designing and grounding representations for robotic agents. Central to Go-Design are "skills" - units of encapsulated intelligence. Go-Design involves iteratively decomposing a complex problem (the context-level skill) into a collection of simpler, collaborating skills. This methodology has two main components:

- Processes, guidelines and techniques for designing a robot's mind in terms of skills.
- A modeling notation for representing a mind's design through *skill diagrams*.

For a more detailed overview of Go-Design readers are referred to Chapter 4. In this chapter we focus on designing a skill-based architecture, from which a detailed-design[1] can be formulated (Chapter 7).

[1] "Detailed" refers to a pseudo-code level of detail.

6.2 Skills

The things a mind can do are called *skills*. Skills are encapsulated, task-based abilities; they are labels we attach to processes which do or achieve something. Skills can be anything - the ability to see, to throw and catch a ball, or to even "think". We could have called "skills" many other names, such as "abilities", "capabilities" or "things a robot can do". Skills can use other skills - for example, being able to "do the grocery shopping" requires the ability to "drive the car" coupled with the ability to "find the shopping centre", and then after arriving at the shopping center, the ability to "find a car park", and to "find the grocery shop inside the shopping center", and then (once all the previous skills have been accomplished successfully) there is the art of "finding and controlling a shopping trolley", and so forth - intelligent behaviour requires a rich assortment of skillful skills.

6.3 Skill Collaboration

A collaborating set of skills can accomplish more than any of the individual skills alone. For example, the ability to "do the grocery shopping" is built (and reliant) upon a set of collaborating "subskills" such as "drive the car", "find the shopping centre", and so forth. The grocery shopping example illustrates three key types of skill collaborations (or "interactions") that we use to design robot minds. Firstly, there are *skill sequences* - skills that operate sequentially over time, one after the other, like a chain. For example, "1. Find the car; 2. Start the car; 3. Drive the car to the shops", and so forth. In a skill sequence the term *skill-transition* is used to describe when one skill stops and another starts. Secondly, there are *concurrent skills*, i.e. skills that operate at the same time - for example, the ability to drive the car while also being able to simultaneously plan (and replan) the best route to the shops. Lastly, there are *skill decompositions*. Skill decompositions are the labels we attach to hierarchical groups of skills that do something as a whole. In other words, skills are composed of *subskills*. For example, some of the subskills we identified of "do the grocery shopping" included "drive the car", "finding and controlling a shopping trolley", and so forth. For each of these subskills, they are, in turn, also composed of subskills. For example, the ability to "drive a car" requires skills such as "perceive and obey road signs and traffic laws", while the ability to perceive road signs in turn requires the ability to interpret and discriminate certain collections of shapes and colours in the visual stream as being particular meaningful road-signs. Thus, generally, for every human behaviour there is a mountain of detail that we (designers of robot minds) either ignore or take for granted - until we try and automate such processes artificially.

6.4 Skill Diagrams

6.4.1 Design Considerations

The design of skill diagrams was motivated by the need to capture, using just one style of diagram, a number of key aspects of both software and grounding design. These key aspects include:

- *Modularity of design* - just as class-diagrams model the classes of an object-oriented design, skill diagrams illustrate the skills of a grounding oriented design.
- The *hierarchical*, *layered* nature of software designs, or conversely, the *decomposition* of the context-level skill into subskills.
- *Flow-of-control* - skill diagrams represent how control-flows throughout the control program (including both concurrent and sequential flows), and thus, the consequences of decision-making.
- The *events* which cause control to pass from one skill to another. These events are called *skill-transitions*.
- The *perceptions* and *decisions* required to detect and recognise skill-transitions. These decisions are called *transition-conditions*.
- The perceptions and decisions required to guide *behaviours*.
- The *knowledge* (i.e. *representations*) required to make decisions, and how that knowledge is used throughout the entire system.
- Lastly, the decisions required to maintain representations with respect to their referents - *the problem of reference*.

In this Chapter we focus on "basic-design", which involves using skill diagrams to model a skill architecture. A skill architecture illustrates the skill decomposition for a given problem, and the skill-transitions (events) which result in control being passed from one skill to another. In Chapter 7 the more detailed aspects of skill diagrams are presented (such as perceptions, decisions, and knowledge).

6.4.2 The Basics

Figure 6.1 provides a legend which describes the elementary notation used in skill diagrams, while Figure 6.2 illustrates a simple skill diagram for the skill "`Do-Grocery-Shopping`".

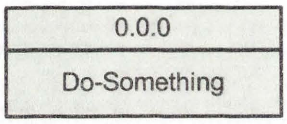
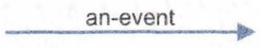
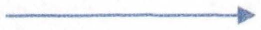
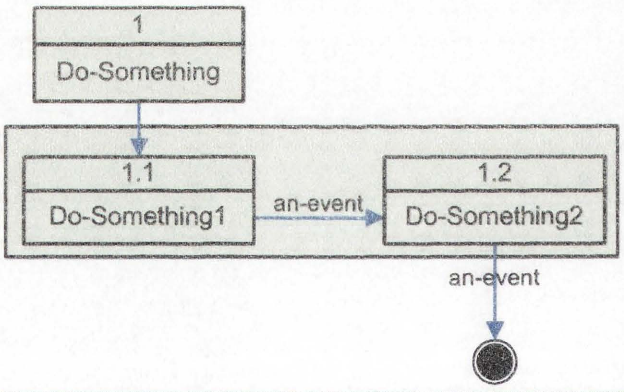
Symbol	Explanation
	A skill. The skill's name is written underneath a numeric identifier.
	A skill-transition. The transition-condition is written above the arrow.
	A skill-transition, without a transition-condition. A condition-less skill-transition occurs automatically (i.e. by default).
	A skill terminator, used to indicate when a skill stops.
 1 Do-Something 1.1 Do-Something1 an-event 1.2 Do-Something2 an-event	A skill diagram illustrating decomposition of a skill by placing a rectangular box (the *decomposition box*) around the subskills. Also observe that the numeric identifier of each sub-skill relates to the parent skill.

Figure 6.1: Skill Diagram Legend.

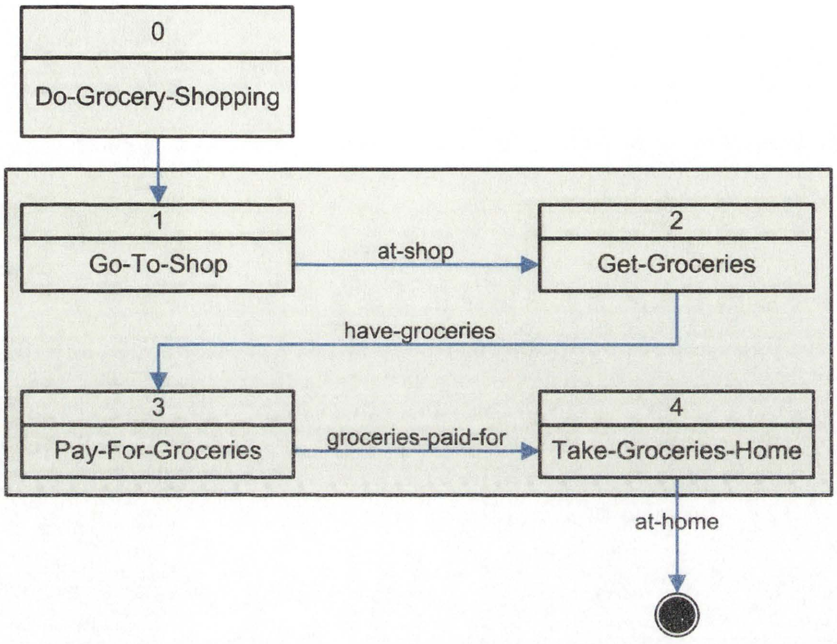


Figure 6.2: Shopping Skill Diagram.

6.4.2.1 Skill Naming

Skills are represented with rectangular boxes, with the name of the skill written below a numeric identifier. The first letter of each word in a skill name should be capitalised, with the words of a skill's name separated by a hyphen. A skill's name should be unique.

6.4.2.2 Numeric Identifiers

The numeric identifier describes how a particular skill relates to other "parent" skills, and thus how a skill relates to other skills in the "skill network". For example, if a skill's identifer is "1", any subskills (other skills that are used in the skill's design) would be labeled as "1.1", "1.2", and so on. Thus, a skill with the numeric identifier "1.2.1" forms part of the implementation of skill "1.2", with skill "1.2" forming part of the implementation of skill "1", and skill "1" forming part of the implementation of the context-level skill (skill "0"). Note, while a numeric identifier is unique (in that no other skill will share the same numeric identifier), it is possible for the same skill to be reused in different diagrams. Therefore, as the same skill could be reused by many different skills, a skill could appear in numerous skill diagrams, each time possessing the same skill name, but a different numeric identifier. Thus, the main purpose of a numeric identifier is to represent where a skill sits in relation to other skills in the "skill network".

6.4.2.3 The Decomposition Box

The purpose of the *decomposition box* is to represent which particular skills (and later, in detailed-design, also representations) form part of a skill's design, and thus decomposition. The box need not be a "box" - indeed, any shape encompassing the relevant skills and representation which form part of a skill's decomposition is sufficient. Note - whether the label for a skill-transition falls inside or outside a decomposition box is irrelevant. The decomposition box is most useful when drawing multiple skill diagrams on the same page or space, as it clearly groups the skills and representations which belong to that skill's design.

6.4.2.4 Flow-Of-Control

Flow-of-control is represented through the use of skill-transitions, with skill-transitions represented with an arrow. Skill-transitions can have transition-conditions, which are testable conditions which signify when one skill should stop and another should start. If a skill-transition does not have a transition-condition, this implies that the skill-transition occurs automatically (i.e. by default). We represent transition-conditions by adding a caption (in lowercase) to the arrows between skills on a skill decomposition diagram. Also note that a black-dot with a circle around it is used to represent the skill's completion (as per UML state diagrams). Flow-of-control begins with the parent skill - thus, in Figure 6.2 flow-of-control begins with `Do-Grocery-Shopping`. A skill diagram can have multiple flow-of-control end points (terminators), but flow-of-control must always start with the parent skill. Lastly, concurrent skills are represented by having multiple transitions active at the same time. For example, consider that during `Collect-Groceries` we may need to be able to both find the particular groceries we need, while also pushing the shopping trolley at the same time. Figure 6.3 illustrates how to represent concurrent skills.

6.5 The Design Process

In the previous section we discussed how skills interact, and the basics of representing skill designs in skill diagrams through the "grocery shopping" example. The example illustrates how skills collaborate to form new skills, and that, in designing robot minds we aim to build not only a set of skills, but to also define the interactions between those skills. The process of "breaking up" (or separating) a skill into a number of other cooperating subskills is a process of *decomposition*. The term *iterative decomposition* refers to the process of decomposing each of the subskills into another set of subskills, and then in turn decomposing each of the new subskills into subskills, until ultimately we have a set of skills which map to either the robot's existing capabilities, or to capabilities that can

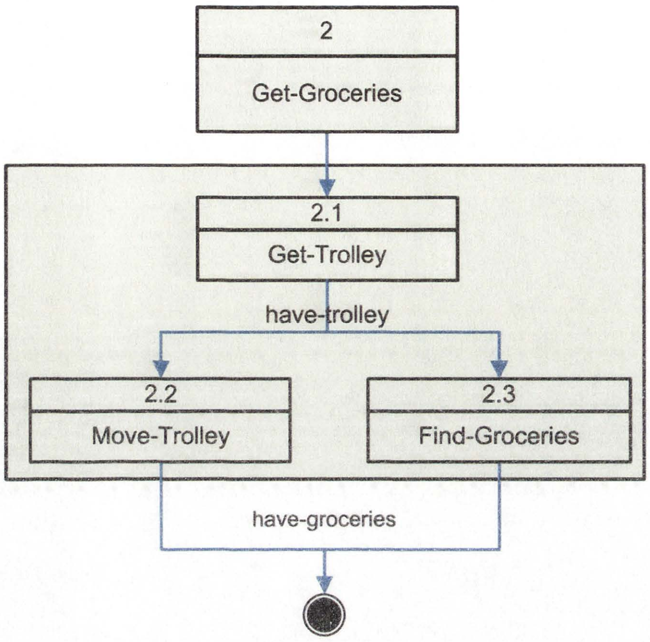


Figure 6.3: The diagrammatic notation for representing concurrent skills. `Move-Trolley` and `Find-Groceries` operate concurrently.

be developed. This process of iteratively decomposing a robot control problem results in layered, hierarchical system designs.

The first stage of Go-Design, "basic-design", provides a set of guidelines for constructing a skill architecture. Designing a skill architecture involves:

1. Decomposing the problem (the context-level skill) into subskills.

2. Identifying and defining the transitions between skills.

Throughout the remainder of this chapter we will discuss the guidelines the methodology offers to assist the developer through this design process. The example we will use in this chapter is designing a robot's mind which allows the robot to find and kick a ball into a goal on a soccer field - a simpler, more feasible task than building a robot capable of performing grocery shopping.

6.5.1 The Context-Level Skill

The first step in designing a robot's mind is to identify the *context-level skill.* We use the term context-level skill to describe the mind's required capabilities, i.e. the entire behavioural and functional capabilities of the system. The context-level skill can be likened to the context-level diagram of a data-flow-diagram or the root of a tree - the context-level skill is the starting point for the iterative decomposition process.

6.5.1.1 Notation: Context-Level Skill

To identify the context-level skill, the skill's numeric identifier is set to "0" (as per the convention used in data-flow-diagrams). Consider designing a robot whose sole purpose is to kick goals on a soccer field. The context-level skill for such a robot is displayed in Figure 6.4.

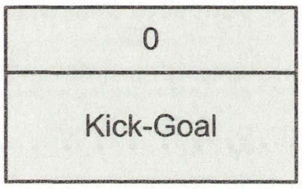


Figure 6.4: A context-level skill diagram for a goal-kicking soccer robot.

6.5.2 Iterative Skill Decomposition

The context-level skill represents the objectives and requirements of the system, the robot, the environment, and the robot's current sensory and effector capabilities[2]. The designer must now decompose the context-level skill into a set of implementable, collaborating skills. Decomposition is a top-down process which involves breaking a large problem into a set of smaller (and hopefully simpler) problems. *Iterative* decomposition is a process of, in turn, breaking each new skill down into new skills, which are ultimately capable of being implemented. Decomposition ends when we have a design in terms of (feasible) skills the robot can do.

6.5.3 Identifying Skills

The first step in problem decomposition is to identify a skill sequence - a set of steps, some of which may run concurrently, that achieve the agent's objectives. The context-level skill diagram in Figure 6.4 reveals little of how the robot *should* kick a goal, or how the robot *could* kick a goal, or when it should even *stop* kicking goals. Skill decompositions are achieved by the designer embodying themselves in the problem, and thus asking "what would I do in the same situation?". When identifying subskills, it is important to remember skills *do* things, and thus a skill name should

[2]Of course, this information is not contained in the skill diagram, but is identified through context-level analysis - see Chapter 5.

always include a verb. The aim is to identify subskills that conceptually help solve the problem (i.e. the current skill). The following steps can assist the designer in identifying skills:

- Look for *verbs* - skills "do something".
- Look for *nouns* - the objects that are having something done to them (by the verb), e.g. "drive the *car*", "kick the *ball*".
- Skills should be named by their core and essential functionality, i.e. their objective, for example "`Get-Ball`" isn't "`Walk-To-Ball`" because we don't want to limit or restrict *how* we might get the ball. In other words, name skills in terms of *what* they achieve, rather than *how* they achieve it.
- Consider if each subskill should be generalised (i.e. abstracted). For example, should "`Get-Ball`" be "`Get-Object`"? In answering such questions the developer must weigh up the benefits of reuse versus the benefits of specialisation. In our current example, as the only object our robot will ever get is a ball, there is no need to generalise. Also, for an abstraction such as `Get-Object` to be created, a parameter to represent the type of object to be "got" needs to be created. These topics are discussed in more detail in Chapter 7.

6.5.3.1 Example

An example skill sequence for the first decomposition of `Kick-Goal` can be seen in Figure 6.5. Note, however, that a skill can be decomposed in different ways. For example, compare Figure 6.5 with Figure 6.6. In Figure 6.6 `Find-Ball` is a subskill of `Get-Ball`, rather than a subskill of `Kick-Goal`. While there is no provably "right" way of decomposing a skill, the different approaches presented in Figures 6.5 and 6.6 have different advantages and disadvantages. For example, in more complex diagrams (complex in that there are many skills and many transitions between those skills), "hiding" a skill by encapsulating it inside another (as in the case of `Find-Ball` being a subskill of `Get-Ball` in Figure 6.6) can make diagrams conceptually simpler. However, this approach is not suitable if a skill needs to interact with other skills at the same level of the skill tree - for example, if `Find-Ball` and `Kick` need to interact directly, then placing `Find-Ball` as a subskill of `Get-Ball` is not appropriate.

6.5.4 Skill Templates

So far we have only considered how to decompose a skill into subskills, and how to represent this decomposition using a skill diagram. The next step in the diagramming process is to identify

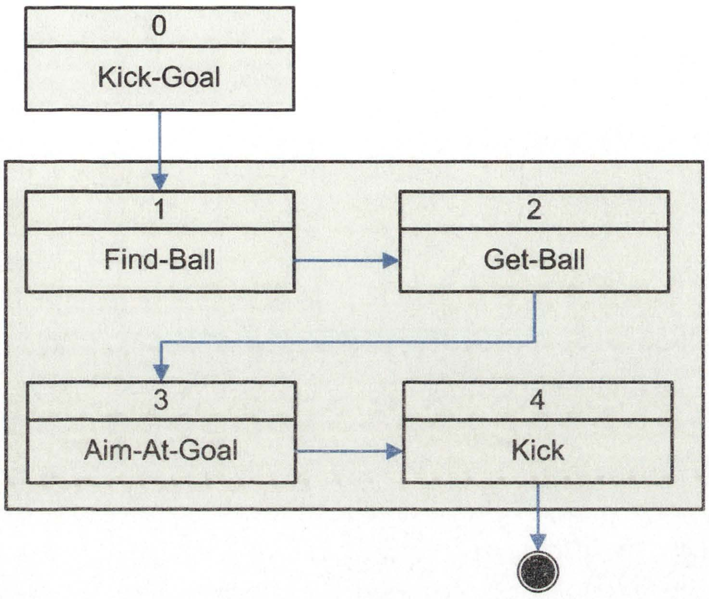


Figure 6.5: A skill sequence for kicking a goal. The skill diagram lacks transition-conditions.

skill-transitions and their transition-conditions. However, to identify and model skill-transitions effectively, we need to identify *what* each skill should achieve (each skill's objectives and requirements), and *when* each skill should operate (i.e. the state of the world when the skill's use is appropriate; its "usage context"). To capture such information, Go-Design provides *skill templates* - documents which prompt the designer to identify key aspects of each skill. While many aspects of each skill's nature are implicit in their naming (e.g. the objective of `Get-Ball` is obviously "to get the ball"), and thus also implicit in the mind of the designer, skill templates provide a means for *explicitly* documenting, in detail, aspects of each skill which can not be easily captured in a skill diagram. Explicitly representing information such as each skill's objectives, requirements, and performance criteria is invaluable in team environments, when different developers are developing different, but inter-related and inter-dependent skills.

Skill templates describe:

- **The *objective* of the skill** - i.e. *why* is the skill being built? In other words, for what context-level requirement is it being built to (at least partially) satisfy?
- **A *description* of the skill** - *how* will the skill achieve its objective?
- ***The skill's entry-conditions and exit-conditions*** - i.e. *when* the skill should and should not operate - imperative for establishing skill-transitions and their transition-conditions.

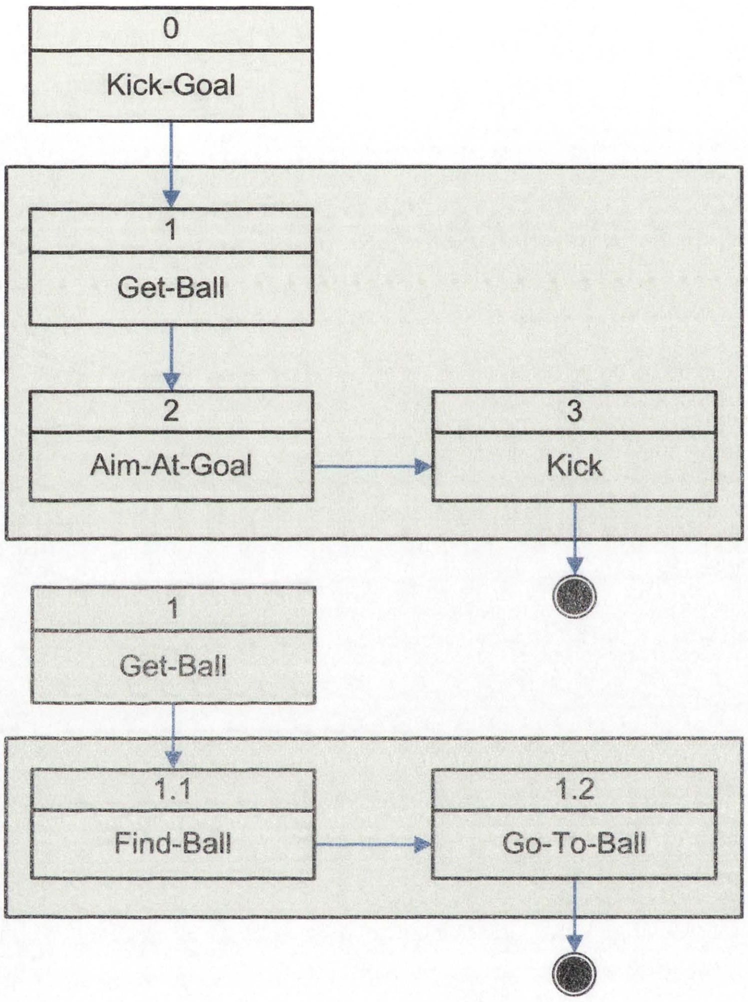
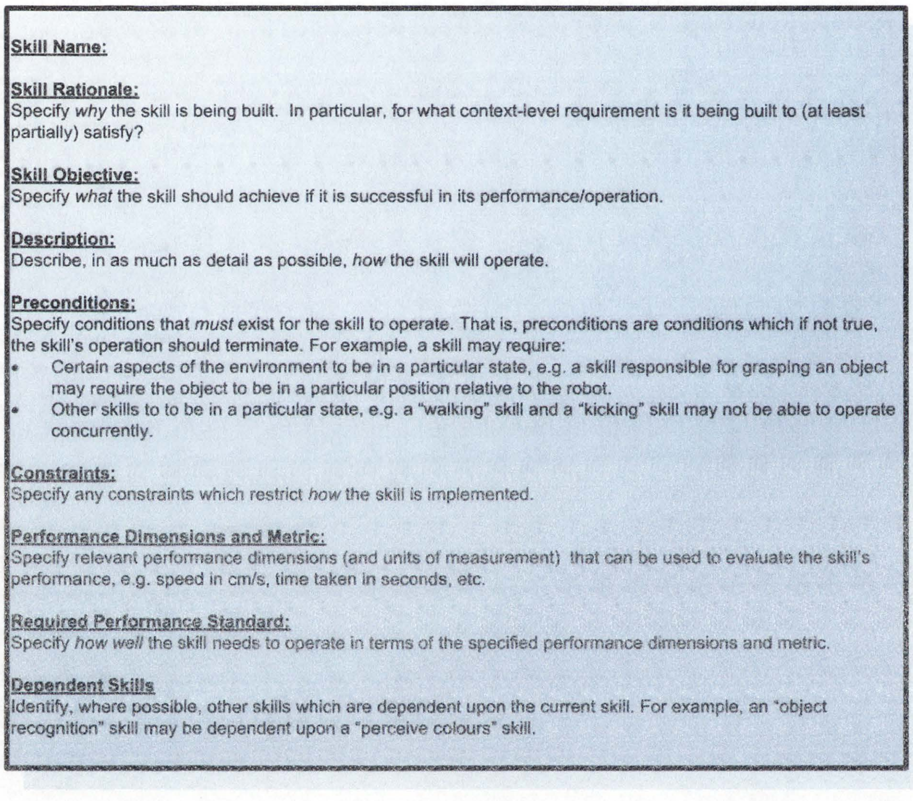


Figure 6.6: An alternate skill diagram for kicking a goal. `Find-Ball` is now a subskill of `Get-Ball`. The skill diagram lacks transition-conditions.



Skill Name:

Skill Rationale:
Specify *why* the skill is being built. In particular, for what context-level requirement is it being built to (at least partially) satisfy?

Skill Objective:
Specify *what* the skill should achieve if it is successful in its performance/operation.

Description:
Describe, in as much as detail as possible, *how* the skill will operate.

Preconditions:
Specify conditions that *must* exist for the skill to operate. That is, preconditions are conditions which if not true, the skill's operation should terminate. For example, a skill may require:

- Certain aspects of the environment to be in a particular state, e.g. a skill responsible for grasping an object may require the object to be in a particular position relative to the robot.
- Other skills to to be in a particular state, e.g. a "walking" skill and a "kicking" skill may not be able to operate concurrently.

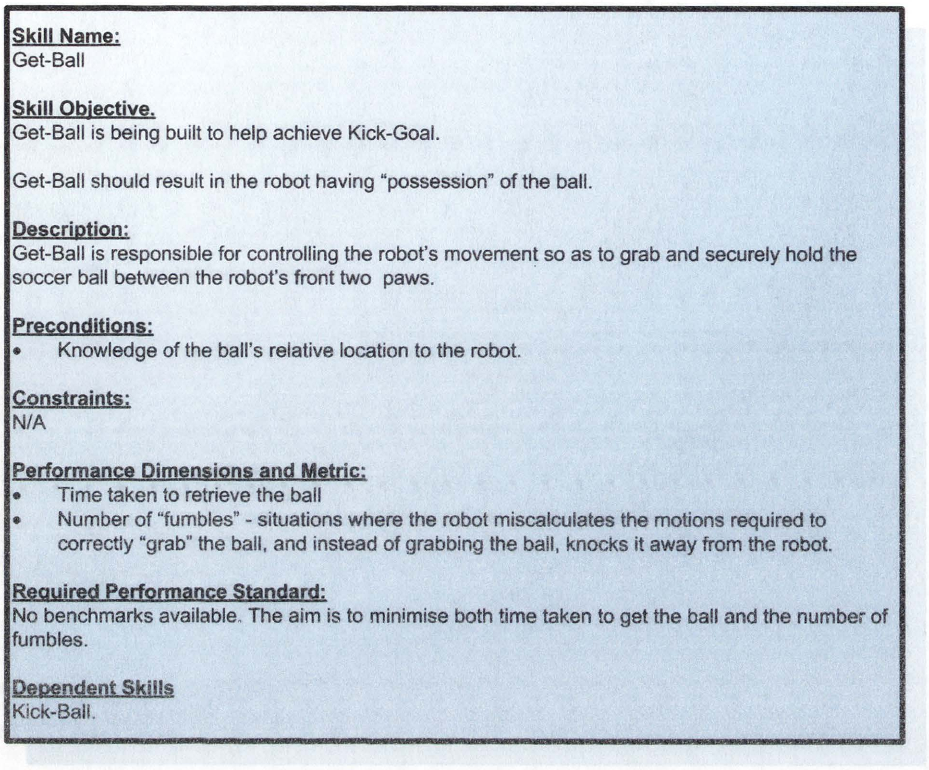
Constraints:
Specify any constraints which restrict *how* the skill is implemented.

Performance Dimensions and Metric:
Specify relevant performance dimensions (and units of measurement) that can be used to evaluate the skill's performance, e.g. speed in cm/s, time taken in seconds, etc.

Required Performance Standard:
Specify *how well* the skill needs to operate in terms of the specified performance dimensions and metric.

Dependent Skills
Identify, where possible, other skills which are dependent upon the current skill. For example, an "object recognition" skill may be dependent upon a "perceive colours" skill.

Figure 6.7: A skill template.



Skill Name:
Get-Ball

Skill Objective.
Get-Ball is being built to help achieve Kick-Goal.

Get-Ball should result in the robot having "possession" of the ball.

Description:
Get-Ball is responsible for controlling the robot's movement so as to grab and securely hold the soccer ball between the robot's front two paws.

Preconditions:
- Knowledge of the ball's relative location to the robot.

Constraints:
N/A

Performance Dimensions and Metric:
- Time taken to retrieve the ball
- Number of "fumbles" - situations where the robot miscalculates the motions required to correctly "grab" the ball, and instead of grabbing the ball, knocks it away from the robot.

Required Performance Standard:
No benchmarks available. The aim is to minimise both time taken to get the ball and the number of fumbles.

Dependent Skills
Kick-Ball.

Figure 6.8: Skill template for `Get-Ball`

- **The *constraints* of the skill** - restrictions and limitations governing how the skill is implemented.
- **Performance dimensions and metric** for assessing the skill's performance.
- **The required performance standard**, described in terms of the performance dimensions and metric.
- Identification, where possible, of any **dependent skills**.

A skill template is displayed in Figure 6.7.

6.5.4.1 Example

Figure 6.8 contains an example skill template for `Get-Ball`.

6.5.5 Identifying Skill-Transitions

Decomposing skills into subskills is meaningless without defining how those skills interact. Skill-transitions represent relevant and important events, which we also term "*whens*"[25]. Thus, identifying transition-conditions involves identifying when one skill should end, and when another should start. For example, in Figure 6.5 there are no transition-conditions between any of the subskills. Thus, in Figure 6.5, when should `Find-Ball` stop and `Get-Ball` start? When should `Get-Ball` stop and `Aim-At-Goal` start? When should the robot `Kick`? And how do we know when the kick is completed? Thus, we need to define the conditions for which each skill terminates and/or passes control to another (i.e. flow-of-control). Simply put, there are two types of transition condition - *entry-conditions* and *exit-conditions*, with the exit-condition of one skill being the entry-condition for another. When identifying skill-transitions, the designer should note that *every* skill (except for the context-level skill), should have at least one entry-condition (else, how does it start?).

6.5.5.1 What-If Analysis

It is imperative that appropriate skill-transitions are identified to design robust robot skills, i.e. so that the robot performs the "right" skill at the "right" times. Therefore, identifying skill-transitions is related to the problem of action-selection - of *when* to do *what*[25]. *What-if analysis* is a brainstorming approach in which the designer considers possible or potential situations or scenarios that may affect system performance. What-if analysis involves the designer hypothesising about *relevant* potential events, i.e. "what happens if ...?". Failure to identify all relevant skill transitions can result in the robot failing to respond appropriately to real-world events, negatively effecting the robot's performance. For example, the robot may perform skills at inappropriate times, get stuck in endless inappropriate repetitive loops, or become "frozen" in a particular state (e.g. because none of the exit-transitions for that particular skill hold true). Thus, what-if analysis is a crucial aspect of grounding by design, in that the designer is responsible for making decisions regarding relevant events, and later in detailed-design, decisions regarding how to perceive such events. The poorer an agent's grounding, the more likely the need for incremental code-and-fix, trial-and-error, revisions of the robot's software due to unanticipated agent-environment interactions. To help avoid this, a "what-if analysis checklist" is presented in the following subsection to assist the designer in eliciting relevant skill-transitions.

6.5.5.2 What-If Analysis Checklist

Qualitative assessments of how a skill can terminate can assist in identifying skill-transitions. For example:

- **What happens if the skill is successful?** Consider the skill's objective or objectives - if the objective or objectives are achieved, what should happen next? Should there be an exit-condition to another skill?

- **What happens if the skill is unsuccessful?** Skills will not always achieve their objective. Robust robot behaviours are heavily reliant upon the designer identifying all the different ways each skill can fail, and designing a strategy to cope with these situations. Common types of unsuccessful exit conditions include:

  - **What happens if the skill "times out"?** In other words, can the skill take too long to achieve its objective, and if so, what should happen?
  - **What happens if a skill's entry-condition is no longer true during the skill's operation?** When designing skills, the designer should consider what knowledge is required to execute that skill (note - in Chapter 7 representation design is covered in detail). What should happen if a skill, once started, lacks insufficient knowledge to complete the skill? For example, for our goal-kicking robot to be able to aim at the goal (i.e. `Aim-At-Goal`), the robot must possess knowledge of the goal's location, relative to the robot. What happens if this knowledge is not available? Is it the responsibility of `Aim-At-Goal` to find the goal? Likewise, what happens if during the execution of `Get-Ball` knowledge of the ball's location becomes uncertain or unknown?
  - **What happens if new events render the skill's continued operation irrelevant?** There are a large (probably infinite) number of potential events that could occur which could make any (or all) of the skills' continued operation irrelevant (e.g. the robot could physically break-down, or the sun could explode) - the task of the designer is to use their understanding of the domain to identify *likely* events. For example, what happens if our soccer-playing robot falls over (e.g. due to an unstable walking gait or a knock from another robot) during the execution of any of the skills?

6.5.5.3 Examples

- **What happens if the skill is successful?** Figure 6.9 displays a revised skill diagram for `Kick-Goal` in which transition-conditions are added for each of the skill-transitions which reflect each skill's objective (success conditions).

- **What happens if the skill "times out"?** Figure 6.10 displays a revised skill diagram which incorporates time-out conditions for the skills `Find-Ball` and `Aim-At-Goal`. Our goal kicking robot will now simply stop if either Find-Ball can not find the ball (i.e. `unable-to-find-ball` is true), or Aim-At-Goal can not find the goal (i.e. `unable-to-find-goal` is true).

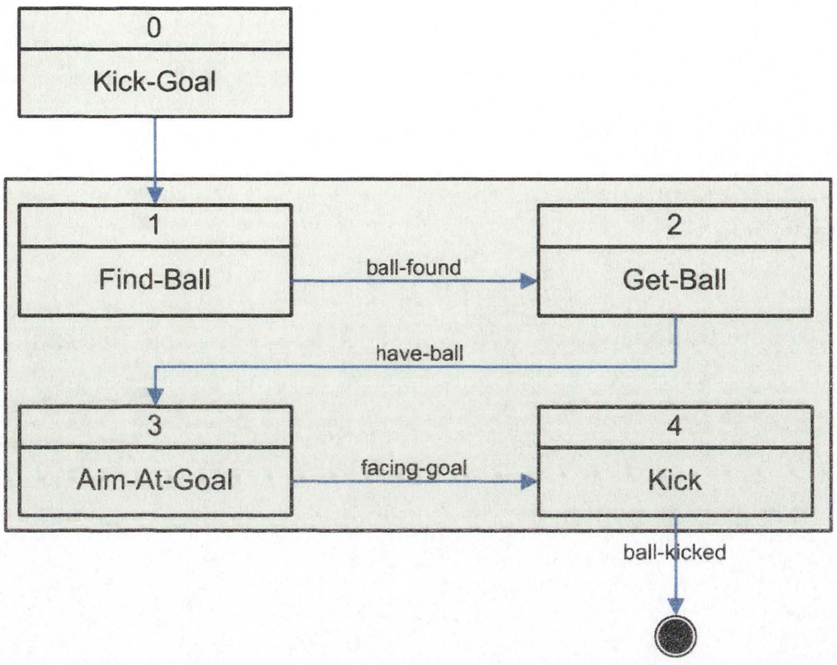


Figure 6.9: A skill sequence for kicking a goal with "successful" transition-conditions.

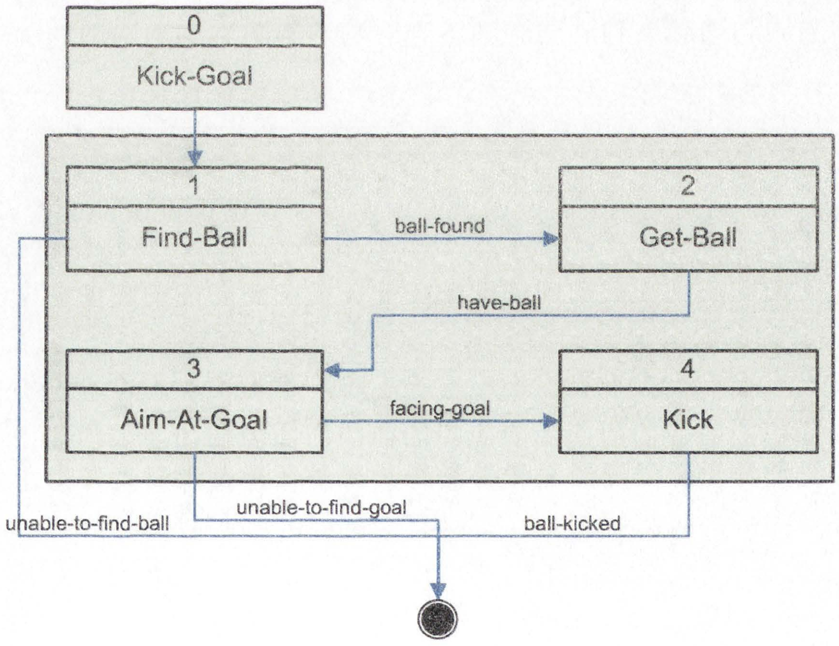


Figure 6.10: A skill diagram for `Kick-Goal` with the new "unsuccessful" transition-conditions `unable-to-find-ball` and `unable-to-find-goal`.

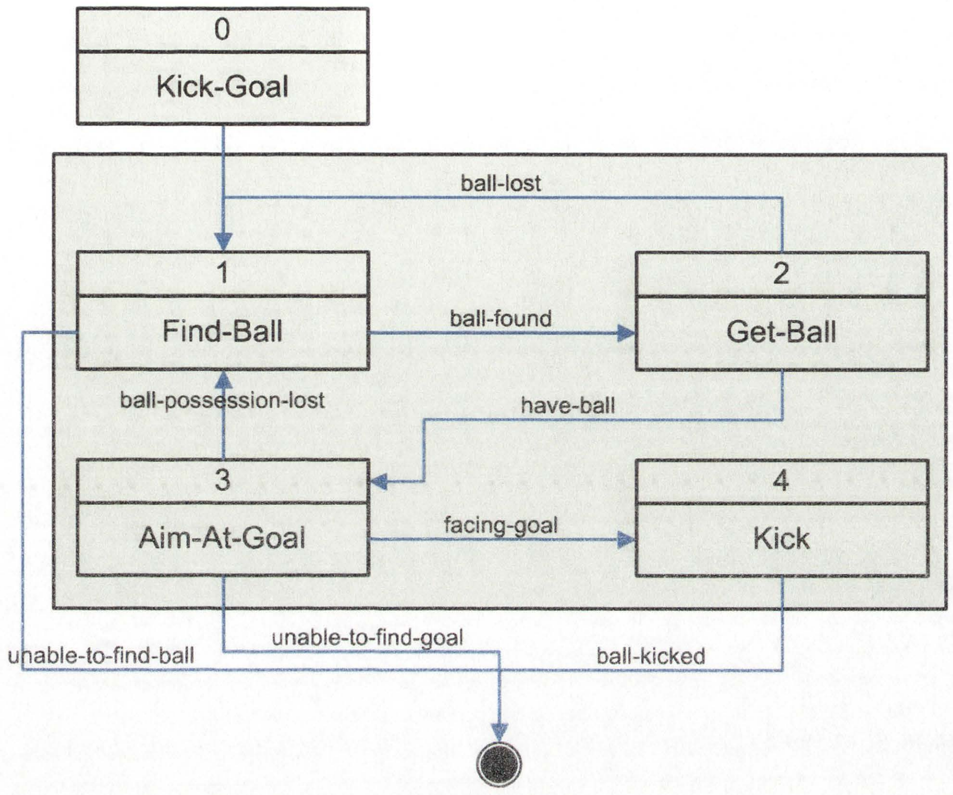


Figure 6.11: A skill diagram for `Kick-Goal` after the addition of the "unsuccessful" skill-transitions `ball-possession-lost` and `ball-lost`.

- **What happens if a precondition for the skill's operation is no longer true?** Consider the transition-conditions `ball-found` and `have-ball` in Figure 6.10. The condition `ball-found` implies that the robot knows where the ball is, and `have-ball` implies the robot has control or possession of the ball. What happens if the whereabouts of the ball becomes unknown during `Get-Ball`? Or alternatively, what happens if possession of the ball is lost before `Aim-At-Goal` is completed? Figure 6.11 contains two new skill-transitions to cater for these situations - `ball-lost` passes control from `Get-Ball` to `Find-Ball` if knowledge of the ball's location becomes too uncertain or unknown, while `ball-possession-lost` passes control from `Aim-At-Goal` to `Find-Ball` should control of the ball be lost while aiming at the goal.

- **What happens if new events render the skill's continued operation irrelevant?** In robotic soccer collisions with other robots can often knock a robot over (onto its side), rendering all other skills irrelevant until the robot is back on its feet. Figure 6.12 has a new

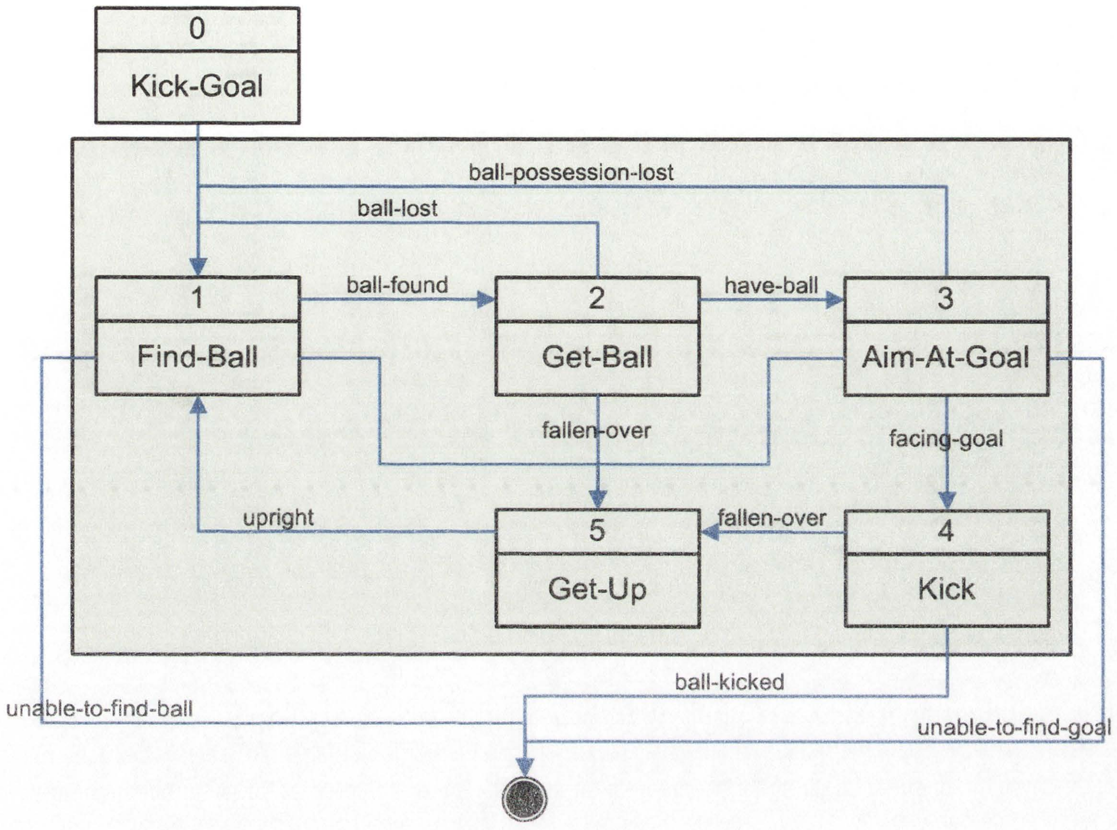


Figure 6.12: `Kick-Goal`, now with the ability to `Get-Up` after falling over.

skill - the ability to `Get-Up` after a fall.

6.5.5.4 Skill-Transitions - Summary

In this section we have seen how to add skill-transitions and "basic" transition-conditions to a skill diagram. The transition-conditions, however, are lacking in detail, being merely labels representing real-world events to the designer. Thus, more detail for transition-conditions is required - in particular, processes for testing the truth value of these transition-conditions need to be defined. For example, what does `ball-found`, `have-ball`, `facing-goal` and `ball-kicked` mean? Is the ball "found" if there are a few pixels of the appropriate colour in the robot's image? At what point does the robot "have" the ball? And is the robot "facing" the goal if the goal is a few degrees to the robot's left or right? Thus, defining the implementation of transition-conditions is required to define what is *precisely meant* by a transition-condition. This topic will be covered in detailed-design in Chapter 7.

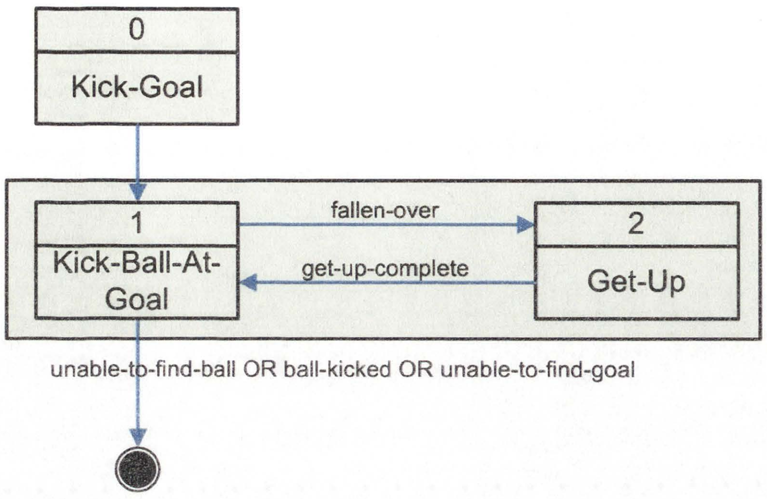


Figure 6.13: `Get-Up` separated from `Kick-Goal`.

6.5.6 Design: Keeping it Simple

As we can now see, our diagrams are now starting to get a little complicated (especially that in Figure 6.12). To keep diagrams simple, there are two main techniques that can be employed, both of which involve simplifying existing skill diagrams by creating new, smaller skill diagrams.

6.5.6.1 Layering

Consider Figure 6.12. Observe that *every* skill in the skill diagram has the transition `fallen-over` to the skill `Get-Up`. This indicates that the `Get-Up` ability should be triggered whenever the robot falls over, regardless of which particular skill is currently being performed. Thus, Figure 6.12 can be simplified by creating a second skill diagram in which the need to `Get-Up` after falling over will override any currently operating skills.

Figure 6.13 displays a new skill diagram in which a new skill, `Kick-Ball-At-Goal`, encapsulates all the skills and skill-transitions contained in Figure 6.12, *except* `Get-Up` and its related skill-transitions. In Figure 6.13, if the robot ever falls over during the process of kicking the ball it will `Get-Up` before continuing. Figure 6.13 can be further elaborated upon, by creating a new skill `Perceive-Posture`, which is responsible for determining if the robot has `fallen-over` or is `upright`, as shown in Figure 6.14. In this skill diagram, there exists a skill-transition without a transition-condition (from `Kick-Ball-At-Goal` to `Perceive-Posture`) - the absence of a transition-condition indicates that the transition-condition is *always* true, and thus every time (i.e. every "frame" or processing cycle) `Kick-Ball-At-Goal` executes, `Perceive-Posture` is called. Thus, `Perceive-Posture`

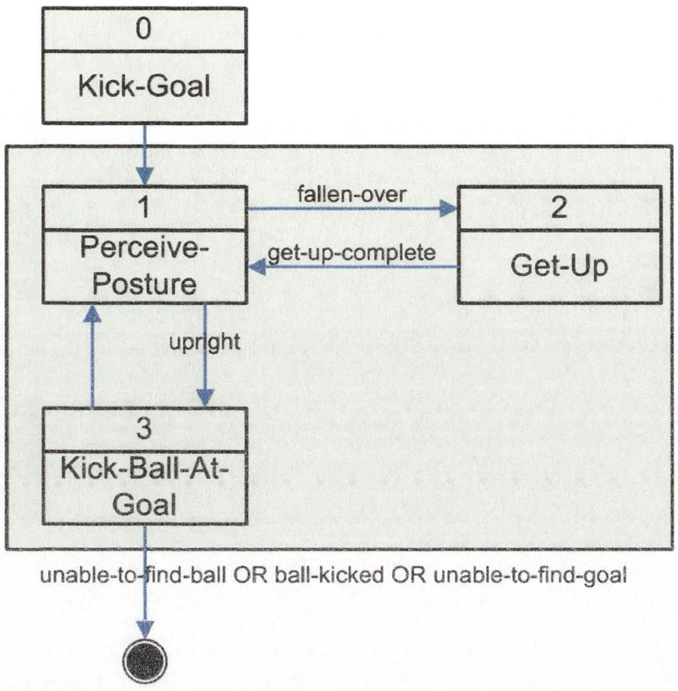


Figure 6.14: `Perceive-Posture`.

acts like a reflex, triggering the `Get-Up` skill whenever the robot has `fallen-over`.

Lastly, Figure 6.15 displays the skill diagram for `Kick-Ball-At-Goal` - the new skill formed from the separation of `Get-Up` from `Kick-Goal`.

6.5.6.2 Amalgamation

Another technique to keep diagrams conceptually simple is to amalgamate skills, thus creating new skill diagrams, but with each skill diagram containing fewer skills. For example, consider Figure 6.16. The skills `Aim-At-Goal` and `Kick` are amalgamated to create a new skill `Aim-And-Kick`, thus reducing the number of skills by one, and hiding the skill-transition `facing-goal` within `Aim-And-Kick`. Likewise, Figure 6.17 displays the further simplification of `Kick-Ball-At-Goal` through the amalgamation of `Find-Ball` with `Get-Ball`, which again reduces the number of visible subskills by one, and also hides the transitions `ball-found` and `ball-lost` within the new version of `Get-Ball`.

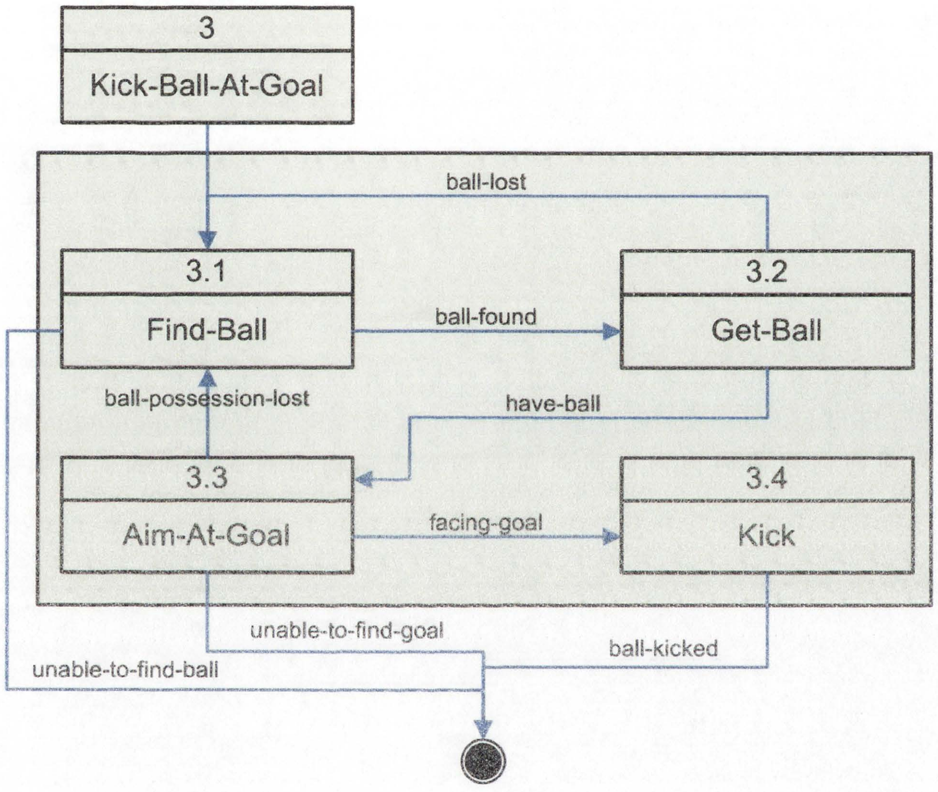


Figure 6.15: `Kick-Ball-At-Goal` - now without `Get-Up`.

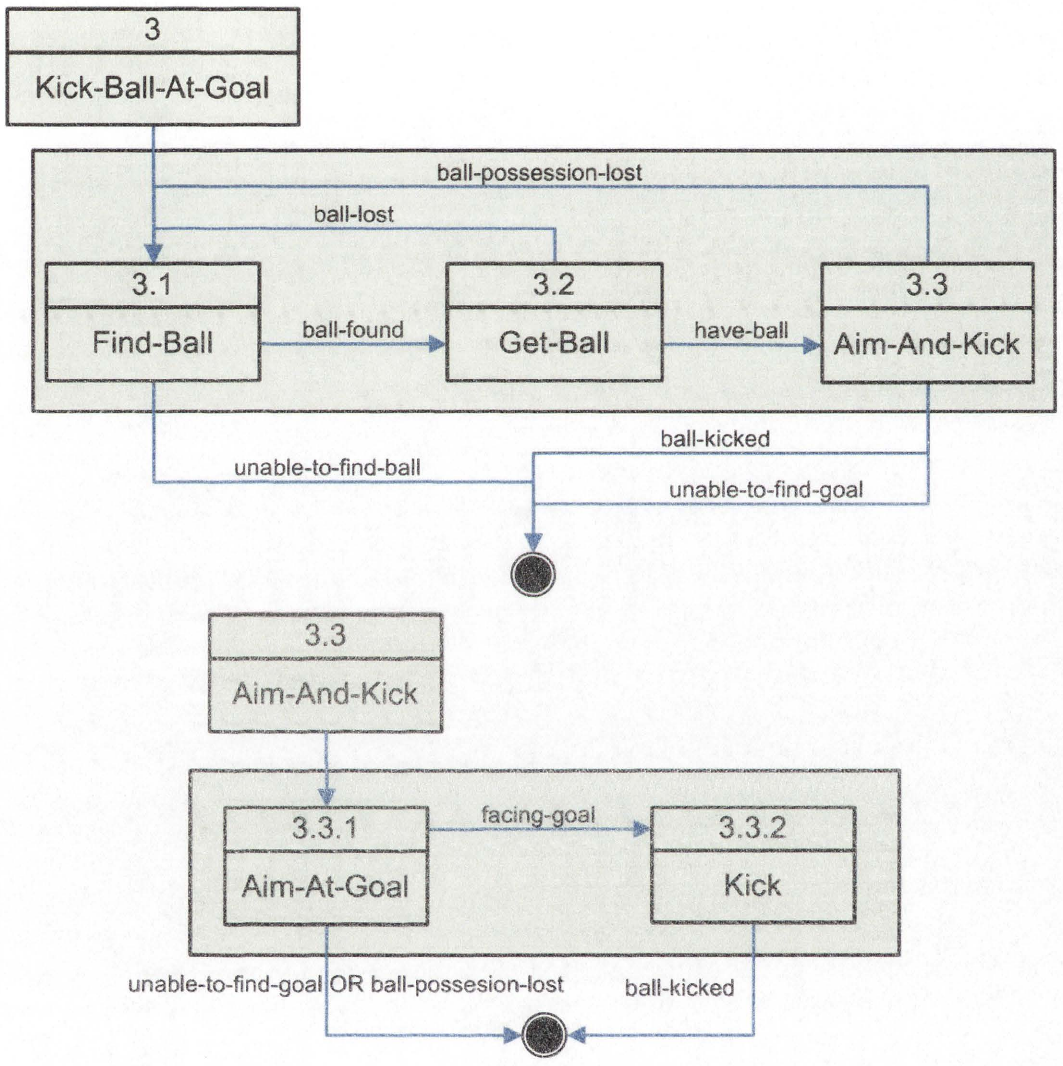


Figure 6.16: Simplified `Kick-Ball-At-Goal` (top) with `Aim-At-Goal` and `Kick` amalgamated into `Aim-And-Kick` (bottom).

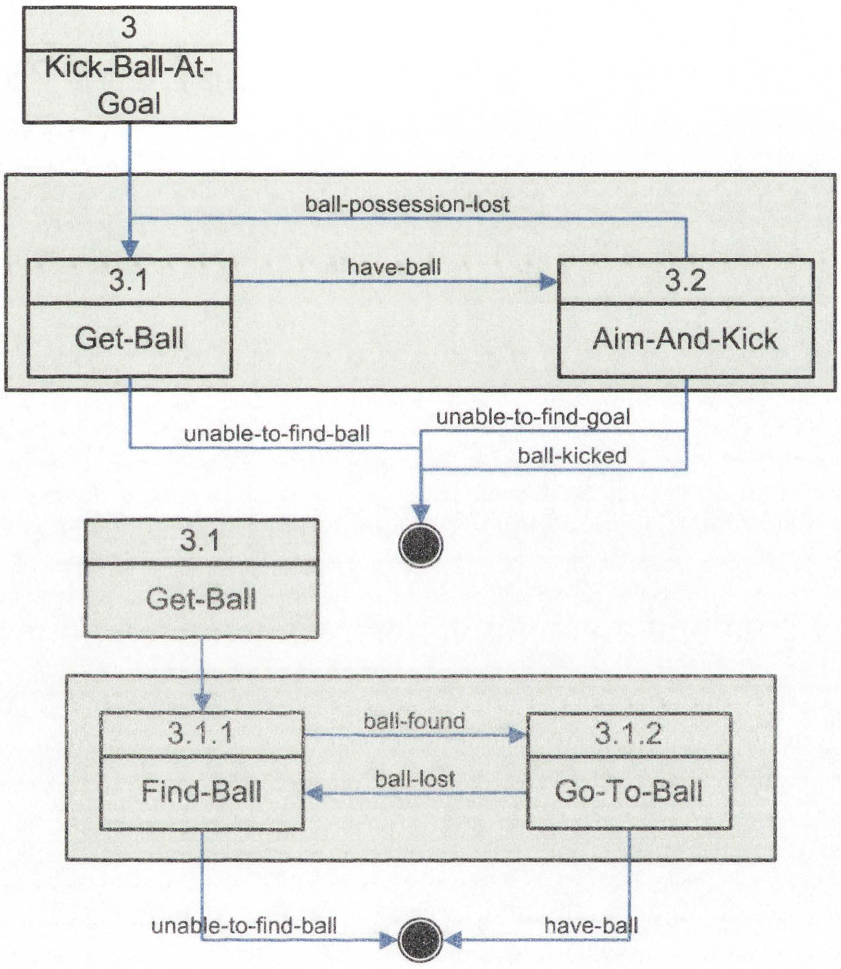


Figure 6.17: Further simplified `Kick-Ball-At-Goal` (top) with `Find-Ball` and `Get-Ball` amalgamated (bottom).

6.6 Summary

The notation and processes we have introduced in this section (skills and skill-transitions) enable us to create skill-based architectures for solving robot control problems. The skill architectures allow the designer to model hierarchical, modular, capability-based skills, and the interactions between these skills through event-based skill-transitions and transition-conditions. As illustrated in Figure 6.14, skill-transitions provide a means for further decomposition of skill diagrams, through the identification of perceptions to detect the events represented by transition-conditions. In Chapter 7 we turn our attention to detailed-design - a process which involves identifying skill types, representation, and knowledge-flows and implementation strategies in terms of pseudocode.

Chapter 7

Grounding Oriented Design: Part II - Detailed-Design

In the previous chapter we considered basic-design - a set of processes which allow the designer to construct a *skill architecture*, consisting of skills and skill transitions. While a skill architecture can play a role in the high-level design of an autonomous robotic system, for the purpose of implementing a software solution more detail is required. Thus, a *detailed-design* is needed which provides a blueprint for translating a skill architecture into software.

7.1 Detailed-Design

In order to design a grounded agent, the designer must understand both the concepts the agent needs to "understand", and the decision-making processes which rely upon those concepts. In detailed-design, we consider how to identify the knowledge, concepts, perceptions and decision-making processes required for a robot's mind to control the robot appropriately. In particular, detailed-design allows the designer to produce a grounded system design which can be easily implemented (i.e. a pseudocode level of detail). Detailed-design involves identifying each skill's "type", the requirements of each skill (including *groundedness*), the representation each skill requires (*concepts*, *percepts* and *memories*), the interfaces between skills, and the dependencies between skills through the use of *flows*.

In this Chapter we will introduce these concepts, and will continue the refinement of the examples presented in the previous chapter.

7.2 Skill Types

The first step of detailed-design is to identify each skill's *type*. Go-Design differentiates between four types of skills - *decisions*, *actions*, *perceptions* and *behaviours*. Every skill falls into one of these categories. Collaborations between these different skill types are used as the starting point in designing and modeling the required decision-making and representation for the robot control problem.

7.2.1 Decisions

Decision-making is at the crux of intelligent behaviour, with decision-making playing a role in both the development of robotic minds (i.e the "design-time" decisions a developer makes about a robot's mind), and during the operation of that robot's mind (i.e. the "run-time" decisions a robot makes autonomously). Decisions range from high-level (even abstract or profound) reasoning (e.g. "what happened?", "what should I do now?", "why am I here?", etc) to, at their lowest-level, simple "if statements" in code (e.g. "*if x is true then do a, else do b*"). Go-Design concerns two main types of decisions, namely *control management* and *knowledge management.* Control management decisions involve choosing between skill-transitions, while knowledge management concerns the manipulation of representation (internal state).

While many decision-making processes in software can be relatively simple (e.g. an if-statement), many decisions can be decomposed (or alternatively are composed of other decisions). For example, consider a soccer playing robot, in possession of the soccer ball, that must decide "should I kick now?" - such a decision may be dependent upon the evaluation of other decisions such as "can I see the goal?", "am I facing the goal?", "are there any robots obstructing my shot at goal?", and so forth.

7.2.2 Actions

Actions are commands which:

1. Execute without perception.

2. The developer can assume to be *deterministic* - that is, the designer can assume they "work" (but of course they might not, due to wear-and-tear or damage, etc).

Simple examples of actions include setting the boolean states of actuators such as an AIBO's ears (up or down), an AIBO's mouth (open or shut), or turning a light on or off. Other examples include setting the joint angle of a robot's limb, or playing a particular audio file, and so forth. In Go-Design, actions are not limited to those that have physical effects upon the world - they can also be virtual or software-based (e.g. saving a file to disk, or sending an email). Ultimately, a skill decomposition must map to actions the robot can do. Note, while actions may be perception-less, many actions can be composed of both decisions and other actions. For example, an action which enables a legged robot to "walk one step" could be decomposed into a sequence of actions which sets the angle of each leg over time. The term *primitive-action* refers to actions that can not be decomposed, e.g. actions provided by the robot's programming interface. As actions have no perception capabilities, actions can not evaluate their own performance. Thus, if evaluation of an action's performance is required, it must be provided by another type of skill.

7.2.3 Perceptions

Perceptions are decisions made about the state of the world. We use the term *perception* to refer to the process of perceiving, and the term *percept* to refer to the memory (the persistent state) of that perception. We make the distinction between decisions and perceptions (a type of decision) due to our concern with grounding. However, perceptions rely upon interpreting both sensory information and existing knowledge to make their decisions regarding the state of the world.

7.2.3.1 Sensations versus Perceptions

Sensations are raw, unprocessed data from the robot's sensors, such as a stream of images from a camera. Thus, perceptions and sensations differ in that perceptions involve making decisions and inferences concerning the state of the world in response to the "value" of sensations, whereas sensations represent raw, uninterpreted input to the robotic system. For example, consider a pixel in a robot's camera. That pixel will have a triplet, numeric value (e.g. "98, 67, 45"). The numeric value of the pixel is a sensation, but interpreting that numeric value as representing a colour (e.g. "red") is a perception.

7.2.4 Behaviours

Lastly, behaviours are complex processes which can incorporate all the different types of skills, such as the ability to "drive a car". Thus, a behaviour can always be decomposed. During the design process, often the "behaviour" label is given to skills we haven't designed how to do yet, as all behaviours can ultimately be decomposed into decisions, perceptions and actions.

7.3 Knowledge Representation

Skills are processes - that is, they *do* things. The "things" they can do include both modifying knowledge, and using knowledge to make decisions. In Go-Design, knowledge is represented by *concepts*, *percepts* and *memories*. Concepts are akin to the classes of object-oriented programming - empty data structures, which when populated (or instantiated in the case of object-oriented programming) are represented as either a percept or a memory. Thus, concepts define the structure of memories, sensations and percepts. In other words, concepts describe what the robot is capable of knowing, while memories and percepts describe what the agent actually knows. Percepts are produced by perceptions, and represent the robot's beliefs about the state of the world. Memories are internal state - variables storing values. Memory, in robot control programs, can take many forms, from simple integer counters or boolean flags, to program state (e.g. memory of what the robot *was* doing is used to decide what to do *now*), or even procedural memory (e.g. *how* to do something). Lastly, note that perceptions are a form of memory - the distinction between memory and perception is made to force the designer to consider the implications of the quality of perceptions, and thus groundedness, on decision-making.

7.4 Skill Diagrams - Detailed-Design

A number of new concepts have been introduced in detailed-design, such as skill types (decisions, actions, behaviours and perceptions), and knowledge representation (concepts, percepts and memories). In this section we discuss how to represent these concepts diagrammatically.

7.4.1 Flows

In the previous chapter (basic-design), we saw how skill collaborations can be modeled through the use of skill-transitions, which are a *control-flow*. Control-flows represent when program control is passed from one skill to another. In this chapter we introduce *knowledge-flows*. There are two types of knowledge-flows - *information-flows* and *concept-flows*. Diagrammatically, flows can be thought of as "connectors", connecting skills, memories, percepts, and concepts. Information-flows represent the flow of information throughout the system, such as the reading of a memory or percept, or the writing (modification) of a memory, percept or concept. An information-flow towards a memory or percept indicates the modification of that knowledge, while an information-flow away from the knowledge store indicates the reading of that knowledge. Lastly, concept-flows link concepts with the knowledge stores (percepts and memories) that rely on them. Figure 7.1 displays the diagrammatic notation for representing flows.

Flow	Explanation
$\xrightarrow{\text{an-event}}$	Skill-transitions are control-flows.
$\longrightarrow$	Condition-less skill-transitions are control-flows.
— — — →	An information flow. The arrow direction is used to indicate the reading or writing of memory.
······························	A concept flow. A concept flow links a concept with a percept, memory or other concept.

Figure 7.1: Flow types in Go-Design.

7.4.2 Skill Types

Figure 7.2 provides an overview of the skill-types, and the notation for representing them. Adding a "type" to a skill is a simple matter of prefixing the skill with the skill's type.

7.4.3 Concepts, Percepts and Memories

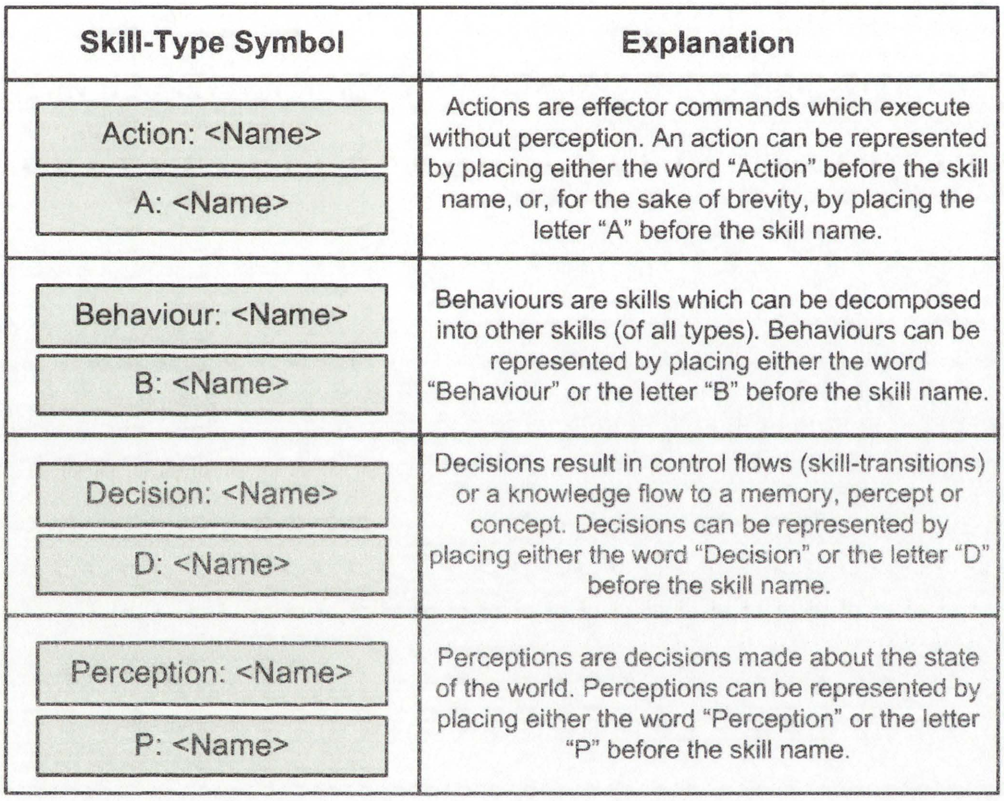
Figure 7.3 displays the notation for representing concepts, percepts and memories. As with skill types, knowledge types are identified by attaching the appropriate prefix to the concept, percept or memory. The details of each concept, percept or memory is listed below the knowledge name.

7.5 The Design Process

In basic-design (Chapter 6), a process of iterative decomposition was used to model each skill through interacting subskills, with interactions between subskills modeled through the use of skill-transitions. Similarly, detailed-design builds upon basic-design through a process of iterative decomposition, but differs in that skill types are identified, and the decomposition process now focuses on designing testable, implementable conditions for skill-transitions, together with the necessary skills for making the required decisions and perceptions about skill-transitions.

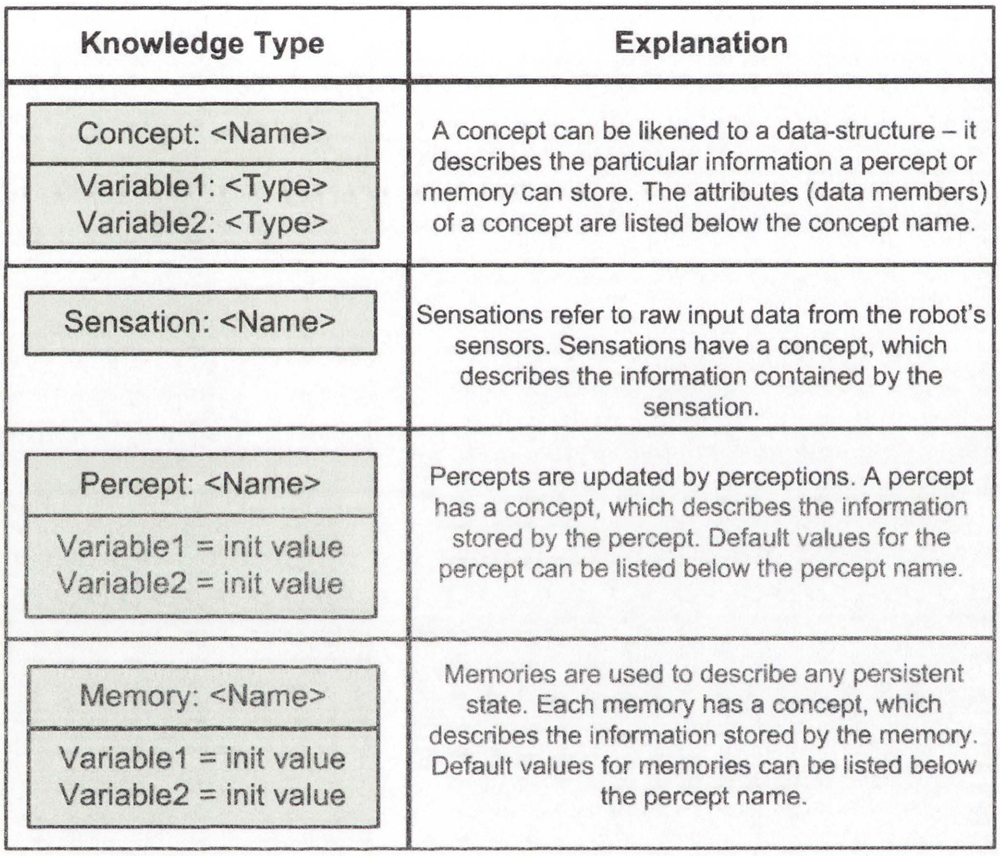
7.5.1 Identify Skill Types

The first step in detailed-design is to identify the skill types of the skills identified in basic-design. As Go-Design decomposes a problem in a top-down manner, during the initial stages of decomposition most skills will be behaviours or perceptions, whereas during later stages of decomposition, more



Skill-Type Symbol	Explanation
Action: <Name> A: <Name>	Actions are effector commands which execute without perception. An action can be represented by placing either the word "Action" before the skill name, or, for the sake of brevity, by placing the letter "A" before the skill name.
Behaviour: <Name> B: <Name>	Behaviours are skills which can be decomposed into other skills (of all types). Behaviours can be represented by placing either the word "Behaviour" or the letter "B" before the skill name.
Decision: <Name> D: <Name>	Decisions result in control flows (skill-transitions) or a knowledge flow to a memory, percept or concept. Decisions can be represented by placing either the word "Decision" or the letter "D" before the skill name.
Perception: <Name> P: <Name>	Perceptions are decisions made about the state of the world. Perceptions can be represented by placing either the word "Perception" or the letter "P" before the skill name.

Figure 7.2: Overview of skill types in Go-Design.



Knowledge Type	Explanation
Concept: <Name> Variable1: <Type> Variable2: <Type>	A concept can be likened to a data-structure – it describes the particular information a percept or memory can store. The attributes (data members) of a concept are listed below the concept name.
Sensation: <Name>	Sensations refer to raw input data from the robot's sensors. Sensations have a concept, which describes the information contained by the sensation.
Percept: <Name> Variable1 = init value Variable2 = init value	Percepts are updated by perceptions. A percept has a concept, which describes the information stored by the percept. Default values for the percept can be listed below the percept name.
Memory: <Name> Variable1 = init value Variable2 = init value	Memories are used to describe any persistent state. Each memory has a concept, which describes the information stored by the memory. Default values for memories can be listed below the percept name.

Figure 7.3: Knowledge Representation in Go-Design.

"primitive" skills such as actions and decisions will be identified, ultimately mapping to sensations and primitive-actions. The main difficulty in identifying skill types is in separating decisions and perceptions (as perceptions are a special type of decision, i.e. a decision about the state of the world). However, not all skill types need to be identified now, as identifying a skill type requires knowledge of how it will be implemented. If the designer is unable to identify a skill's type, decompose it. Be aware that a skill's type may change with different implementations. For example, a movement such as panning the robot's head (looking left to right, then back from right to left, and so forth) could be implemented as a series of motor positions, where each position is calculated by adding or subtracting (depending on the direction) a particular angle to the head's current position. One implementation may sense and perceive the head's current position (and thus be a *behaviour*), while another may never sense the head's current position, but instead use memory (i.e. a variable) to remember what the head's current position should be, based upon the value of the cumulative angle addition or subtraction (and thus be an *action*). Perhaps surprisingly, on the Sony AIBO, the latter approach of moving *without* actively perceiving the robot's head position results in smoother movement, whereas an approach which involves regularly perceiving the robot's head position results in "jerky" movement due to inaccuracies with the robot's sensors. Thus, labeling a skill's type is beneficial as it describes in a skill diagram aspects of the implementation of that skill, while hiding the mechanics of the implementation. Also, considering a skill's type is beneficial as it forces the designer to consider the skill's purpose, how it will be implemented, and its reliance upon and interaction with the environment.

7.5.1.1 Guidelines for Identifying Skill Types

To help identify skills, the following guidelines are provided:

- ***Behaviours*** - behaviours are composed of other skills, and thus need to be decomposed (unless the behaviour has been reused from a previous project, or is an external, 3rd party piece of software). Behaviours:
  - *Can always be decomposed* into other behaviours and/or perceptions.
  - *Require perception* to guide them - their performance is related to the state of the environment.
  - Are often *non-deterministic* (i.e. they may fail), e.g. due to incorrect assessment of the state of the environment.
  - *Take time* - they rarely occur instantaneously (as opposed to some actions, decisions, or perceptions, which may only take one processing cycle or frame to be completed).
  - Usually have a *physical effect* upon the world or the robot.

- ***Actions*** are commands which execute without perception, and are assumed by the designer to be deterministic. For example, commanding a motor on a legged robot to move to a particular angle, setting the colour of a pixel on a monitor, or making a sound through a speaker. While actions do not use perception, they may use memory and decisions. For example, an action such as a "kick" may be executed as a series of joint angles over time, with memory storing the previous position (which is used to calculate the next position), and the current position of the motors is never actually sensed or queried. Actions:
  - Are *perception-less.*
  - Are *deterministic*, but of course any action may, in reality, fail. Thus, in deciding whether a skill should be an action or behaviour, the designer needs to consider whether perception is required to detect failure, or whether the risk of failure is so small or irrelevant that it can be ignored in terms of design.
  - Can be *composed* of other actions and decision.
  - Are often (but not always) *procedural* in design, in that they involve sequences of other actions or decisions.
  - Can be *physical* or *virtual* in their effect. Actions can be physical, such as moving a limb, or software-based such as sending an email, or performing a calculation.
- ***Decisions*** - Decisions *choose* - either between skill-transitions, the value of variables (memory), or to change the structure of a concept. A skill is a decision if:
  - The skill involves *choosing* between different *skill-transitions.*
  - The skill can be framed as an "*if-then-else*" statement.
  - The skill concerns the problem of *action-selection*, i.e. "when" to do "what".
  - The skill sets the value of (non-perceptual) variables in memory.
- ***Perceptions*** - Perceptions are a special type of decision - a decision about the state of the world, at least partly based upon immediate (i.e. current) sensory or perceptive information, with that belief about the world being stored in a percept. A skill is a perception if:
  - It involves storing information about the state of the external world, based upon the robot's sensory experience of the world.

  A skill is *not* a perception if:
  - It concerns action-selection or behaviour-selection, e.g. "if the floor is dirty then clean it" is a decision, while the assessment of whether the floor is dirty is a perception.

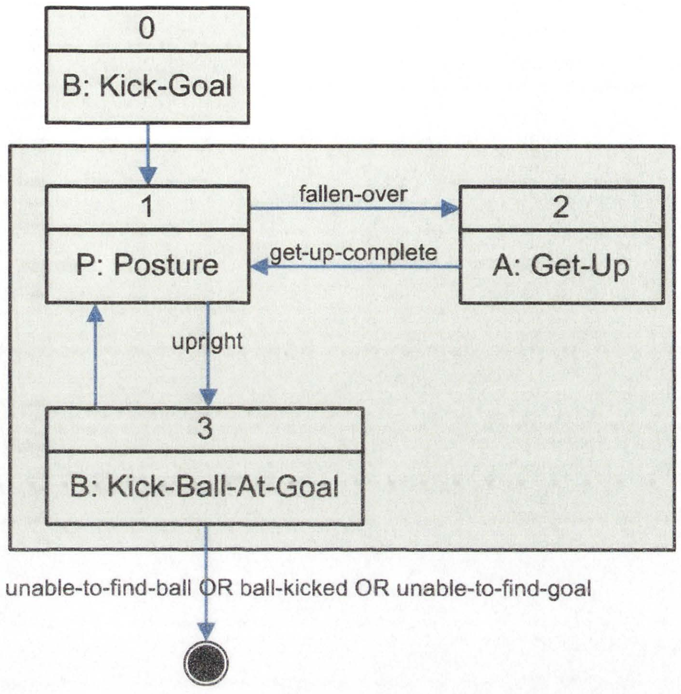


Figure 7.4: Skill types for Kick-Goal.

– Likewise, "can I see the ball?" is a decision if it involves the querying of memory, whereas the actual process of "seeing" the ball is a perception.

7.5.1.2 Example

Figure 7.4 displays the decomposition of the context-level skill `Kick-Goal`, with skill types added to the skills. More examples with skill-types will be presented throughout the chapter as we decompose `Kick-Goal`.

7.5.2 Decompose Transition-Conditions

The next step in the design process is to define transition-conditions. To be capable of implementation, transition-conditions must be testable, logical conditions. In Go-Design, as each skill is responsible for testing their own exit-conditions, each skill with an exit transition-condition must be decomposed and designed. In designing implementations for each transition-condition, the designer must consider what is precisely *meant* by each transition-condition. That is, as skill-transitions are events, the next design step is to specify precisely *when* a transition-condition is true, and *how* it will be detected. As we are concerned with "how", and thus implementation, this will require not only decomposition, but also identification of knowledge requirements, such as concepts (data structures), memories and percepts (internal state), and decision-making. Moreover, interfaces between skills will need to be defined - that is, how does one skill pass control to another? It is similar to defining the public method interface of a software module, except we are not only specifying the interface, but we are specifying *when* the methods of that interface are invoked, and by whom. Thus, when defining skill-transitions, the designer should specify precisely how and when one skill interacts with, or commands, another.

7.5.2.1 Example

In Figure 7.4 there are several transition-conditions, namely `fallen-over`, `get-up-complete`, `upright`, `unable-to-find-ball`, `ball-kicked`, and `unable-to-find-goal`. As each skill is responsible for testing their own exit-conditions, in Figure 7.4 `Perceive-Posture` is responsible for `fallen-over` and `upright`, `Get-Up` is responsible for `get-up-complete`, and `Kick-Ball-At-Goal` is responsible for `unable-to-find-ball`, `ball-kicked`, and `unable-to-find-goal`.

Let's consider `Perceive-Posture`, which is responsible for `fallen-over` and `upright`. When, precisely, is the robot `fallen-over`? When, precisely, is the robot `upright`? How will these events be perceived? Furthermore, are `fallen-over` and `upright`, in terms of representation, the converse of each other and simply the same boolean value, except one is when the condition is true and the other false? For example, perhaps `upright` is really "`fallen-over` is false"? Figure 7.5 displays a decomposition of `Perceive-Posture`. In this skill diagram (based upon a real implementation), two decisions are responsible for determining if the robot has `fallen-over`, namely `Accel-Over-Threshold` and `Counter-Over-Threshold`. Pseudocode illustrating the implementation of this decision-making is represented in Figure 7.6. Figure 7.5 also displays the concepts, percepts and memory required to support the decision-making process. Rather than have two separate variables - `fallen-over` and `upright` - a single boolean variable is used, where "`fallen-over` is false" is equivalent to "`upright` is true".

Figure 7.7 displays the skill diagram for `Get-Up`, which is responsible for the skill-transition `get-up-complete` (see Figure 7.4). In this diagram we see the representation of actions as "motions" - sequences of "positions" (in which positions are joint angle settings) executed over time. We also see a new skill, `Do-Motion`, which performs generic motions, of which the `Get-Up` routine is an example. Note that in this skill diagram we see the display of a method interface for `Do-Motion` (the methods `Start()` and `Continue()`), and the representation of a "while loop" (`Do-Motion.Continue()` is repeatedly called until the motion is completed). Figure 7.8 displays the decision-making logic as pseudocode for `Get-Up`.

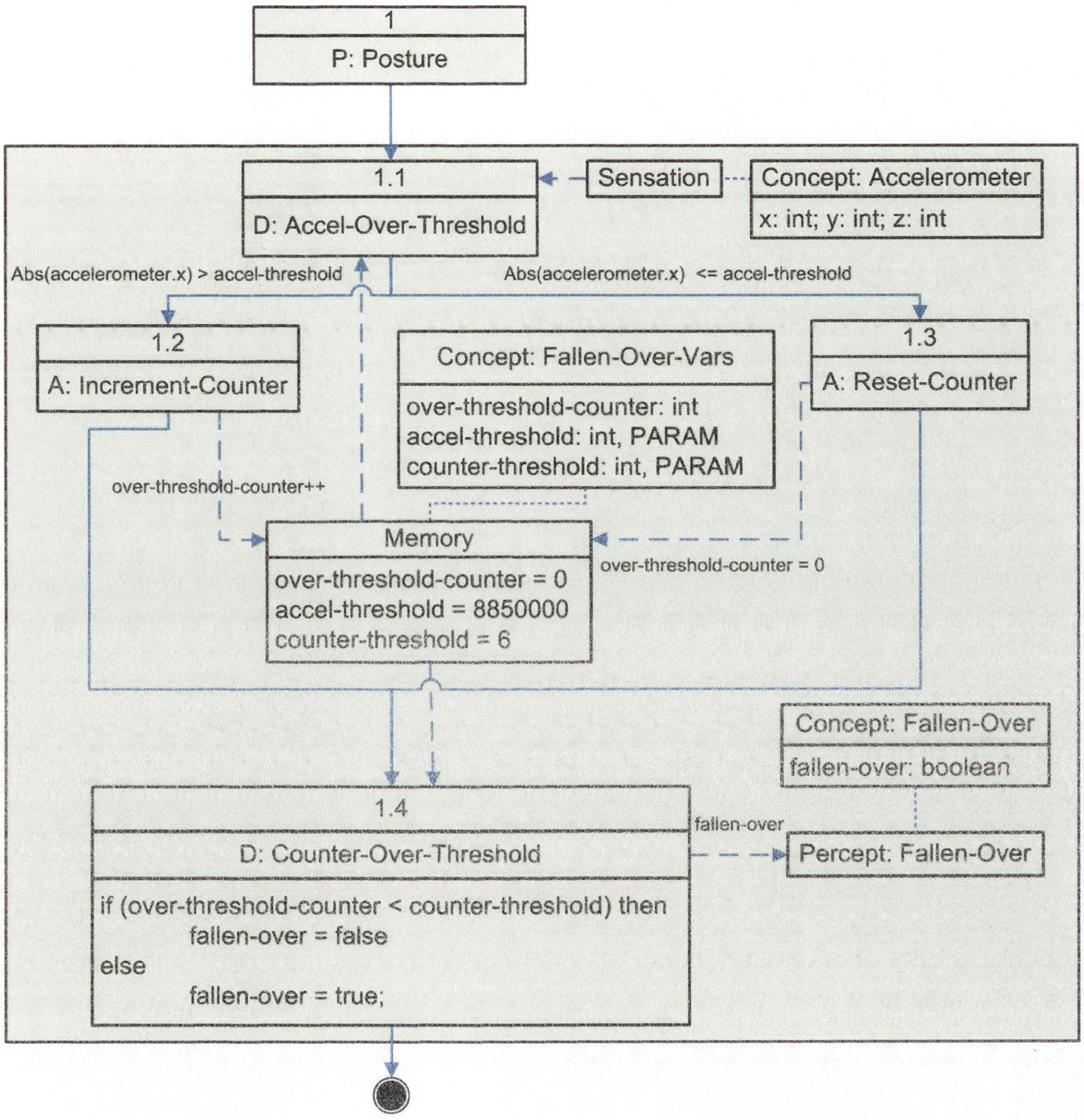
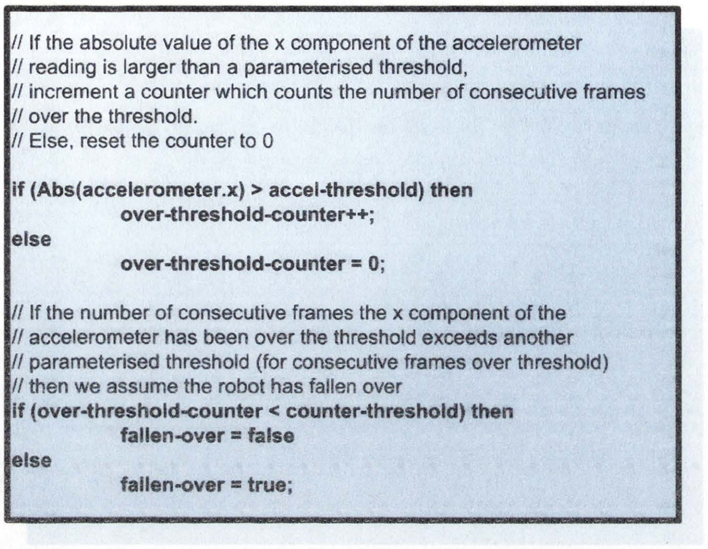

1
P: Posture
1.1
D: Accel-Over-Threshold
Sensation
Concept: Accelerometer
x: int; y: int; z: int
Abs(accelerometer.x) > accel-threshold
Abs(accelerometer.x) <= accel-threshold
1.2
A: Increment-Counter
Concept: Fallen-Over-Vars
over-threshold-counter: int
accel-threshold: int, PARAM
counter-threshold: int, PARAM
1.3
A: Reset-Counter
over-threshold-counter++
Memory
over-threshold-counter = 0
accel-threshold = 8850000
counter-threshold = 6
over-threshold-counter = 0
Concept: Fallen-Over
fallen-over: boolean
fallen-over
Percept: Fallen-Over
1.4
D: Counter-Over-Threshold
if (over-threshold-counter < counter-threshold) then
fallen-over = false
else
fallen-over = true;

Figure 7.5: Detailed-Design for `Perceive-Posture`.



```
// If the absolute value of the x component of the accelerometer
// reading is larger than a parameterised threshold,
// increment a counter which counts the number of consecutive frames
// over the threshold.
// Else, reset the counter to 0

if (Abs(accelerometer.x) > accel-threshold) then
        over-threshold-counter++;
else
        over-threshold-counter = 0;

// If the number of consecutive frames the x component of the
// accelerometer has been over the threshold exceeds another
// parameterised threshold (for consecutive frames over threshold)
// then we assume the robot has fallen over
if (over-threshold-counter < counter-threshold) then
        fallen-over = false
else
        fallen-over = true;
```

Figure 7.6: Pseudocode representing the decision-making for `Perceive-Posture`.

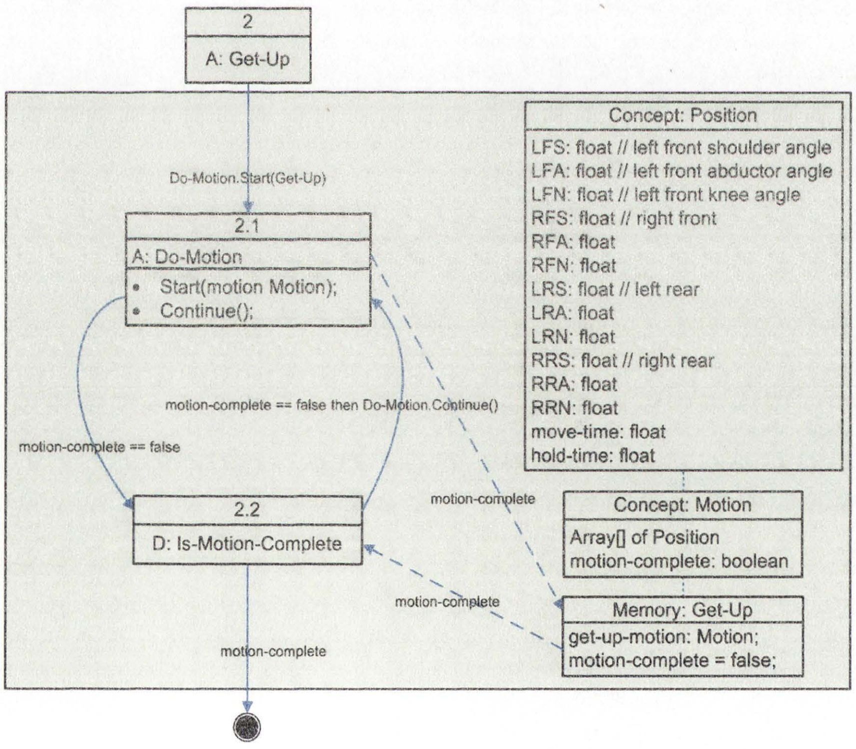
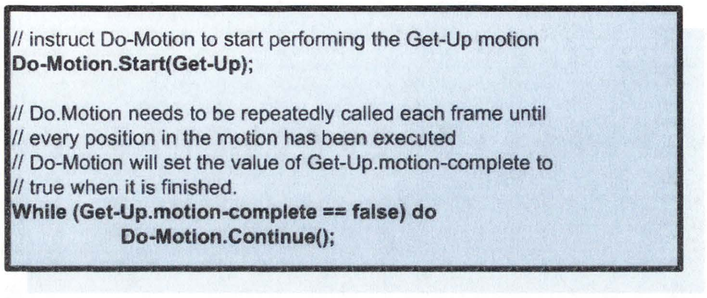


Figure 7.7: Detailed skill diagram for `Get-Up`.



```
// instruct Do-Motion to start performing the Get-Up motion
Do-Motion.Start(Get-Up);

// Do.Motion needs to be repeatedly called each frame until
// every position in the motion has been executed
// Do-Motion will set the value of Get-Up.motion-complete to
// true when it is finished.
While (Get-Up.motion-complete == false) do
        Do-Motion.Continue();
```

Figure 7.8: Pseudocode representing the decision-making for `Get-Up`.

7.5.3 Individual Skill Design

In the previous section, the aim of designing transition-conditions guided the decomposition process. However, as the several detailed examples illustrated, designing detailed transition-conditions requires design of the skill itself. In this section we consider guidelines to assist with the design of individual skills.

7.5.3.1 Skill Design: Decisions

Decision-making is the most elementary and essential component of intelligence. When designing decisions, ensure:

- **Is the decision a memory management decision or a control management decision?** In other words, what are the choices the decision can choose from? For a decision to be a decision, it must have choices. Do those choices concern memory management (writing to memory) or control management (choosing between skill transitions)?

- **For control management decisions, define transition-conditions for choosing between skill transitions.** If the decision results in multiple exit-transitions, check the mutual exclusivity of transition-conditions - i.e. can multiple exit-transitions from the same skill be true at the same time? If so, is this deliberate?

- **For memory management decisions, define input information flows, and the decision-making process for choosing the value(s) of the outgoing knowledge-flows.** Memory management decisions rely on input information to make decisions, they then process that information, and then write new information to memory - identify all three processes.

- **Decompose, if possible, the decision.** If you, as the designer, can't explain how a skill works, it needs to be decomposed, exploded and designed. Remember, decisions can be composed of other decisions. Thus, if unable to represent algorithmically how the decisions works, identify new decisions which reduce the complexity of the problem.

7.5.3.2 Skill Design: Perceptions

Perceptions are decisions about the state of the world, and as such, the design of robust and accurate perceptive capabilities is imperative for the development of grounded robotic systems. The objective of a perception is to detect a specific real-world event or entity, with the *referent* of a perception being the "thing" being perceived. The designer should identify precisely the real-world phenomena a perception refers to, and this is especially important in team development environments, as all developers should have a shared understanding of the purpose of a perception.

Consider our goal-kicking soccer robot, and the transition-conditions `ball-lost` and `ball-found`. If the robot's vision system fails to detect the ball in a single vision frame (i.e. the ball is not detected for 0.04 of a second in a vision system running at 25 frames per second), but the ball was detected in the previous vision frame, is the ball location unknown? Is either `ball-lost` or `ball-found` true in this circumstance? Understanding *how* perceptions work is imperative to designing robust behaviours. If the designer's concept of what a perception perceives is different from what the conditions of the perception actually test, this can lead to the designing of ungrounded behaviours. Consider, we could have a "loose", highly uncertain definition of `ball-found` - i.e. any glimpse of the colour of the ball in the robot's visual stream is enough to set `ball-found` to true. Alternatively, we could have a "strict", highly certain version of `ball-found`, in which `ball-found` is only true when there is a large, round (and thus un-occluded) shape in the center of the robot's visual stream. These two different interpretations of the meaning of `ball-found` will have very different implications for how a designer would attempt to implement a behaviour such as `Get-Ball`. Thus, for each perception, *we need to define the specific event it refers to* in terms of logical conditions.

For each perception, the developer should:

- **Identify the referent of the perception.** In supporting documentation, identify the referent of the perception.

- **Identify what, exactly, needs to be perceived?** Every perception has a percept, and each percept has a concept. Thus, define the concept or concepts for the perception, therefore specifying the data structures for storing knowledge about the referent. Diagrammatically, every perception should have an outgoing knowledge-flow to a percept.

- **Identify *when* the perception should be operating.** Many perceptions will need to operate all the time, regardless of their dependent decision-making processes, and thus such perceptions should be scheduled at the context (root) level of the skill decomposition. For

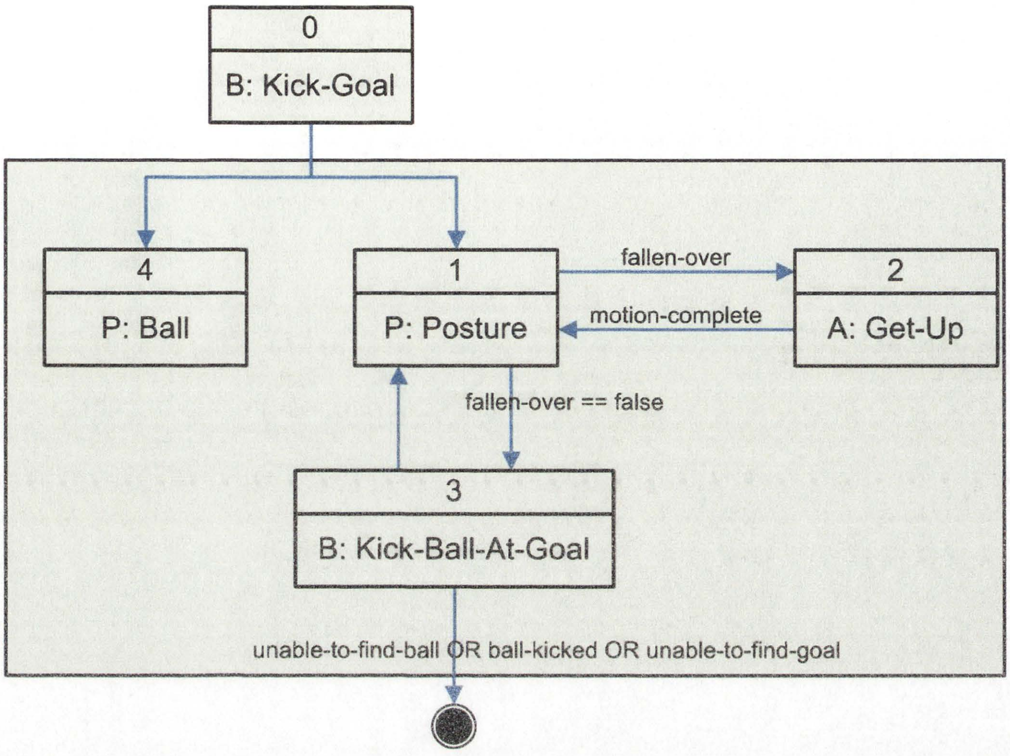


Figure 7.9: Revised `Kick-Goal` to incorporate `Perceive-Ball` operating every processing cycle.

example, a vision-based perception such as `Perceive-Ball` should operate every frame. Figure 7.9 displays a revised Kick-Goal skill diagram in which Perceive-Ball is scheduled to occur every processing cycle, while Figure 7.10 contains a decomposition of Perceive-Ball.

- **Identify the input information to the perception.** Every perception requires input knowledge, either from a sensation, or from other percepts. Thus, on a skill diagram, every perception should have at least one incoming information-flow from either a percept or a sensation.

- **Identify the required *groundedness* of the perception**. What are the dimensions of error? What is an acceptable margin of error? For each perception, we also need to identify measures of performance. What makes one implementation better than another? e.g. `ball-found` may utilise a number of groundedness criteria, which may be given different levels of importance - e.g. number of false-positives, number of false negatives, distance at which the ball may be found, closeness to the ball when it is found, accuracy of the relative distance and relative heading measurements, and so forth.

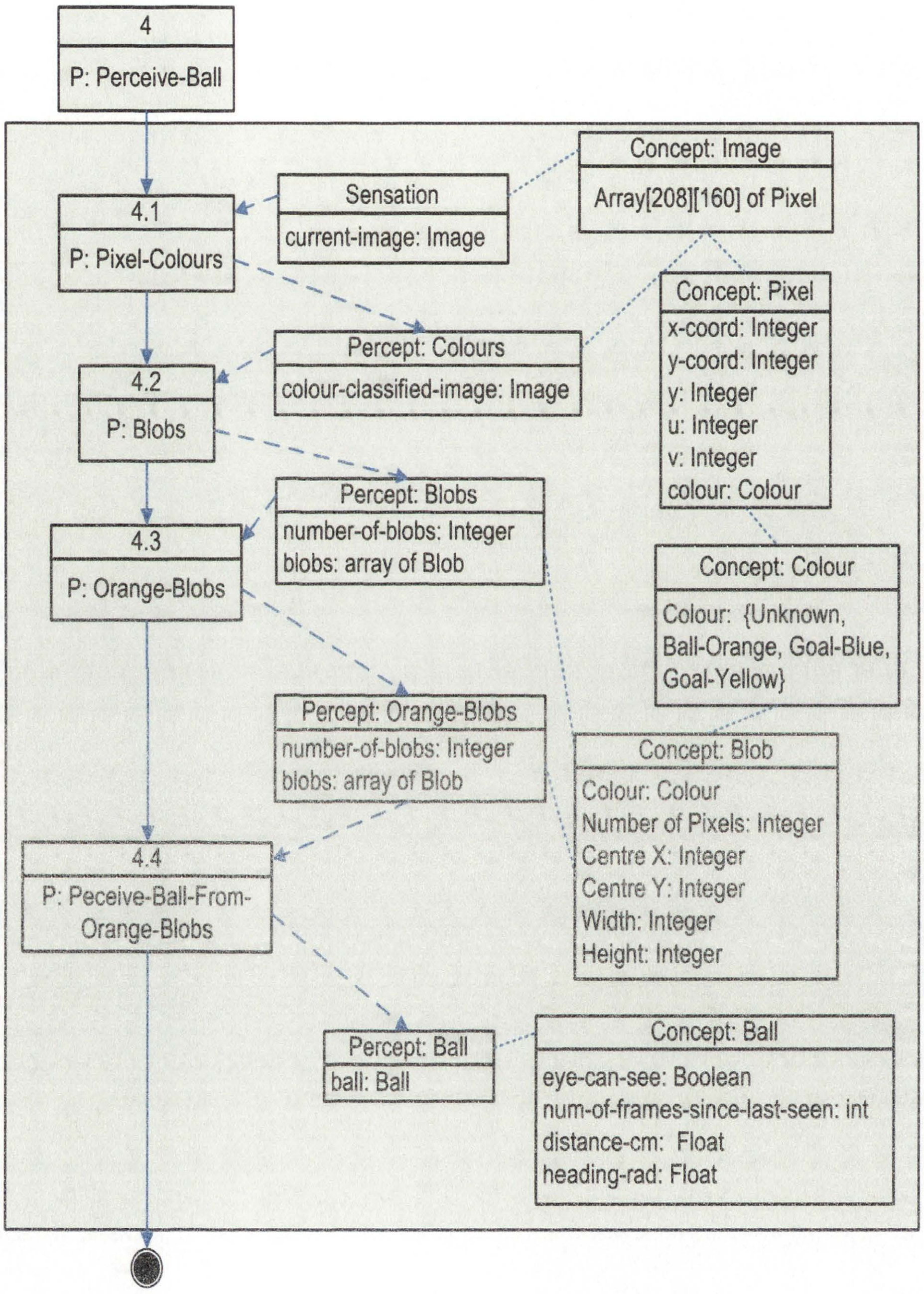


Figure 7.10: Skill diagram for `Perceive-Ball`.

7.5.3.3 Skill Design: Actions and Behaviours

The design of actions and behaviours is considered concurrently in Go-Design as actions form the building-blocks of behaviours. That is, while some behaviours can be decomposed into other behaviours, at least some behaviours in the skill decomposition must be composed of actions. In other words, all skill designs must map to actions the robot is capable of performing. The main difference between actions and behaviours is that actions are perception-less, while behaviours rely upon perception to guide them. In contrast to the design process employed in the rest of Go-Design, the design and development of actions can occur in a bottom-up manner. The term *primitive-action* refers to actions that can not be decomposed, e.g. actions provided by the robot's programming interface. Other actions are then layered on top of these, e.g. `Move-Leg` forms the basis of `Walk-A-Step`. Consequently, actions should always be tested in a bottom-up manner, e.g. if `Move-Leg` doesn't function as it should, this will affect `Walk-A-Step`, which will affect `Spin-On-Spot`, and so on. In contrast to the design of actions, the design of behaviour is a process of decomposition. All behaviours need to be decomposed into actions the robot is capable of performing.

When designing ***actions***:

- **Identify any memory required by the action.** Many actions (especially those which are not instantaneous in their effect) require internal state to remember the current state of the action's operation.

- **Ensure there is no use of perception** - otherwise, if perception is needed, treat the skill as a behaviour. Actions should be deterministic. If they are not they should be a behaviour. For example, consider `Spin-On-Spot`, an action (possibly behaviour) used when searching for the ball. If there are other robots or obstacles in the environment, than an obstacle avoidance capability needs to be incorporated, as we would not want our robot to collide with other objects while spinning. Obstacle avoidance will require perception of obstacles, and thus the use of perception. However, this raises design issues - should the perception and avoidance of obstacles be incorporated into Spin-On-Spot, or should `Avoid-Obstacles` be separated from `Spin-On-Spot`[1]? Three different designs are presented in Figures 7.11, 7.12 and 7.13. Figure 7.11 displays `Spin-On-Spot` as an action with no perceptive capabilities, and thus no obstacle avoidance capabilities. Figure 7.12 displays `Spin-On-Spot` as a behaviour, with obstacle avoidance capabilities embedded within the skill. However, this skill diagram is overly complicated, and can be simplified by the separation of `Perceive-Obstacles` and `Avoid-Obstacles`, as displayed in Figure 7.13. The latter skill diagram is not only simpler, it

[1]See Section 6.5.6.

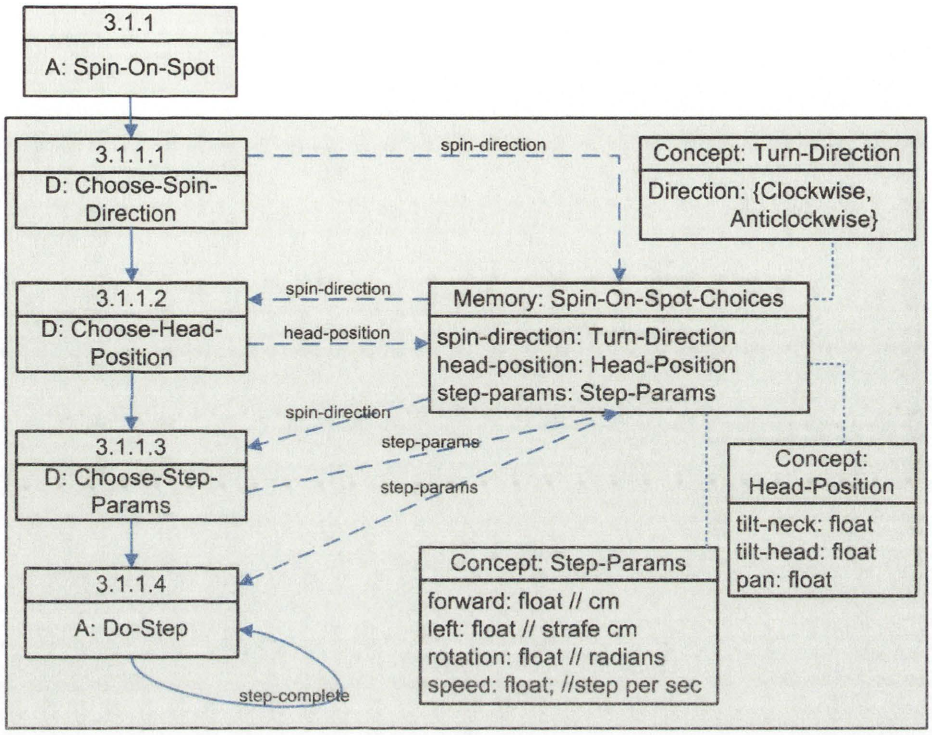


Figure 7.11: Skill diagram for `Spin-On-Spot` in which there is no obstacle avoidance capability.

also promotes reuse - `Spin-On-Spot` can be reused by other skills that may not require obstacle avoidance capabilities.

- **Decompose, if possible, the action.** If you, as the designer, can't explain how an action works, it needs to be decomposed, exploded and designed. Remember, actions can be composed of both other actions and decisions (unless they are a primitive-action). Thus, if unable to represent algorithmically how the action works, identify new skills which reduce the complexity of the problem.

In contrast to the design process of actions, the design process for behaviours follows that outlined in basic-design (Chapter 6) - iterative decomposition until decisions, actions and perceptions can be identified.

7.5.4 Reviewing a Detailed Skill Architecture

At this point in the design process, a detailed skill-based architecture for a robot control problem is emerging. The skill architecture allows the designer to model hierarchical, modular, capability-based

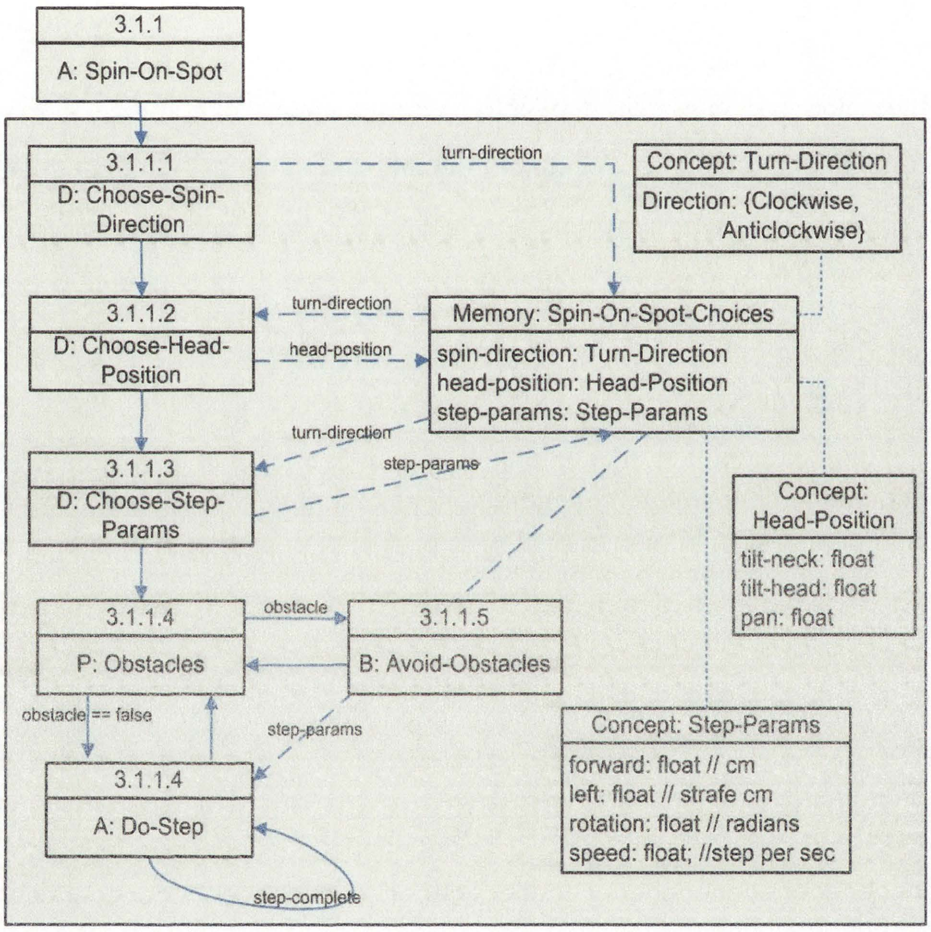


Figure 7.12: Skill diagram for `Spin-On-Spot` in which the perception of obstacles and the behaviour to avoid obstacles is incorporated.

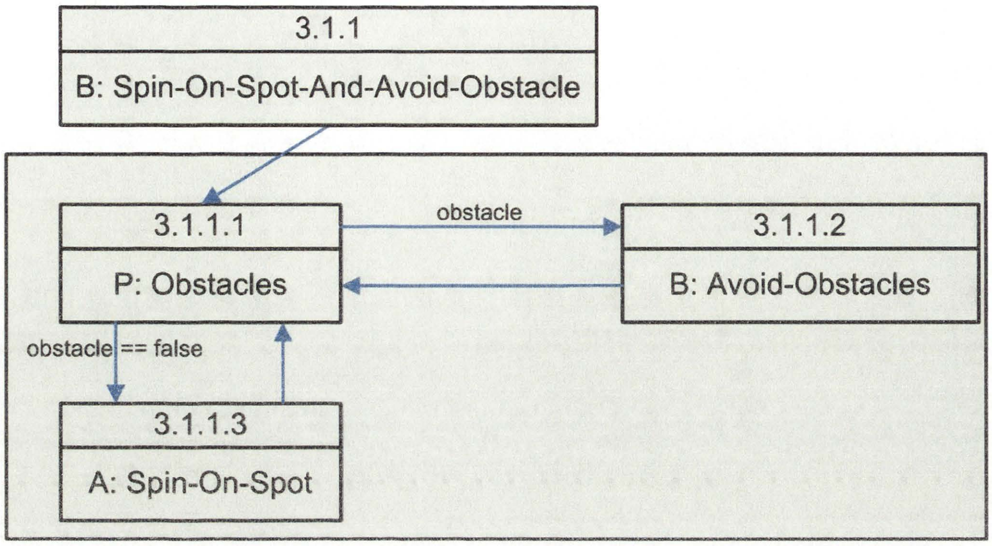


Figure 7.13: Simplified and improved skill diagram for `Spin-On-Spot` with obstacle avoidance capabilities. The Spin-On-Spot action is reused from Figure 7.11.

skills, and the interactions between these skills through event-based skill transitions and transition-conditions. The guidelines presented in this chapter will assist the designer in designing knowledge representation (concepts, percepts and memories), identification of the skills which manipulate and depend on that knowledge, and the design of actions and behaviours which control the robot's effectors. The last step in Go-Design is to review, and revise accordingly, the detailed-design, so as to ensure that the design is as thorough, consistent, and grounded as possible. In this section a skill design review checklist is presented.

7.5.4.1 Design Review Checklist

The designer should:

1. **Check all behaviours have been decomposed.** Behaviours, by definition, can be decomposed into other skills.

2. **Check skills that should run every frame (i.e. processing cycle) do run every frame**, i.e. they should never sit in any sequence after any possible skill terminators! Skill scheduling (e.g. every frame, on demand, and so forth) along with other assumptions about the skill's usage should be marked in the skill's template.

3. **For every decision, check there is at least one inward knowledge-flow.** Decisions require knowledge - they should not be made randomly. For example, in Figures 7.11 and 7.12,

the decision `Choose-Spin-Direction` does not have an incoming knowledge flow. Figures 7.14 and 7.15 display the corrected skill diagrams.

4. **Check that each decision results in either a control management decision or a memory management decision.** A decision must make a choice, whether it be choosing between a skill transition or choosing the value of a memory or percept.

5. **Ensure all decisions are decomposed to the point where pseudocode can be used to represent the decision making process.** For example, in Figure 7.5 the decisions `Accel-Over-Threshold` and `Counter-Over-Threshold` can not be decomposed any further.

6. **For every memory management decision, check that there is an outward knowledge-flow to the modified memory.** If a decision modifies memory, ensure there is a knowledge-flow to that memory. If the memory contains multiple attributes (i.e. data members), the knowledge-flow should be labeled with the particular variable that is modified. For example, in Figure 7.15 `Choose-Head-Position` modifies `head-position`, while `Choose-Step-Params` modifies `step-params`.

7. **Check every perception has a percept.** The result of a perception is stored in a percept, and should be indicated by a knowledge-flow.

8. **Check every percept has a concept.** Each percept should have a corresponding concept.

9. **Ensure every percept has a perception.** For example, the Figure 7.14 contains the percept Ball, but no corresponding perception has yet been defined.

10. **Check every perception has an input.** Every perception must have at least a sensation or a percept as an incoming information-flow.

11. **Avoid building perceptions with multiple exit-conditions.** Rather, for control management, construct a decision which uses the percept to arbitrate between skill-transitions.

12. **Check every memory has a corresponding concept.**

13. **Check memories are initialised with default values where appropriate.**

14. **Ensure each subskill's exit-transitions match those of the parent diagram.** When decomposing a skill the exit-transitions for that skill match those of the parent diagram.

15. **Check each skill's identifier and naming is unique and consistent.** During the design process the identifiers can lose sequence. Ensuring consistency with respect to each skill's identifer and naming avoids any confusion, especially in team development environments.

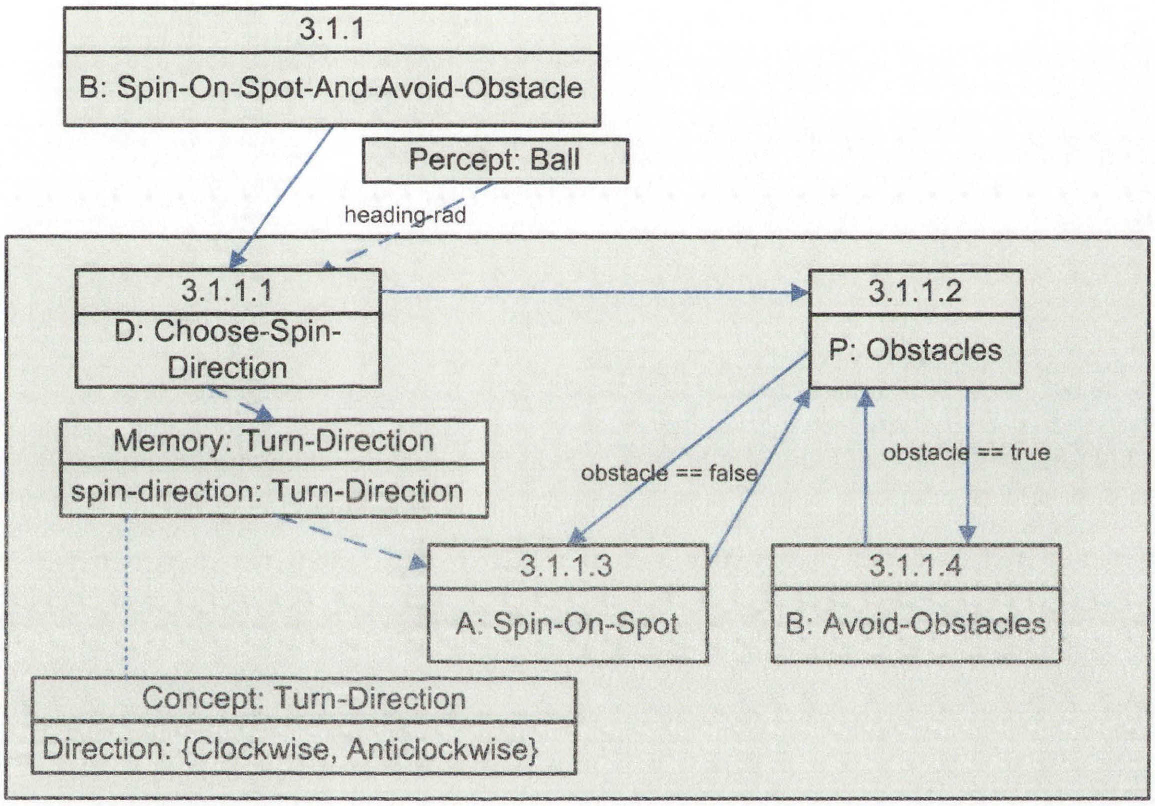


Figure 7.14: Corrected skill diagram for `Spin-On-Spot-And-Avoid-Obstacle`, which now chooses the spin direction for `Spin-On-Spot`. Figure 7.15 shows the corrected `Spin-On-Spot`. Note, the `Ball` percept is outside the decomposition box to represent the global nature of this percept.

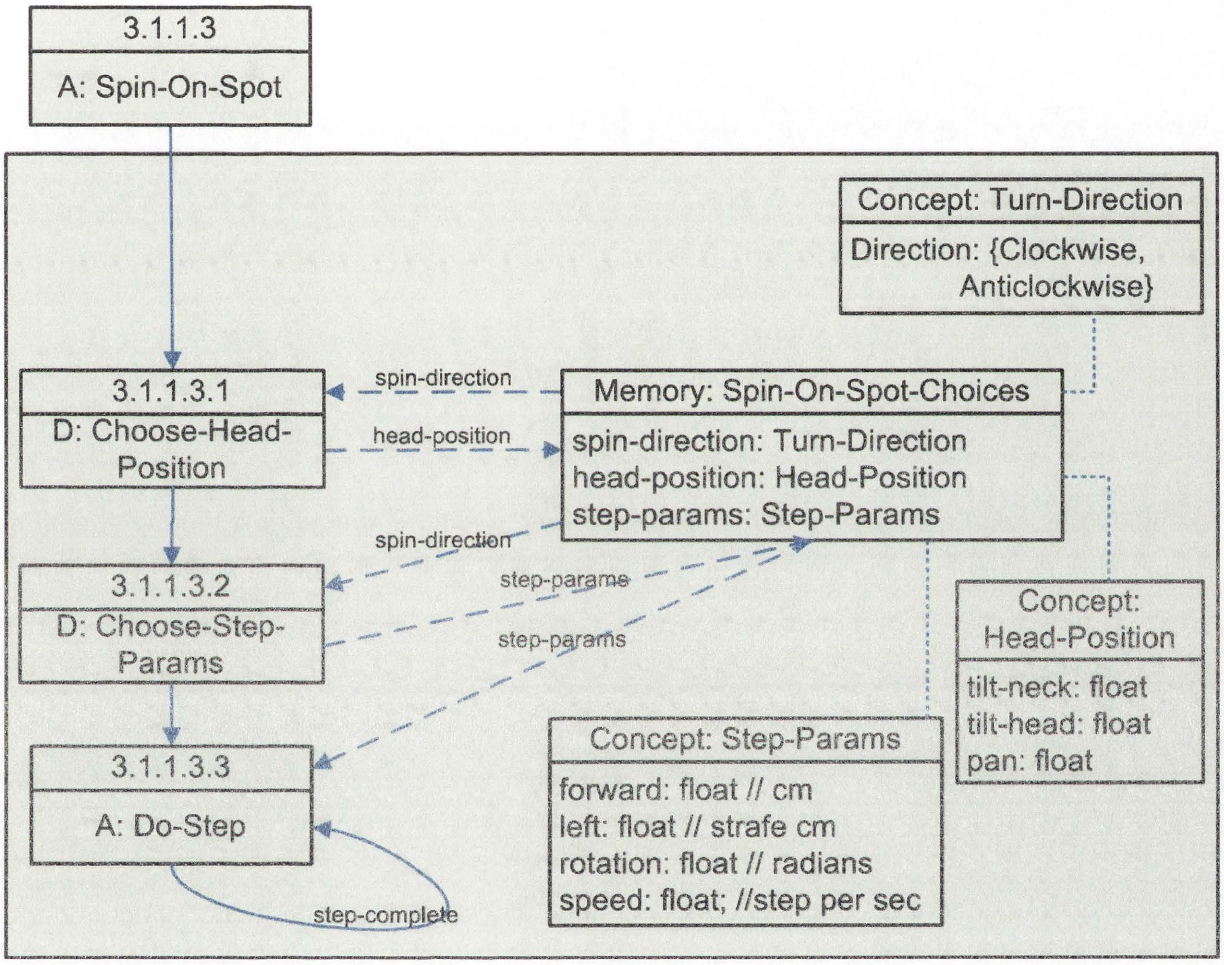


Figure 7.15: Corrected `Spin-On-Spot`, which no longer chooses spin-direction. Rather, this memory is now set by `Spin-On-Spot-And-Avoid-Obstacle`, as displayed in Figure 7.14.

7.6 Summary

In this chapter, a methodology for extending a skill architecture to a detailed-design has been presented. A detailed-design adds value to a skill architecture by guiding the developer through the process of iterative decomposition, until a design is sufficiently detailed that it provides a blueprint for software implementation. In particular, in this Chapter, Go-Design provided guidelines for:

- Identifying skill types - in particular decisions, perceptions, actions and behaviours. Skill types serve to guide the developer through the detailed-design process. For example, identification of behaviour skill-types indicates that that particular skill should be decomposed. Ultimately, all decision-making within a robot control program can be represented by decisions and perceptions, while all effector control can be represented by actions and behaviours.

- Designing transition-conditions. Each transition-condition of a skill architecture requires decision-making processes to determine *when* those conditions are true. This forces the design to have a thorough understanding of the important, relevant events which effect system performance.

- Specific guidelines for designing decisions, perceptions, actions and behaviours.

- Specific guidelines for designing knowledge representation schemas, as well as explicit identification of both the skills which modify knowledge, and the skills which rely upon that knowledge.

- A review checklist for ensuring design integrity, completeness and consistency.

Next, we turn our attention to reviewing Go-Design, comparing it to existing methodologies for robotics, and considering future work.

Chapter 8

Discussion and Conclusion

In this chapter:

- A review of Go-Design is presented, in which the benefits and limitations of the methodology are discussed, and Go-Design is compared to other related methodologies.
- A discussion and exploration of the issues faced in developing robotic agents capable of autonomous grounding is discussed.
- Lastly, a summary of the thesis and its findings is presented.

8.1 Go-Design Evaluation

Go-Design is a methodology for designing control programs for autonomous mobile robots. The methodology, while concerned with the holistic problem of robot control, has a particular concern with the grounding problem. In this section we consider the benefits and limitations of the Go-Design methodology, together with a comparative assessment with other existing, related methodologies.

8.1.1 Benefits of Go-Design

Go-Design has numerous benefits. In particular:

- **Context-level analysis**. Context-level analysis provides a step-by-step guide to eliciting and documenting requirements for robot control problems, which is especially useful for less experienced designers.
- **A simple, expressive modeling notation**. Go-Design's modeling notation allows for, in one diagram, modeling of flow-of-control through event-based skill-transitions, hierarchical decomposition (skills and subskills), identification of skill types, and designing and modeling of

representation (concepts are equivalent to data structures, while percepts and memories represent instances of those data structures). Go-Design's modeling notation is not only suitable for modeling robot control problems, but also software in general.

- **Skill types help guide the decomposition process**. During detailed-design, the identification of skill types allows the designer to be aware of when further decomposition of a skill into subskills is required, thus allowing the designer to judge when a design is "detailed enough". For example, behaviours can always be decomposed, perceptions require decisions, and decisions require implementations. Lastly, behaviours, decisions and perceptions must map to actions the robot is capable of performing.

- **Perception modeling.** The role of perception is not only integral to the grounding problem, but it is also one of the key distinguishing aspects between writing traditional software and software for controlling autonomous mobile robots. Skill types in Go-Design force the developer to label and design perceptual skills and their representational requirements (percepts). Moreover, other dependent skills, such as other perceptions, behaviours and decision-making processes, can be traced through the use of flows.

- **Flow-of-control modeling.** By modeling flow-of-control throughout the system through the use of skill-transitions, the designer can easily understand the dependencies and effects of decision-making processes and perceptual knowledge on system performance. Skill-transitions represent decision-making processes about real-world events that lead to control being passed from one skill to another.

- **Decision modeling.** Decision-making in software is often embedded and implicit (i.e. buried in "if-statements"). Go-Design identifies "decisions" as a type of skill to force explicit modeling of decision-making. Explicit modeling of decision-making forces the designer to model the possible outcomes (*choices*) of each decision, and thus the designer must consider what the desirable and undesirable (or "right" and "wrong") decision outcomes are.

- **Design of knowledge representation.** As Go-Design requires the designer to identify skill types, and each skill type has specific knowledge requirements, Go-Design forces the designer to explicitly identify the knowledge required and modified by the system in terms of conceptual data structures. Moreover, the scope of that knowledge use can be easily identified through the use of flows (i.e. identification of the skills which modify and require specific concepts, percepts and memories).

- **Transparency of design**. Skill diagrams allow for a clear, transparent representation of a system design. They document each skill's responsibilities and interactions with other skills, as well as detailed identification of all related knowledge representation. The transparency of design allows designers to explain (and understand) a robot's behaviour.

- **Assists with the management of complex designs**. The transparent and hierarchical, decomposition-based nature of skill diagrams, allows for simple, concise representation of a large, complex design. In team environments, teams can communicate precisely (each skill has a unique identifier), and the modular design allows for straightforward distribution of tasks between team members.
- **Practical.** Lastly, detailed-design is simply that - detailed. Grounding, and implementing artificial intelligence for robotics, is a software problem. To narrow the gap between conceptual design and implementation, detailed-design requires skill implementations defined in terms of pseudocode.

8.1.2 Limitations of Go-Design

The limitations of Go-Design relate to the fact that it is grounding through *design* - rather than *autonomous* grounding. As discussed in Chapter 3, grounding by design is both brittle and laborious. That is, at design-time it is difficult (except in extremely simple environments or for very simple tasks) for the designer to foresee all the possible situations the agent will encounter. It is a laborious process as the designer must ground *a priori* the agent's control program, and this process is time consuming as it generally requires a great deal of trial-and-error programming. For autonomous grounding to be possible, robots will need to be capable of learning through interaction and experience of the environment, and thus revising their own skill architectures and knowledge representations. In particular, Go-Design skill diagrams are relatively *static* - whereas, for autonomous grounding, skill architectures will need to be highly adaptable. That is, at run-time:

- skill-transitions may need to change, i.e. skill sequences may need to change order;
- concept structures may need to change or be created, i.e. what the robot is capable of knowing will need to be revised during the robot's operation;
- new skills will need to be learned and incorporated into an existing skill architecture.

While these capabilities have not been directly addressed by the Go-Design methodology, they are discussed in more detail in Section 8.2.

8.1.3 Comparative Assessment

While most grounding research concerns how to develop agents which can autonomously develop their own representations (i.e. *autonomous* grounding), the fact all robotic systems are grounded through human design on a case-by-case, ad-hoc basis has been mostly overlooked. As discussed in Chapter 3, there are only a few existing methodologies concerned specifically with the design of solutions for robotic control problems (for example Brooks' subsumption architecture[20, 21, 23, 24], Bryson's Behaviour-Oriented Design[25], and Real-time Control Systems Architecture[3, 4]), and even less that are concerned with the grounding problem (e.g. Roy's grounding framework[108]). In this section we review and evaluate Go-Design in comparison to these existing methodologies.

8.1.3.1 Subsumption Architecture

Brooks' subsumption architecture[20, 21, 23, 24] is a longstanding, behaviour-based approach to developing reactive robots. As discussed in Section 3.3.3.1, the subsumption architecture involves organising reactive behaviours into layers, where each layer implements a particular competency, with lower-level layers more "reflex-like", and higher-level layers more abstract or longer term in their planning. Each behaviour is individually hand-built, tested and *physically grounded* (see Section 2.1.4) through a tight coupling of perception and action. Agents built using the subsumption architecture lack a central, monolithic model of the world, with representation kept to a minimum and localised to each behaviour.

There are similarities between the subsumption architecture and Go-Design. For example, both feature modular design, i.e. the subsumption architecture focuses on behaviours and layers, while with Go-Design the units of abstraction are skills, with the modules of both design methods being capable of being developed, implemented and tested in isolation. As with the subsumption architecture, representation can be distributed in Go-Design (though it need not be - it is an implementation issue) - however, the *design* of representation in Go-Design is distributed, i.e. representation is designed to support localised decision-making.

The main differences between the subsumption architecture and Go-Design is Go-Design is less restrictive with regard to architecture, representation, and the use of planning, while more detailed with respect to decomposing and implementing individual modules. For example, Go-Design is less restrictive in that behaviours in Go-Design need not be reactive, while the principles of subsumption can be modeled using Go-Design (e.g. the ability to detect and respond to falling overrides all other skills in the `Kick-Goal` example, as seen in Chapter 7). Go-Design is more detailed with regard to individual behaviour design because detailed-design provides a set of specific guidelines for designing the implementations of individual behaviours. Also, Go-Design differs from the subsumption architecture in that rather than coordinate complex action-selection problems through inhibition and suppression of layers, Go-Design employs skill-transitions and transition-conditions. Bryson[25] argues it is a conceptually simpler task for an engineer to describe a behaviour in terms of a sequence of events, as this is a characteristic of our own planning processes.

8.1.3.2 BOD

Behavior-Oriented Design[25], or "BOD", is a methodology for engineering behaviour-based agents (embodied or software-based) with multiple, potentially conflicting, goals or tasks (see Secion 3.3.3.4). BOD is a behaviour-based approach to intelligence, and is influenced by the Brooks' subsumption architecture[20, 21, 23, 24]. BOD differs from the subsumption architecture in that action-selection (i.e. behaviour selection) relies upon "reactive planning", and provides (both in design and code) a specific data structure called POSH (Parallel-rooted, Ordered, Slip-stack Hierarchical Reactive Plans) to perform this. With POSH, reactive plans can be thought of as a hierarchical, prioritised sequence of behaviours that should be executed in specific circumstances (contexts), with the designer's task being to specify when and how each behaviour is expressed.

As with the subsumption architecture, BOD and Go-Design are similar in that they are both modular, with the unit of abstraction in BOD being the behaviour and the unit of abstraction in Go-Design being the skill. Both BOD and Go-Design use a top-down iterative decomposition design process, with Go-Design's decomposition process influenced by BOD's simple, straightforward approach of identifying "whats","whens", and "hows". Go-Design, however, differs from BOD in that Go-Design is not restricted to the use of reactive planning for action-selection. With BOD, there is also little problem analysis, no requirements elicitation, no identification of behaviour types (in comparison to skill types), no clear guidelines to identify when to stop decomposing, and no consideration for evaluating the groundedness of an agent.

8.1.3.3 Wasson's Representation Design Methodology

Wasson[136] presents a methodology for designing representations for autonomous mobile robots (see Section 3.3.3.2). The glaring difference between Wasson's representation design methodology and Go-Design is scope - Wasson's representation design methodology focuses purely on representation design, not issues of control, decision-making, and grounding, as does Go-Design. However, Wasson's view of the representation problem is similar to that discussed in Chapter 3, in that Wasson considers there to be two main problems in designing representation systems. Firstly, choosing *what* to represent, stating the "most fundamental question in the design of a representation system is, what should be represented?" (p. 252), and secondly, the *maintenance* of representations with respect to dynamic environments. These two problems are similar to the problems of relevance and reference - subproblems of the grounding problem. Another similarity includes the use of task decomposition to identify subtasks as a means for identifying required representation. However, Wasson's representation diagrams require two separate diagrams to illustrate task decomposition and flow-of-control, whereas both concepts can be illustrated through the use of skill diagrams.

8.1.3.4 RCS

RCS[3, 4] (Real-time Control Systems Architecture) is a methodology and architecture suitable for software-intensive, real-time control problems, such as those posed by autonomous mobile robots (see Section 3.3.3.3). RCS uses a methodology to iteratively partition system tasks into control nodes, with each control node sharing a generic node model. Each control node contains a process for behaviour generation, world modeling, sensory processing, and value judgment, together with a knowledge database. The placement of a control node in an RCS architecture hierarchy indicates the the scope and time span of the node, with higher level nodes broader in their planning scope.

RCS has some similarities to Go-Design. It is modular in nature, with control nodes being the main unit of design. Control nodes also have structural similarities to skills - they have different sub-processing types, such as behaviour generation, world modeling, sensory processing, and value judgment. Behaviour generation can be likened to behaviour skill types, world modeling and sensory processing likened to the perception skill type, and value judgements can be likened to the decision skill type. The design process in both methodologies involves iterative, top-down decomposition of tasks into subtasks, and both methodologies have a requirements analysis phase. With RCS each control node is designed as a finite state machine, and the design process involves identifying the state transitions and their dependencies upon the world - this is similar to defining skill-transitions and their transition-conditions. RCS is also the only general purpose robotics methodology (besides Go-Design), which treats the symbol grounding problem as a problem *per se*.

RCS differs from Go-Design in that its hierarchical structure is determined by planning scope, whereas the Go-Design planning hierarchy is determined by conceptual groupings - i.e. how the designer decomposes a problem. Also, each control node is more structured in its architecture than a skill, as each node must possess processes for behaviour generation, world modeling, sensory processing, and value judgment. The RCS architecture results in highly complex hierarchical designs with no simple notation for representing them (as evidenced by Figure 3.5). Lastly, as of 2005, there "remain many features of the 4D/RCS reference model architecture that have not yet been fully implemented in any application"[4], therefore leaving some doubt as to the practicality of the RCS architecture.

8.1.3.5 Roy's Grounding Framework

Roy[108] offers a grounding framework for grounding language in the world for robotic agents (see Section 3.3.3.5). The fundamental difference between Roy's grounding framework and Go-Design is scope - while Roy's grounding framework focuses on the grounding problem, Roy's interpretation of the grounding problem restricts it so that it is only concerned with language, with grounding defined as the ability to "use words to refer to entities in the world".

8.2 Future Work

Recall that in Chapter 1, one of the thesis objectives was to explore how the grounding methodology and framework can be used for designing and comparing long-term solutions to the autonomous grounding problem (the problem of how to build artificial systems which can ground themselves). In this section we consider issues related to autonomous grounding, identified during the development of Go-Design.

8.2.1 Understanding Decomposition

Decomposition is a weak method which involves breaking a complex problem into smaller subproblems - the rationale being that solving the smaller subproblems is a simpler, more tractable and feasible process than solving the problem as a whole. Generally, robot control problems lack an explicit, defined structure (as opposed to, for example, programming a chess computer). In such situations, decomposition is used by designers to cope with the complexity of the problem[114], and decomposition is common to all methodologies for designing programs for controlling autonomous mobile robots. Likewise, problem decomposition is integral to many aspects of software design - for example, in object-oriented programming a problem task is decomposed into objects with roles and responsibilities, in behaviour-based robotics the problem task is decomposed into behaviours, while in Go-Design the problem task is decomposed into skills. Despite the remarkable presence of decomposition in design, it is a fairly neglected topic in the psychology and science of design[74, 88]. With respect to grounding, knowing what requires representing (the problem of relevance) requires knowing the subtleties of what decisions need to be made - both to achieve the task, but also to maintain the correspondence between representation and referent. Therefore, understanding the process of decomposition is imperative for designing autonomous grounding capabilities.

8.2.2 Go-Design Development Environment

One of the limitations of Go-Design is the static nature of its designs. For autonomous grounding to become a reality, programs need to be capable of change and adaption in real-time. Thus, in terms of Go-Design, the structure of a skill architecture needs to be capable of self-modification to adapt to a changing world. However, traditional software, including Go-Design, does not change except with discrete revisions. Program revision and maintenance is resource (e.g. time) consuming, and importantly, when the control program of a robotic agent is changed, the changes are made by the programmer, and not by the program itself. An adaptable system will need to be capable of changing its programming instructions for future conditions based upon experience (i.e. through learning) in real-time. Examples of such approaches, as discussed in Section 2.3.6, fall under the umbrella of developmental robotics[13, 35, 140, 138, 139]. Developmental robotics aims to create artificial intelligences which exhibit autonomous mental development, in the same way human cognitive and behavioural development occurs through infancy to adulthood. The practical motivations behind developmental robotics is to develop robots capable of learning new tasks that a human programmer does not anticipate at the time of programming.

To enable Go-Design to model both dynamic control structures and to improve productivity, future plans involve the development of a Go-Design development environment, in which skill diagrams can be used not only as a design tool, but also as an implementation tool. The highly detailed nature of Go-Design skill diagrams offers an extensible framework for designing transparent control systems for autonomous robots, and this transparency is vital for understanding how developmental control programs have evolved through interaction and experience of the environment. Benefits of a Go-Design development environment would include real-time debugging and modification of skills, skill-transitions, transition-conditions, and concepts. Lastly, as discussed in previous work[116], a Go-Design development environment could provide seamless integration of off-board resources, such as those provided by the semantic web, which offers a medium for knowledge sharing and reuse between agents.

8.2.3 Evaluating Grounding Approaches

In other related fields, the need for methods to facilitate comparative assessment and evaluation of algorithms has long been recognised. For example, in the field of computer vision empirical assessment of algorithms often involves the comparison of processed images with images which represent the "ground truth" (i.e. "correctly" processed images, as determined or perceived by a human observer) through the use of various metrics[16]. Despite the obvious benefits of such empirical evaluation techniques (i.e. the comparison and accurate assessment of the state-of-the-art), and the recognised fact that grounding performance can be a matter of degree[48, 143], empirical evaluation techniques are lacking with respect to grounding.

8.2.4 Escaping the Chinese Room

Long-term grounding-related research is, generally speaking, motivated by two key issues:

1. The need to develop programs which can be meaning users, rather than only being meaningful to the program writers and users. This problem has be called first-hand semantics[145], or intrinsic meaning[67].

2. The need to develop programs which can ground themselves autonomously, i.e. programs that can autonomously find relevant and useful structure in the world, represent it, and maintain that representation with respect to a changing world.

How, in the longer-term, can these issues be overcome? In this section we consider two key issues which may contribute to a grounding solution: improved understanding and modeling of the concept of "meaning", and the incorporation of a *predictive* element to grounding solutions.

8.2.4.1 Understanding Meaning

As discussed in Chapters 2 and 3, meaning is a concept which has troubled philosophers throughout history. Two distinct aspects of meaning are appealed to in grounding literature - firstly, the idea that the grounding problem is the problem of creating intrinsic, first-hand semantics or meaning for an artificial computational system; and secondly, referential meaning - that the grounding problem can (at least partly) be solved by somehow connecting (in the right way) symbols with sensorimotor data. While "meaning" is a crucial, integral component of the grounding problem, nearly all grounding-related research is devoted to modeling it (usually implicitly) as a problem of maintaining correspondence or reference between a representation and a referent. One notable exception, as discussed in Section 2.2.4, is Roy[108, 109] who models three aspects of meaning in language - emotional, connotative meaning (e.g. "my father gave me that cup - it has great meaning for me"), functional meaning (e.g. "this coffee is cold" can imply "get a hot coffee"), and referential meaning (as in a theory of reference, e.g. "I meant that one"). As Steels[121] observes, there is an enormous gap between the richness of human meaning, and the reality of representation use in computer-based systems. This inadequacy of meaning modeling is highlighted through the use of the "concept" representation-type in Go-Design, and the examples presented in Chapter 7. For example, consider the concept `fallen-over`, and its definition in terms of rules applied to accelerometer sensor data, as displayed in Figure 7.5. It would appear illogical (even absurd) to attribute any intrinsic meaning to fallen-over, whether such rules for determining if the robot is `fallen-over` are learned autonomously or hand-coded by the designer. Likewise, consider attributing meaning to any visual-based concept, and the display of raw image data and the corresponding image displayed in a format suitable for computer monitors in Figure 5.2. How can any visual "concept", simply defined in rules applied to collections of numbers, be meaningful to a computer program, rather than meaningful solely to its designer? If a solution to the grounding problem concerns the development of a program capable of intrinsic, first-hand semantics, it is not clear how this can be achieved by defining higher-level "symbolic" representations with lower-level "subsymbolic" sensorimotor representations. Thus, if researchers continue to define the grounding problem in relation to meaning, future work must concern modeling and defining "meaning".

8.2.4.2 Grounding through Prediction

An important part of the grounding problem is the development of systems capable of grounding themselves, rather than being grounded through design, i.e the development of systems capable of autonomous grounding. Several authors[104, 108, 113] have proposed that grounding requires a predictive component. The premise behind such arguments is that perception requires hypothesising about the future state of the world, and then comparing the actual state of the world against what is sensed. Thus, a hypothesis is tested by comparing the predicted state of the world with the sensed state of the world. If a hypothesis is confirmed, the strength (or probability) of a theory about the world being correct is strengthened, whereas if a hypothesis is not confirmed as expected, beliefs are revised. Thus, coupled with interaction with the world, a robot can create a closed loop in which every processing cycle offers the agent the opportunity to conduct an experiment, testing its beliefs about the world. Such an approach to autonomous grounding has been rarely been tested, but may offer a viable solution to assist with the problem of autonomous grounding.

8.3 Summary and Concluding Comments

This thesis has presented Grounding Oriented Design (Go-Design) - a methodology for designing control programs for autonomous mobile robots. The methodology, while concerned with the holistic problem of robot control, has a particular concern with the grounding problem[67] - a longstanding, poorly understood issue that has interested philosophers, computer scientists, and cognitive scientists alike. The grounding problem, in its various guises, refers to the task of creating *meaningful* representations for artificial agents. Throughout this thesis, it has been argued that in the context of practical robotics, grounding is the process of "embedding" an agent in an environment to perform a task. Or more colloquially, an agent is grounded when it knows what it needs to know, to do what we need it to do. In such a context, a grounded agent possesses a representation which faithfully reflects relevant aspects of the world. In contrast, an ungrounded agent could be, for example, delusional or suffering from hallucinations ("false positives"), overly concerned with irrelevant things (e.g. the frame problem[93]), or incapable of reliably perceiving, recognizing or anticipating relevant things in a timely manner ("false negatives").

While most grounding research concerns how to develop agents which can autonomously develop their own representations (i.e. *autonomous* grounding), the fact that all robotic systems are grounded through human design on a case-by-case, ad-hoc basis has been overlooked. That is, currently, the majority of the grounding process is performed by program designers, and not autonomously by the programs we design. Designers find meaning, structure, and patterns in the world that can be used in the design of robot control programs, but due to our poor understanding of this process, little (if any) of our knowledge of "how to ground" is encoded in programs. Thus, today's agents are grounded by *design* - i.e. we ground robots by understanding the world for them. We identify structure and consistency in the world that can be used for decision-making. When grounding agents there are two main problems - choosing what to represent (the problem of relevance), and the problem of maintaining that representation with respect to a changing world (the problem of reference).

Due to the difficult nature of the grounding problem, designers tend to ground robotic agents on a case-by-case, task-by-task basis. As a consequence, the systems we build tend to be brittle in the face of unanticipated changes in the environment or task, and as such, robotics development is a highly iterative code-and-fix paradigm, in which developing systems capable of "scaling up" to human levels of intelligence has (so far) proven unattainable. Thus, the development of Go-Design is motivated by the need for a systematic approach for designing, implementing, and thus grounding, the "minds" of robotic agents. As discussed in Chapter 3 there are few existing methodologies concerned specifically with the design of solutions for robotic control problems, and even less that are concerned with the grounding problem.

Due to our interest in grounding, Go-Design focuses on understanding (both that of the the developer and robot) by providing a simple, expressive means for modeling knowledge and decision-making in robotic systems through the use of skill diagrams. Go-Design offers guidelines and processes for not only iteratively decomposing a robot control problem into a set of collaborating skills, but for also designing the representation to support those skills. Go-Design begins with context-level analysis, in which a set of guidelines assist the developer in understanding the nature of the robot control problem. Context-level analysis is followed by two design phases: basic-design which involves constructing a skill-architecture, and a detailed-design in which a skill-architecture is used to design the agent's representation and decision-making processes. A groundedness framework[143] is used for describing and assessing the groundedness of either the complete system or of individual skills. Examples of the methodology's use and benefits were provided, while suggestions for future work (such as the development of a dynamic Go-Design development environment, and autonomous grounding through prediction) were discussed.

Appendix A

Grounding Oriented Design: The Step-by-Step Guide

This chapter presents a summary of the key steps of Go-Design.

A.1 Context-Level

The first step of Go-Design is understanding the nature of the current problem - a process called *context-level analysis.*

A.1.1 Context-Level: Objectives

1. **Identify the *objectives* of the system.** *Why* is the robotic system being built? Understanding the purpose of the system will help with understanding the intent of the project's requirements.

2. **If there are multiple objectives, prioritise them in order of importance.** Identify trade-offs or possible conflicts between different objectives, and then place a value on their relative importance. For example, safety versus speed.

A.1.2 Context-Level: Constraints

Constraints specify *how* requirements must be achieved, and thus must be identified before design and development commences. In particular, the developer should identify:

1. **What are the project's resource constraints?** e.g. available processing power for real-time processing; development time, deadlines; money; man-power, laptops, batteries, etc.

2. **What are the project's implementation constraints?** i.e. are there any particular constraints governing how the context-level requirements should be achieved? e.g. do any particular algorithms need to used?

A.1.3 Context-Level: Current Capabilities

The fact we need to build something indicates we lack particular capabilities. We address the question of "what do we build?" by comparing the existing capabilities of the system with the required capabilities of the system. The robot's current capabilities are dependent upon both the robot's hardware, and the existing software for utilising that hardware.

A.1.3.1 The Robot(s)

Context-level analysis requires understanding the capabilities of the robot or robots utilised for the project. The developer should specify and understand:

1. **What sensors does it have? What are their physical characteristics? What aspects of the world can the sensors allow us to perceive?** For example, cameras will have characteristics such as field-of-view, distance sensors will have a limited range, and so forth.
2. **What effectors does it have? What are their physical characteristics?** Robots have physical limits, e.g. how much they can carry, how quickly they can move, etc.
3. **What are the processing capabilities of the robot? Does the processing take place on-board or off-board?** On-board processing will usually be limited, whereas off-board processing will be less restrictive. Awareness of the robot(s)' computational resources (memory and processing speed) is required for establishing the feasibility of different algorithmic solutions.
4. **What is the robot's battery life?** This may affect the robot's design and suitability for the task, i.e. the need to frequently recharge the robot's battery may not be well suited to some tasks.
5. **What communication capabilities does the robot possess?** For example, does the robot possess serial communication, removable flash drives, wireless TCP/IP, and so forth. The ability to communicate with the robot may affect how program revisions are uploaded, how the robot is programmed, tested and debugged, how communication between robots in multi-robot systems is performed, and the external resources that can be accessed (such as the internet, the semantic web, and so forth).

A.1.3.2 Software

Hardware is of little use without software to control it. For example:

1. **How is the robot programmed?** Does it have an operating system? What language can it be programmed in? Does it have an API? Does it have a development environment?

2. **What development and debugging tools are available?**

3. **What existing perceptive capabilities does the robot have? What perceptive skills can be reused from previous projects?** Sensations describe the raw data from robot's sensors, whereas perceptions are interpretations of that data as being about something, e.g. the numeric value attributed to a pixel representing a colour.

4. **For each perceptual skill, how reliable are they?** What is the nature of their error? How is their performance affected by or related to environmental influences? How can they be tested? Do visualisations need to be developed (e.g. writing software to stream the robot's vision so it can be displayed on a computer monitor).

5. **What existing effector and behaviour capabilities does the robot have?** What physical actions can the robot perform? What "routines" can be used? For example, to make a robot walk do we need to write a locomotion module or can we simply call a command "walk"?

6. **How reliable and accurate are the existing effector and behavioural capabilities?** For example, consider an inverse kinematics walking engine - does a command which instructs the robot move 1cm forward *really* move the robot *exactly* 1cm forward? If not, what are the consequences of such error and how can they be overcome?

7. **Identify the capabilities of any existing communication software.** For example, are there FTP servers? Is it a case of socket programming, using reliable TCP/IP streams, or some other higher-level communication protocol? Communication capabilities are required not only for multi-agent systems, but for programming and testing.

8. **What preexisting decision-making and planning processes does the robot have?** e.g. a reasoning engine, a skill-architecture, etc.

9. **What other resources can assist with the development process?** For example, third party tools, newsgroups, and access to domain experts.

A.1.4 Context-Level: Required Capabilities

In the previous sections we considered the project's objectives (i.e. *why* we are building the robot mind), as well as the existing capabilities of the robotic system. The next step is to consider the requirements of the robot's mind - in other words, *what* the robot's mind needs to allow the robot to do. Go-Design has two processes to assist with requirements understanding - *requirement templates* for documenting individual requirements, and a *requirements elicitation checklist* to help identify requirements.

A.1.4.1 Requirement Templates

1. **Identify the *requirements* of the system** - what it is that the robot(s) must do? For each requirement, specify:
   (a) ***What* the robot should do**;
   (b) ***When* the robot should do it**, i.e. the context of the requirement;
   (c) ***Why* the robot should do it**, i.e. *the need*;
   (d) **The nature of any *constraints* on how the requirement should be achieved**, e.g. "the robot must be able to localise, but the localisation algorithm must use a Particle Filter";
   (e) **The *importance* or *priority* of the requirement**, e.g. System Critical, Essential, Desirable, Luxury, etc.
   (f) **Performance dimensions and metric**, i.e. how attempts to satisfy the requirement can be measured (e.g. the dimension speed and the metric time); and lastly,
   (g) **The required performance standard** in terms of measurable performance dimensions.

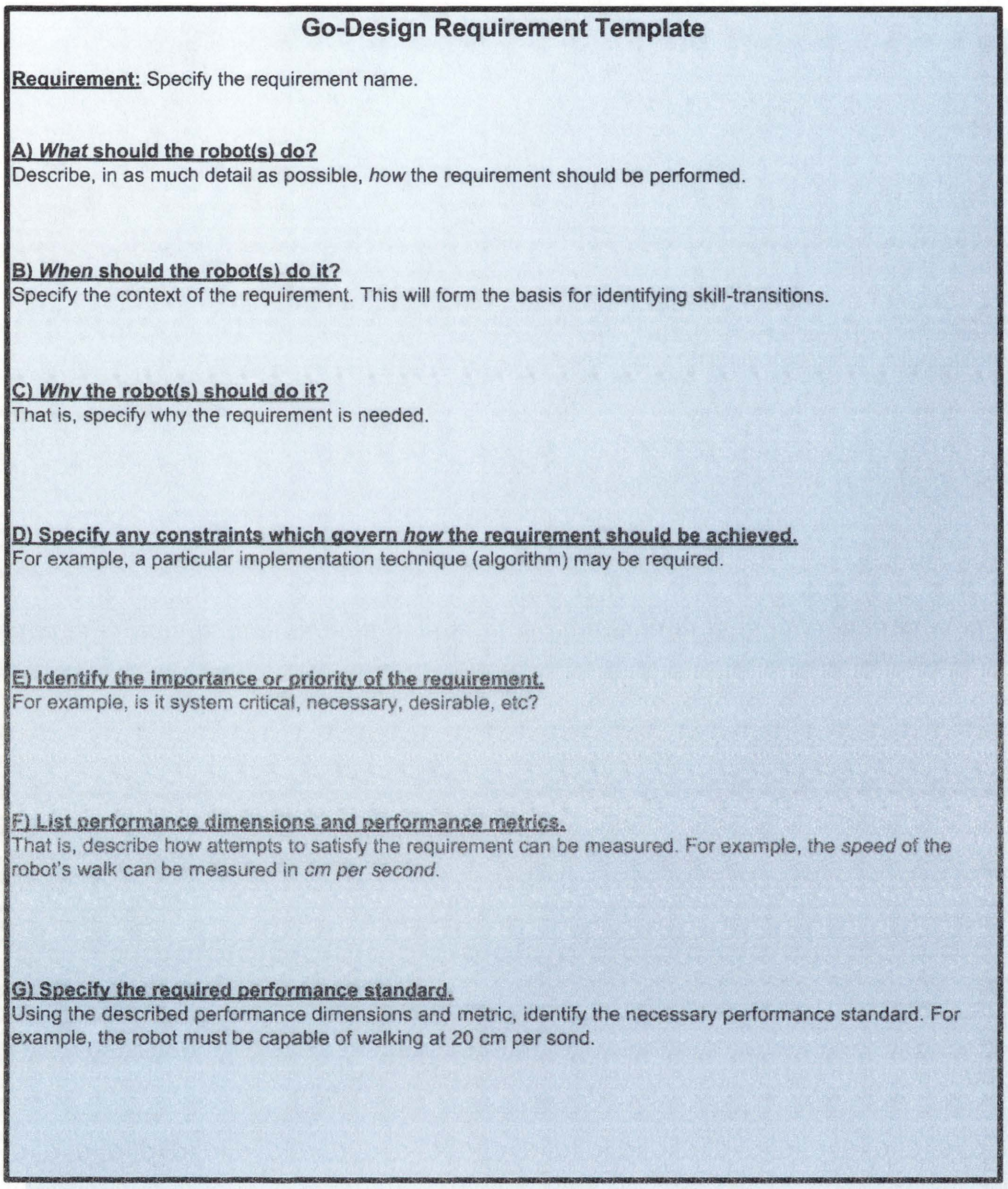
Figure A.1 contains a blank requirement template.

A.1.4.2 Requirements Elicitation Checklist

Table A.1 contains a checklist to assist with eliciting project requirements.

A.2 Basic-Design

Basic-design is the first step of the design process. Basic-design involves designing a skill-based architecture to solve the required task. A *skill architecture* consists of *skills*, *skill-transitions* and *transition-conditions*.



Go-Design Requirement Template

Requirement: Specify the requirement name.

A) *What* should the robot(s) do?
Describe, in as much detail as possible, *how* the requirement should be performed.

B) *When* should the robot(s) do it?
Specify the context of the requirement. This will form the basis for identifying skill-transitions.

C) *Why* the robot(s) should do it?
That is, specify why the requirement is needed.

D) Specify any constraints which govern *how* the requirement should be achieved.
For example, a particular implementation technique (algorithm) may be required.

E) Identify the importance or priority of the requirement.
For example, is it system critical, necessary, desirable, etc?

F) List performance dimensions and performance metrics.
That is, describe how attempts to satisfy the requirement can be measured. For example, the *speed* of the robot's walk can be measured in *cm per second*.

G) Specify the required performance standard.
Using the described performance dimensions and metric, identify the necessary performance standard. For example, the robot must be capable of walking at 20 cm per sond.

Figure A.1: Go-Design Requirement Template.

A.2.1 Skills

Skills are encapsulated abilities. Skills can represent any type of ability - complex behaviours such as playing soccer or driving a car; thought processes such as learning, performing calculus or deductive reasoning; perceptions such as being able to perceive colour, or being able to see a soccer ball or traffic hazards; and lastly, simple actions such as the ability to consciously blink an eyelid or to move a leg to a particular position.

Figure A.2 provides a legend which describes the elementary notation used in skill diagrams, while Figure A.3 illustrates a simple skill diagram for the skill "`Do-Grocery-Shopping`". Skills are represented with rectangular boxes, with the name of the skill written below a numeric identifier. The first letter of word in a skill name should be capitalised, with the words of a skill name separated by a hyphen. Subskills of a skill are identified by their numeric identifier. For example, if a skill's identifer is "1", any subskills would be identified as "1.1", "1.2" and so forth. Skill decompositions are represented not only by numeric identifiers, but by encompassing the subskills which constitute the decomposition with another, larger rectangular box (the *decomposition box*). A skill's name should be unique, and thus the same skill can appear with different numeric identifiers if it is reused by two different parent skills. Skill sequences require the use of skill-transitions, with skill-transitions represented with an arrow. Skill-transitions can have transition-conditions, which are testable conditions which signify when one skill should stop and another should start. We represent transition-conditions by adding a caption (in lowercase) to the arrows between skills on a skill decomposition diagram. Also note that a black-dot with a circle around it is used to represent the skill's completion (as per UML state diagrams). If a skill-transition lacks a transition-condition, this implies the skill-transition occurs automatically (i.e. by default). Flow-of-control begins with the parent skill - thus, in Figure 6.2 flow-of-control begins with `Do-Grocery-Shopping`. A skill diagram can have multiple flow-of-control start and end points. Lastly, concurrent skills are represented by having multiple transitions active at the same time.

A.2.2 The Basic-Design Process

The first stage of Go-Design, "basic design", provides a set of guidelines for constructing a skill architecture. Designing a skill architecture involves:

1. Decomposing the problem (the context-level skill) into subskills.

2. Identifying and defining the collaborations (the transitions) between skills.

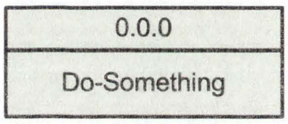
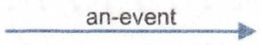
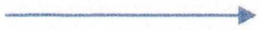
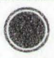
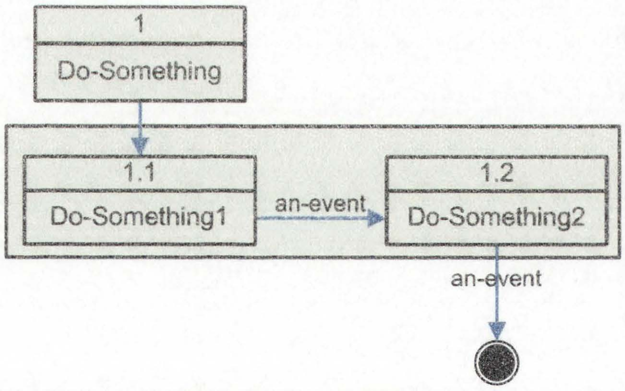
Symbol	Explanation
	A skill. The skill's name is written underneath a numeric identifier.
	A skill-transition. The transition-condition is written above the arrow.
	A skill-transition, without a transition-condition. A condition-less skill-transition occurs automatically (i.e. by default).
	A skill terminator, used to indicate when a skill stops.
 1 Do-Something 1.1 Do-Something1 an-event 1.2 Do-Something2 an-event	A skill diagram illustrating decomposition of a skill by placing a rectangular box (the *decomposition box*) around the subskills. Also observe that the numeric identifier of each sub-skill relates to the parent skill.

Figure A.2: Skill Diagram Legend.

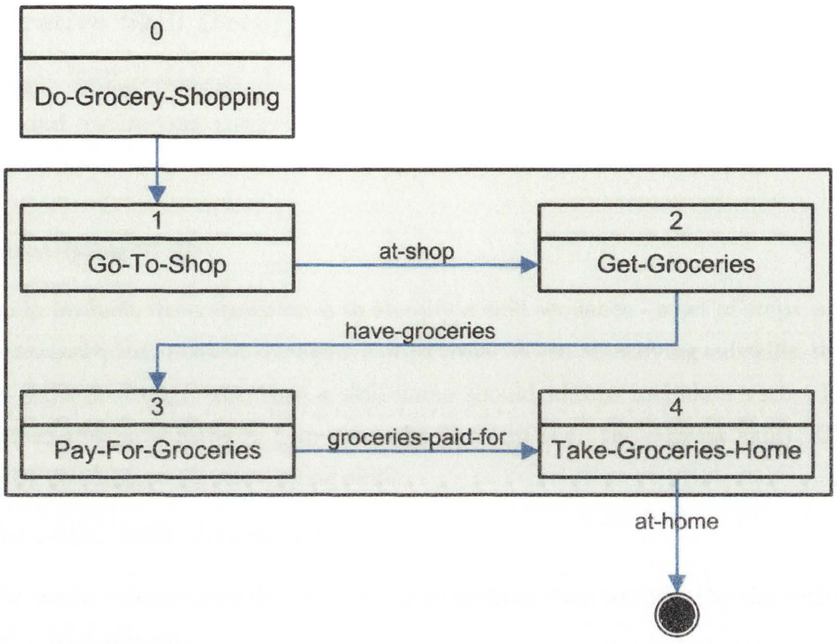


Figure A.3: Shopping Skill Diagram.

A.2.3 The Context-Level Skill

We use the term context-level skill to describe the mind's required capabilities, i.e. the entire behavioural and functional capabilities of the system. The context-level skill can be likened to the context-level diagram of a data-flow-diagram or the root of a tree - the context-level skill is the starting point for the iterative decomposition design process. To identify the context-level skill, the skill's numeric identifier is set to "0" (as per the convention used in data-flow-diagrams). Consider designing a robot whose sole purpose is to kick goals on a soccer field. The context-level skill for such a robot is displayed in Figure A.4.

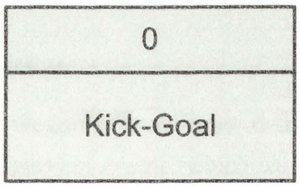


Figure A.4: A context-level skill diagram for a goal-kicking soccer robot.

A.2.4 Iterative Skill Decomposition

The context-level skill represents the objectives and requirements of the system, the robot, the environment, and the robot's current sensory and effector capabilities. The designer must now decompose the context-level skill into a set of implementable, collaborating skills.

A.2.4.1 Identifying Skills

The first step in problem decomposition is to identify a skill sequence - a set of steps, some of which may run concurrently, that achieve the agent's objectives. When identifying subskills, it is important to remember skills *do* things, and thus a skill name should *always* include a verb. The aim is to identify subskills that conceptually help solve the problem (i.e. the current skill). The following steps can assist the designer in identifying skills:

- Look for *verbs* - skills "do something".
- Look for *nouns* - the objects that are having something done to them (by the verb), e.g. "drive the *car*", "kick the *ball*".
- Skills should be named by their core and essential functionality, i.e. their objective, for example "`Get-Ball`" isn't "`Walk-To-Ball`" because we don't want to limit or restrict *how* we might get the ball. In other words, name skills in terms of *what* they achieve, rather than *how* they achieve it.
- Consider if each subskill should be generalised (i.e. abstracted). For example, should "`Get-Ball`" be "`Get-Object`"? In answering such questions the developer must weigh up the benefits of reuse versus the benefits of specialisation. In our current example, as the only object our robot will ever get is a ball, there is no need to generalise. Also, for an abstraction such as `Get-Object` to be created, a parameter to represent the type of object to be "got" needs to be created.

A.2.4.2 Identifying Skill Transitions

Decomposing skills into subskills is meaningless without defining how those skills interact. Skill transitions represent relevant and important events, which we also term "*whens*"[25]. Thus, identifying transition-conditions involves identifying when one skill should end, and when another should start. There are two types of transition condition - *entry-conditions* and *exit-conditions*, with the exit-condition of one skill being the entry-condition for another. When identifying skill-transitions, the designer should note that *every* skill (except for the context-level skill), should have at least one entry-condition (else, how does it start?).

To help identify skill-transitions, Go-Design employs *what-if analysis*. What-if analysis is a brainstorming approach in which the designer considers possible or potential situations or scenarios that may affect system performance. What-if analysis involves the designer hypothesising about *relevant* potential events, i.e. "what happens if ...?". For example, qualitative assessments of how a skill can terminate can assist in identifying skill transitions. For example:

- **What happens if the skill is successful?** Consider the skill's objective or objectives - if the objective or objectives are achieved, what should happen next? Should there be an exit-condition to another skill?
- **What happens if the skill is unsuccessful?** Skills will not always achieve their objective. Common types of unsuccessful exit conditions include:
  - **What happens if the skill "times out"?** In other words, can the skill take too long to achieve its objective, and if so, what should happen?
  - **What happens if a skill's entry-condition is no longer true during the skill's operation?** When designing skills, the designer should consider what knowledge is required to execute that skill. What should happen if a skill, once started, lacks insufficient knowledge to complete the skill?
  - **What happens if new events render the skill's continued operation irrelevant?** There are a large (probably infinite) number of potential events that could occur which could make any (or all) of the skills' continued operation irrelevant (e.g. the robot could physically break-down, or the sun could explode) - the task of the designer is to use their understanding of the domain to identify *likely* events.

A.2.5 Skill Templates

During basic-design, skill diagrams model only a skill's name and interactions with other skills. In many development circumstances (especially team environments), more documentation will be required so that all team members can gain a clear idea of each skill's purpose. To capture such information, Go-Design provides *skill templates* - documents which prompt the designer to identify key aspects of each skill. While many aspects of each skill's nature are implicit in their naming (e.g. the objective of `Get-Ball` is obviously "to get the ball"), and thus also implicit in the mind of the designer, skill templates provide means for *explicitly* documenting, in detail, aspects of each skill which can not be easily captured in a skill diagram. Skill templates describe:

- **The *objective* of the skill** - i.e. *why* is the skill being built? In other words, for what context-level requirement is it being built to (at least partially) satisfy?

- **A *description* of the skill** - *how* will the skill achieve its objective?
- ***The skill's entry-conditions and exit-conditions*** - i.e. *when* and *when not* the skill should and should not operate - imperative for establishing skill transitions and their transition conditions.
- **The *constraints* of the skill** - restrictions and limitations governing how the skill is implemented.
- **Performance dimensions and metric** for assessing the skill's performance.
- **The required performance standard**, described in terms of the performance dimensions and metric.
- Identification, where possible, of any **dependent skills**.

A skill template is displayed in Figure A.5.

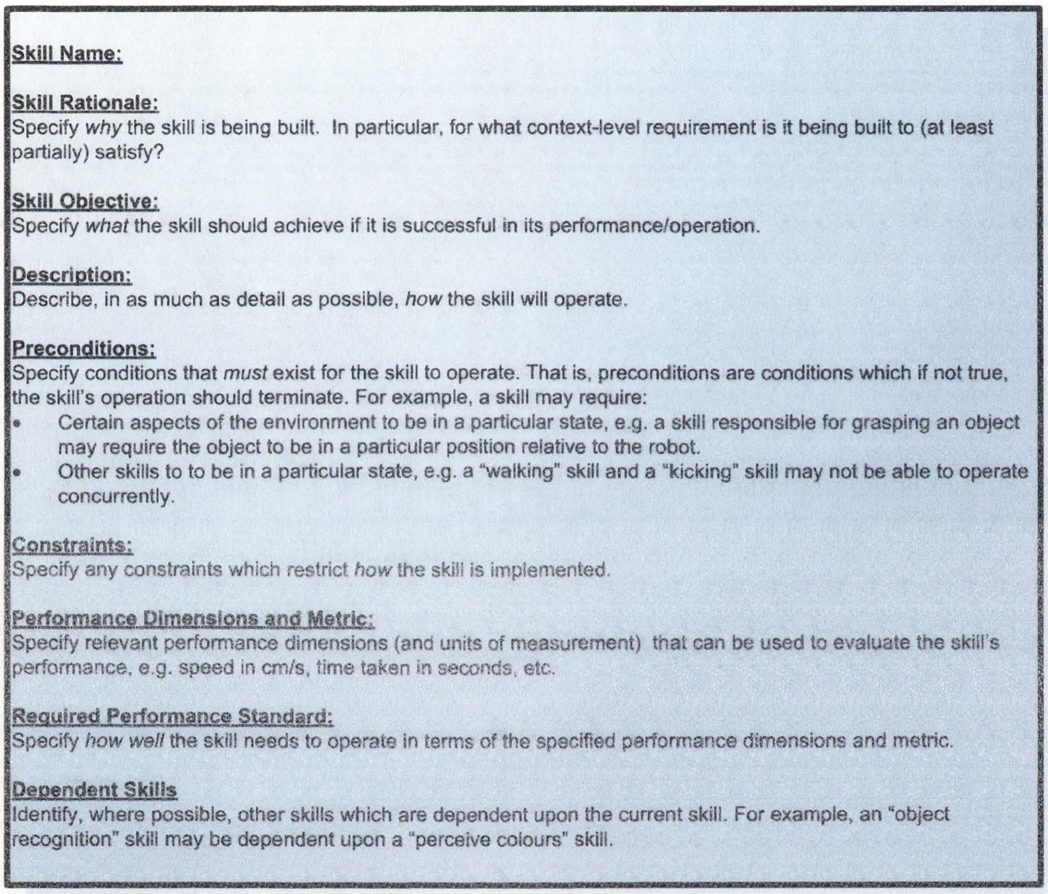
A.2.6 Design: Keeping it Simple

To keep diagrams simple, there are two main techniques that can be employed, both of which involve simplifying existing skill diagrams by creating new, smaller skill diagrams.

1. **Check for skills which can be layered** - that is, incorporated into a more general, higher-level skill (that is, moved closer to the context-level skill). Such skills often need to be executed regardless of what other skills the agent is performing, and tend to be more "reflex-like", e.g. general acts of perception, or detecting and recovering from accidents such as a legged robot falling over.
2. **Check for skills which can be amalgamated** - that is, check for multiple skills which can be amalgamated into a new skill. This can result in extra skill diagrams, but with each skill diagram containing fewer skills.

A.3 Detailed-Design

The product of basic-design is a skill architecture, consisting of skills and skill transitions. While a skill architecture can play a role in the high-level design of an autonomous robotic system, for the purpose of implementing a software solution more detail is required. Thus, a *detailed-design* is needed which provides a blueprint for translating a skill architecture into software.



Skill Name:

Skill Rationale:
Specify *why* the skill is being built. In particular, for what context-level requirement is it being built to (at least partially) satisfy?

Skill Objective:
Specify *what* the skill should achieve if it is successful in its performance/operation.

Description:
Describe, in as much as detail as possible, *how* the skill will operate.

Preconditions:
Specify conditions that *must* exist for the skill to operate. That is, preconditions are conditions which if not true, the skill's operation should terminate. For example, a skill may require:

- Certain aspects of the environment to be in a particular state, e.g. a skill responsible for grasping an object may require the object to be in a particular position relative to the robot.
- Other skills to to be in a particular state, e.g. a "walking" skill and a "kicking" skill may not be able to operate concurrently.

Constraints:
Specify any constraints which restrict *how* the skill is implemented.

Performance Dimensions and Metric:
Specify relevant performance dimensions (and units of measurement) that can be used to evaluate the skill's performance, e.g. speed in cm/s, time taken in seconds, etc.

Required Performance Standard:
Specify *how well* the skill needs to operate in terms of the specified performance dimensions and metric.

Dependent Skills
Identify, where possible, other skills which are dependent upon the current skill. For example, an "object recognition" skill may be dependent upon a "perceive colours" skill.

Figure A.5: A skill template.

A.3.1 Skill Types and Knowledge Representation

In detailed-design, we consider how to identify the knowledge, concepts, perceptions and decision-making processes required for a robot's mind to control that robot appropriately. Detailed-design involves:

- Identifying each skill's "type" as either an *action*, a *perception*, a *decision*, or a *behaviour*.
- Designing the representation each skill requires, including the *concepts* (data structures), *percepts* (the stored memory of a perception) and *memories* (all other forms of internal state).
- Documenting the requirements of each skill, including required *groundedness*.
- Identifying the dependencies between skills and knowledge representation through the use of *flows*.

Figure A.6 displays a summary of the different skill types, while Figure A.7 displays the different knowledge types, and Figure A.8 displays the notation for representing flows.

A.3.2 Identify Skill Types

The first step of detailed design is to identify each skill's *type*. Go-Design differentiates between four types of skills - *decisions*, *actions*, *perceptions* and *behaviours*. Every skill falls into one of these categories. Collaborations between these different skill types are used as the starting point in designing and modeling the required decision-making and representation for the robot control problem.

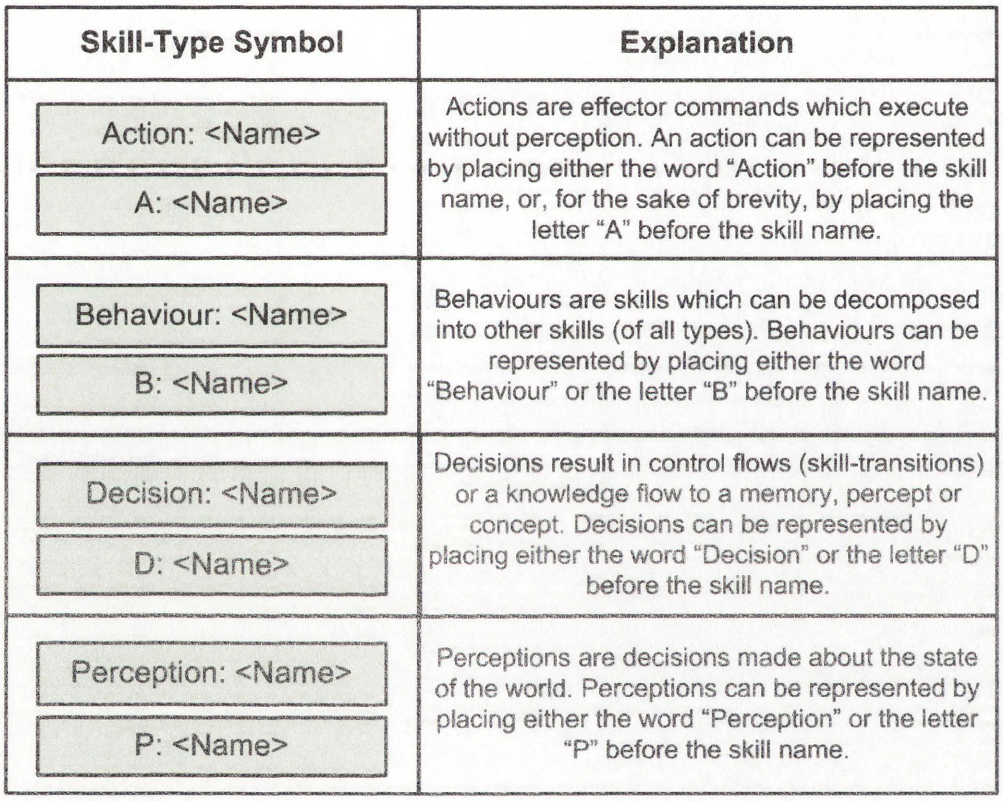
A.3.2.1 Decisions

Go-Design concerns two main types of decisions, namely *control management* and *knowledge management*. Control management decisions involve choosing between skill-transitions, while knowledge management concerns the manipulation of representation (internal state). Decisions range from high-level (even abstract or profound) reasoning (e.g. "what happened?", "what should I do now?", "why am I here?", etc) to at their lowest-level, simple "if statements" in code (e.g. "*if x is true then do a, else do b*").

A.3.2.2 Actions

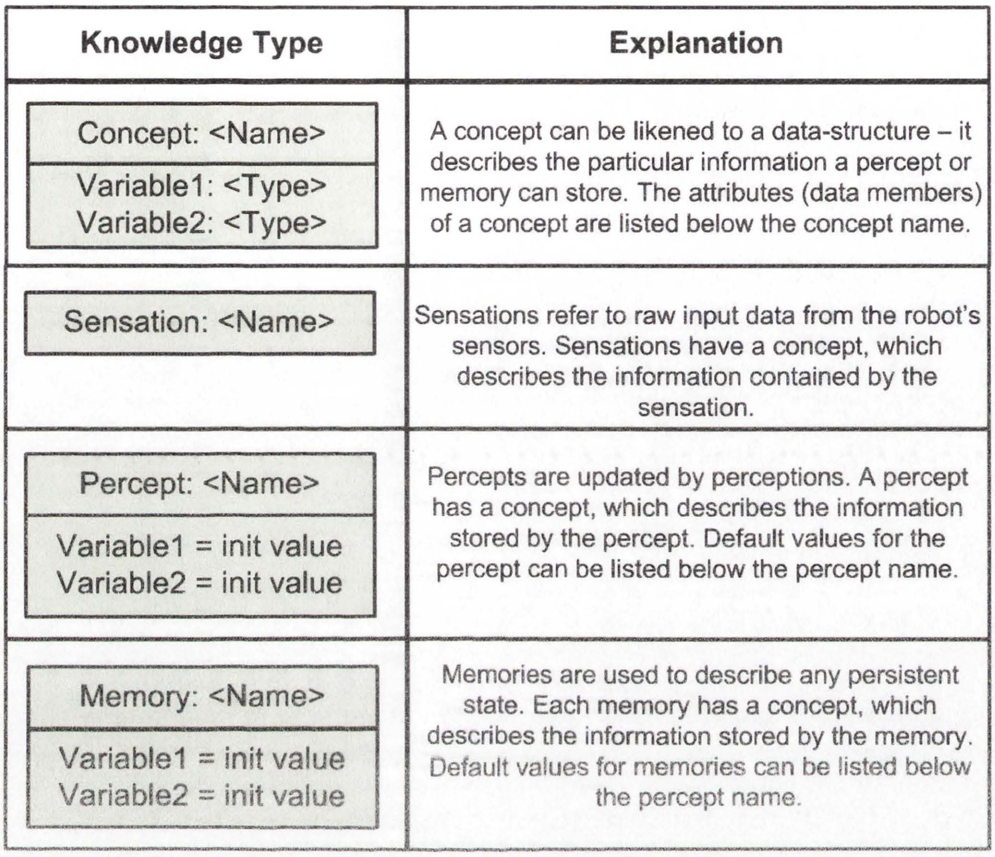
Actions are commands which:

1. Execute without perception.



Skill-Type Symbol	Explanation
Action: <Name> A: <Name>	Actions are effector commands which execute without perception. An action can be represented by placing either the word "Action" before the skill name, or, for the sake of brevity, by placing the letter "A" before the skill name.
Behaviour: <Name> B: <Name>	Behaviours are skills which can be decomposed into other skills (of all types). Behaviours can be represented by placing either the word "Behaviour" or the letter "B" before the skill name.
Decision: <Name> D: <Name>	Decisions result in control flows (skill-transitions) or a knowledge flow to a memory, percept or concept. Decisions can be represented by placing either the word "Decision" or the letter "D" before the skill name.
Perception: <Name> P: <Name>	Perceptions are decisions made about the state of the world. Perceptions can be represented by placing either the word "Perception" or the letter "P" before the skill name.

Figure A.6: Overview of skill types in Go-Design.

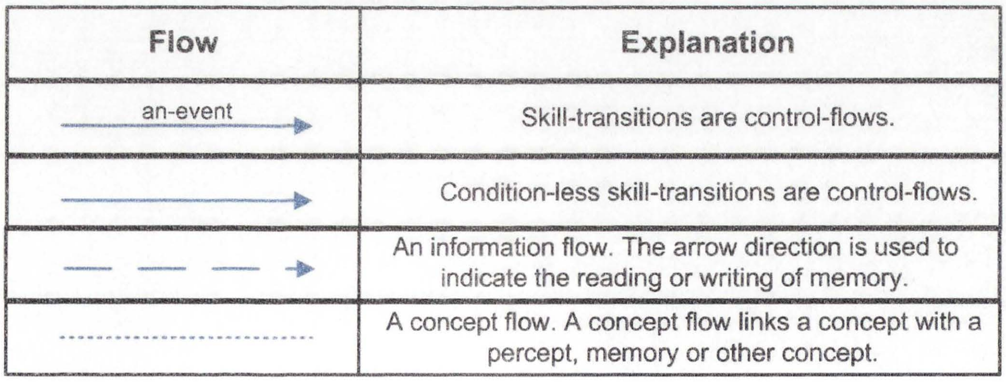


Knowledge Type	Explanation
Concept: <Name> Variable1: <Type> Variable2: <Type>	A concept can be likened to a data-structure – it describes the particular information a percept or memory can store. The attributes (data members) of a concept are listed below the concept name.
Sensation: <Name>	Sensations refer to raw input data from the robot's sensors. Sensations have a concept, which describes the information contained by the sensation.

Percept: <Name>
Variable1 = init value
Variable2 = init value
	Percepts are updated by perceptions. A percept has a concept, which describes the information stored by the percept. Default values for the percept can be listed below the percept name.

Memory: <Name>
Variable1 = init value
Variable2 = init value
	Memories are used to describe any persistent state. Each memory has a concept, which describes the information stored by the memory. Default values for memories can be listed below the percept name.

Figure A.7: Knowledge Representation in Go-Design.



Flow	Explanation
an-event →	Skill-transitions are control-flows.
→	Condition-less skill-transitions are control-flows.
— — — →	An information flow. The arrow direction is used to indicate the reading or writing of memory.
···························	A concept flow. A concept flow links a concept with a percept, memory or other concept.

Figure A.8: Flow types in Go-Design.

2. The developer can assume to be *deterministic* - that is, the designer can assume they "work" (but of course they might not, due to wear-and-tear or damage, etc).

In Go-Design, actions are not limited to those that have physical effects upon the world - they can also be virtual or software-based (e.g. saving a file to disk, or sending an email). Note, that while actions may be perception-less, some can be decomposed. As actions have no perception capabilities, actions can not evaluate their own performance. Thus, if evaluation of an action's performance is required, it must be provided by another type of skill. The term *primitive-action* refers to actions that can not be decomposed, e.g. actions provided by the robots programming interface.

A.3.2.3 Perceptions

Perceptions are decisions made about the state of the world. We use the term *perception* to refer to the process of perceiving, and the term *percept* to refer to the memory (the persistent state) of that perception. Perceptions contrast with *sensations*, as sensations refer to raw sensory input to the robotic system.

A.3.2.4 Behaviours

Lastly, behaviours are complex processes which can incorporate all the different types of skills, such as the ability to "drive a car". Thus, a behaviour can always be decomposed. During the design process, often the "behaviour" label is given to skills we havent designed how to do yet, as all behaviours can ultimately be decomposed into decisions, perceptions and actions.

A.3.2.5 Guidelines for Identifying Skill Types

To help identify skills, the following guidelines are provided:

- ***Behaviours*** - behaviours are composed of other skills, and thus need to be decomposed (unless the behaviour has been reused from a previous project, or is an external, 3rd party piece of software). Behaviours:
  - *Can always be decomposed* into other behaviours and/or perceptions.
  - *Require perception* to guide them - their performance is related to the state of the environment.
  - Are often *non-deterministic* (i.e. they may fail), e.g. due to incorrect assessment of the state of the environment.
  - *Take time* - they rarely occur instantaneously (as opposed to some actions, decisions, or perceptions, which may only take one processing cycle or frame to be completed).

  - Usually have a *physical effect* upon the world or the robot.

- ***Actions*** are commands which execute without perception, and are assumed by the designer to be deterministic. For example, commanding a motor on a legged robot to move to a particular angle, setting the colour of a pixel on a monitor, or making a sound. While actions do not use perception, they may use memory and decisions. For example, an action such as a "kick" may be executed as a series of joint angles over time, with memory storing the previous position (which is used to calculate the next position), and the current position of the motors is never actually sensed or queried. Actions:

  - Are *perception-less.*
  - Are *deterministic*, but of course any action may, in reality, fail. Thus, in deciding whether a skill should be an action or behaviour, the designer needs to consider whether perception is required to detect failure, or whether the risk of failure is so small or irrelevant that it can be ignored in terms of design.
  - Can be *composed* of other actions and decision.
  - Are often (but not always) *procedural* in design, in that they involve sequences of other actions or decisions.
  - Can be *physical* or *virtual* in their effect. Actions can be physical, such as moving a limb, or software-based such as sending an email, or performing a calculation.

- ***Decisions*** - Decisions *choose* - either between skill-transitions, the value of variables (memory), or to change the structure of a concept. A skill is a decision if:

  - The skill involves *choosing* between different *skill-transitions.*
  - The skill can be framed as an "*if-then-else*" statement.
  - The skill concerns the problem of *action-selection*, i.e. "when" to do "what".
  - The skill sets the value of (non-perceptual) variables in memory.

- ***Perceptions*** - Perceptions are a special type of decision - a decision about the state of the world, at least partly based upon immediate (i.e. current) sensory or perceptive information, with that belief about the world being stored in a percept. A skill is a perception if:

  - It involves storing information about the state of the external world, based upon the robot's sensory experience of the world.

  A skill is *not* a perception if:

  - It concerns action-selection or behaviour-selection, e.g. "if the floor is dirty then clean it" is a decision, while the assessment of whether the floor is dirty is a perception.
  - Likewise, "can I see the ball?" is a decision if it involves the querying of memory, whereas the actual process of "seeing" the ball is a perception.

A.3.3 Decompose Transition-Conditions

The next step in the design process is to define transition-conditions. To be capable of implementation, transition-conditions must be testable, logical conditions. In Go-Design, as each skill is responsible for testing their own exit-conditions, each skill with an exit transition-condition must be decomposed and designed. In designing implementations for each transition-condition, the designer must consider what is precisely *meant* by each transition-condition. That is, as skill-transitions are events, the next design step is to specify precisely *when* a transition-condition is true, and *how* it will be detected. As we are concerned with "how", and thus implementation, this will require not only decomposition, but also identification of knowledge requirements, such as *concepts* (data structures), *memories* and *percepts* (internal state), and *decisions.*

A.3.4 Individual Skill Design

In the previous section, the aim of designing transition-conditions guided the decomposition process. In this section we consider guidelines to assist with the design of individual skills.

A.3.4.1 Skill Design: Decisions

When designing decisions, ensure:

- **Is the decision a memory management decision or a control management decision?** In other words, what are the choices the decision can choose from? For a decision to be a decision, it must have choices. Do those choices concern memory management (writing to memory) or control management (choosing between skill transitions)?

- **For control management decisions, define transition-conditions for choosing between skill transitions.** If the decision results in multiple exit-transitions, check the mutual exclusivity of transition-conditions - i.e. can multiple exit-transitions from the same skill be true at the same time? If so, is this deliberate?

- **For memory management decisions, define input information flows, and the decision-making process for choosing the value(s) of the outgoing knowledge flows.** Memory management decisions rely on input information to make decisions, they then process that information, and then write new information to memory - identify all three processes.

- **Decompose, if possible, the decision.** If you, as the designer, can't explain how a skill works, it needs to be decomposed, exploded and designed. Remember, decisions can be composed of other decisions. Thus, if unable to represent algorithmically how the decisions works, identify new decisions which reduce the complexity of the problem.

A.3.4.2 Skill Design: Perceptions

Perceptions are decisions about the state of the world, and as such, the design of robust and accurate perceptive capabilities is imperative for the development of grounded robotic systems. The objective of a perception is to detect a specific real-world event or entity, with the *referent* of a perception being the "thing" being perceived. The designer should identify precisely the real-world phenomena a perception refers to, and this is especially important in team development environments, as all developers should have a shared understanding of the purpose of a perception. Thus, for each perception, *we need to define the specific event it refers to* in terms of logical conditions.

For each perception, the developer should:

- **Identify the referent of the perception.** In supporting documentation, identify the referent of the perception.
- **Identify what, exactly, needs to be perceived?** Every perception has a percept, and each percept has a concept. Thus, define the concept or concepts for the perception, therefore specifying the data structures for storing knowledge about the referent. Diagrammatically, every perception should have an outgoing knowledge-flow to a percept.
- **Identify *when* the perception should be operating.** Many perceptions will need to operate all the time, regardless of their dependent decision-making processes, and thus such perceptions should be scheduled at the context (root) level of the skill decomposition.
- **Identify the input information to the perception.** Every perception requires input knowledge, either from a sensation, or from other percepts. Thus, on a skill diagram, every perception should have at least one incoming knowledge from either a percept or a sensation.
- **Identify the required *groundedness* of the perception**. What are the dimensions of error? What is an acceptable margin of error?

A.3.4.3 Skill Design: Actions and Behaviours

The design of actions and behaviours is considered concurrently in Go-Design as actions form the building-blocks of behaviours. That is, while some behaviours can be decomposed into other behaviours, at least some behaviours in the skill decomposition must be composed of actions. In other words, all skill designs must map to actions the robot is capable of performing. The main difference between actions and behaviours is that actions are perception-less, while behaviours rely upon perception to guide them. In contrast to the design process employed in the rest of Go-Design, the design and development of actions can occur in a bottom-up manner. The term *primitive-action* refers to actions that can not be decomposed, e.g. actions provided by the robot's programming interface. Other actions are then layered on top of these, e.g. `Move-Leg` forms the basis of `Walk-A-Step`. Consequently, actions should always be tested in a bottom-up manner, e.g. if `Move-Leg` doesn't function as it should, this will affect `Walk-A-Step`, which will affect `Spin-On-Spot`, and so on. In contrast to the design of actions, the design of behaviour is a process of decomposition. All behaviours need to be decomposed into actions the robot is capable of performing.

When designing ***actions***:

- **Identify any memory required by the action.** Many actions (especially those which are not instantaneous in their effect) require internal state to remember the current state of the action's operation.
- **Ensure there is no use of perception** - otherwise, if perception is needed, treat the skill as a behaviour. Actions should be deterministic. If they are not they should be a behaviour.
- **Decompose, if possible, the action.** If you, as the designer, can't explain how an action works, it needs to be decomposed, exploded and designed. Remember, actions can be composed of both other actions and decisions (unless they are a primitive-action). Thus, if unable to represent algorithmically how the action works, identify new skills which reduce the complexity of the problem.

In contrast to the design process of actions, the design process for behaviours follows that outlined in basic-design (Chapter 6) - iterative decomposition until decisions, actions and perceptions can be identified.

A.3.5 Reviewing a Detailed Skill Architecture

At this point in the design process, a detailed skill-based architecture for a robot control problem is emerging. The skill architecture allows the designer to model hierarchical, modular, capability-based skills, and the interactions between these skills through event-based skill-transitions and transition-conditions. The guidelines presented in this chapter will assist in the designer in designing knowledge representation (concepts, percepts and memories), identification of the skills which manipulate and depend on that knowledge, and the design of actions and behaviours which control the robot's effectors. The last step in Go-Design is to review, and revise accordingly, the detailed-design, so as to ensure that the design is as thorough, consistent, and grounded as possible. In this section a skill design review checklist is presented.

A.3.5.1 Design Review Checklist

The designer should:

1. **Check all behaviours have been decomposed.** Behaviours, by definition, can be decomposed into other skills.
2. **Check skills that should run every frame (i.e. processing cycle) do run every frame**, i.e. they should never sit in any sequence after any possible skill terminators! Skill scheduling (e.g. every frame, on demand, and so forth) along with other assumptions about the skill's usage should be marked in the skill's template.
3. **For every decision, check there is at least one inward knowledge-flow.** Decisions require knowledge - they should not be made randomly.
4. **Check that each decision results in either a control management decision or a memory management decision.** A decision must make a choice, whether it be choosing between a skill transition or choosing the value of a memory or percept.
5. **Ensure all decisions are decomposed to the point where pseudocode can be used to represent the decision making process.**
6. **For every memory management decision, check there is an outward knowledge-flow to the modified memory.** If a decision modifies memory, ensure there is a knowledge-flow to that memory. If the memory contains multiple attributes (i.e. data members), the knowledge-flow should be labeled with the particular variable that is modified.
7. **Check every perception has a percept.** The result of a perception is stored in a percept, and should be indicated by a knowledge-flow.
8. **Check every percept has a concept.** Each percept should have a corresponding concept.
9. **Ensure every percept has a perception.**

10. **Check every perception has an input**. Every perception must have at least a sensation or a percept as an incoming information-flow.

11. **Avoid building perceptions with multiple exit-conditions**. Rather, for control management, construct a decision which uses the percept to arbitrate between skill-transitions.

12. **Check every memory has a corresponding concept**.

13. **Check memories are initialised with default values where appropriate**.

14. **Ensure each subskill's exit-transitions match those of the parent diagram**. When decomposing a skill the exit-transitions for that skill match those of the parent diagram.

15. **Check each skill's identifier and naming is unique and consistent.** During the design process the identifiers can lose sequence. Ensuring consistency with respect to each skill's identifer and naming avoids any confusion, especially in team development environments.

A.4 Detailed-Design Example Diagrams

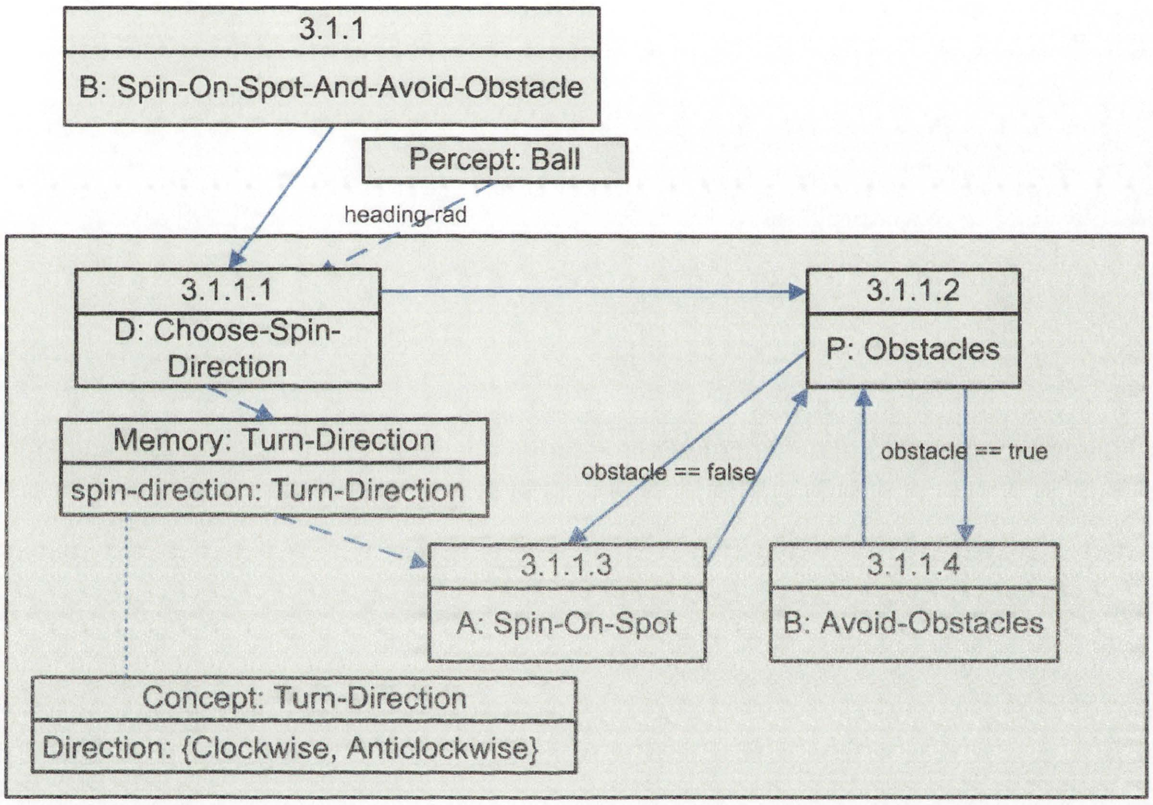


Figure A.9: `Spin-On-Spot-And-Avoid-Obstacle`. Note, the `Ball` percept is outside the decomposition box to represent the global nature of this percept.

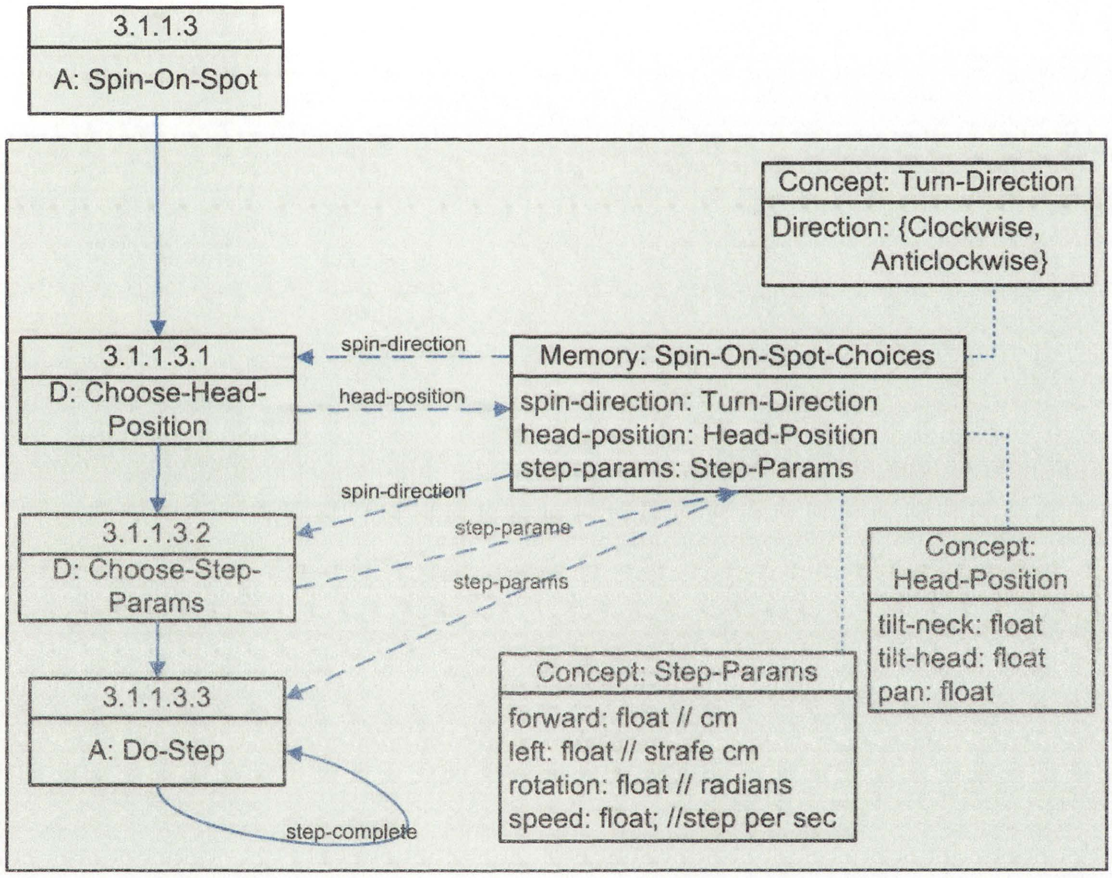


Figure A.10: `Spin-On-Spot`, implemented as an action, not a behaviour.

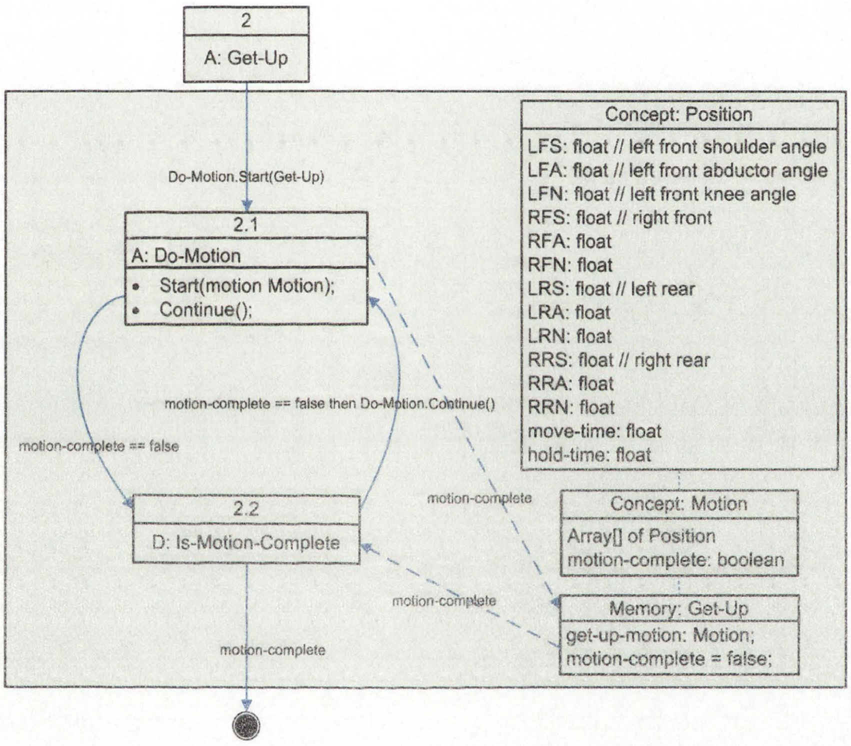
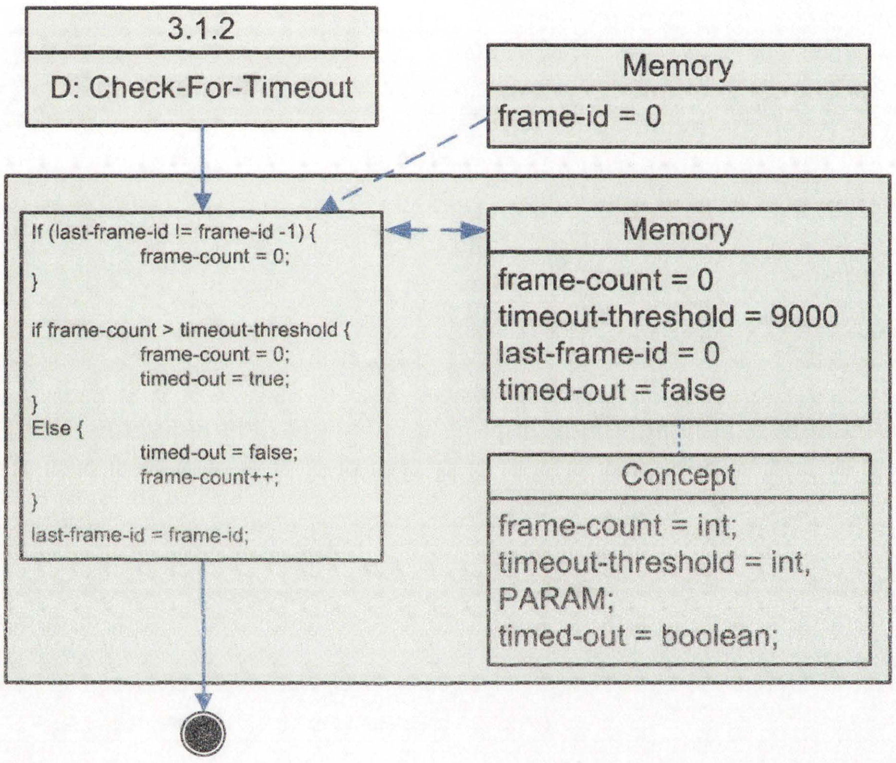


Figure A.11: Detailed skill diagram for `Get-Up`.



```
If (last-frame-id != frame-id -1) {
                frame-count = 0;
}

if frame-count > timeout-threshold {
                frame-count = 0;
                timed-out = true;
}
Else {
                timed-out = false;
                frame-count++;
}
last-frame-id = frame-id;
```

Figure A.12: An implementation of the decision `Check-For-Timeout`. Note, `frame-id` is outside the decomposition-box to represent the global nature of this variable.

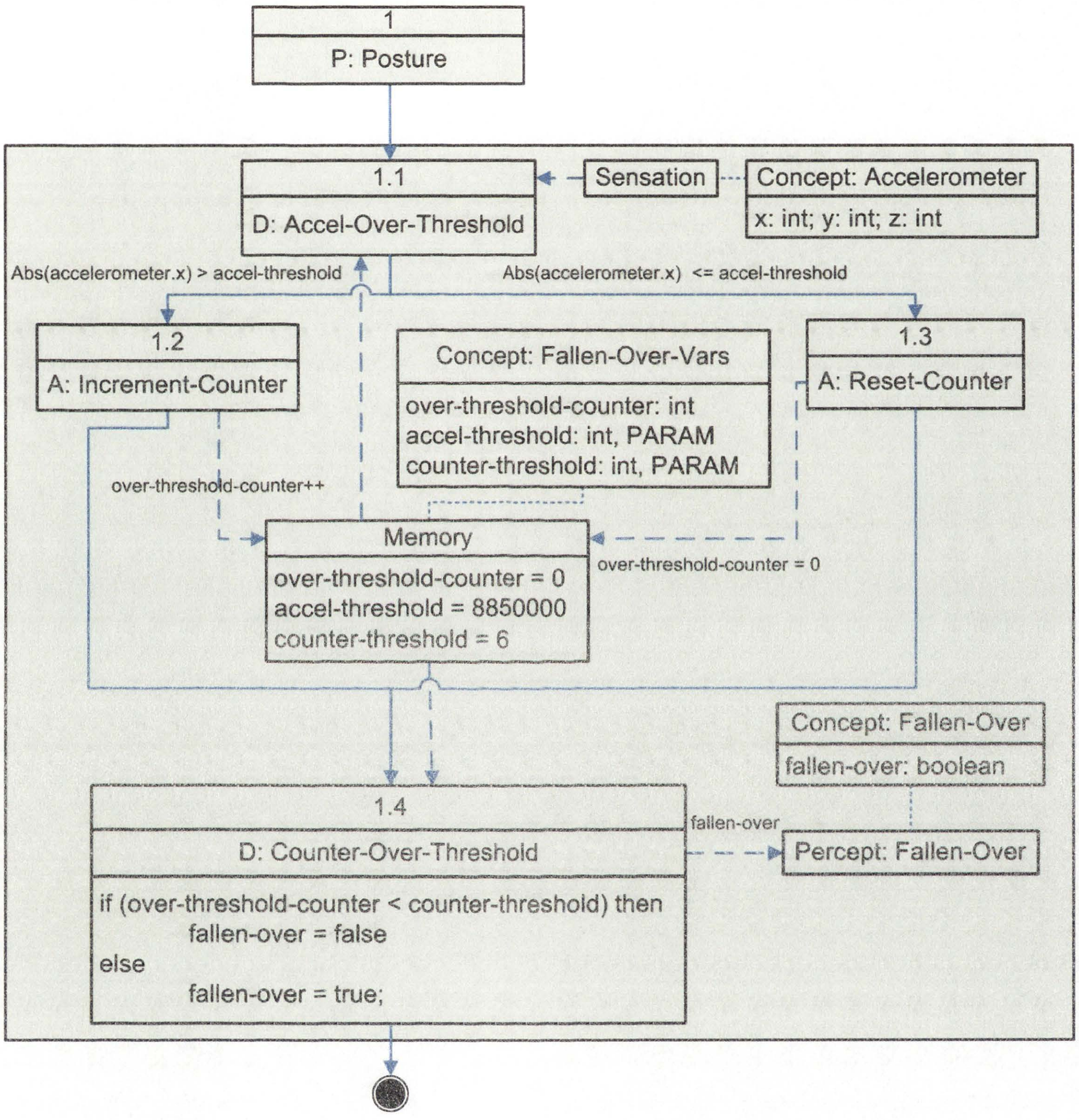

1
P: Posture
1.1
D: Accel-Over-Threshold
Sensation
Concept: Accelerometer
x: int; y: int; z: int
Abs(accelerometer.x) > accel-threshold
Abs(accelerometer.x) <= accel-threshold
1.2
A: Increment-Counter
Concept: Fallen-Over-Vars
over-threshold-counter: int
accel-threshold: int, PARAM
counter-threshold: int, PARAM
1.3
A: Reset-Counter
over-threshold-counter++
Memory
over-threshold-counter = 0
accel-threshold = 8850000
counter-threshold = 6
over-threshold-counter = 0
Concept: Fallen-Over
fallen-over: boolean
1.4
D: Counter-Over-Threshold
if (over-threshold-counter < counter-threshold) then
fallen-over = false
else
fallen-over = true;
fallen-over
Percept: Fallen-Over

Figure A.13: Detailed-Design for `Perceive-Posture`.

Table A.1: Requirements Elicitation Checklist

Describe the behaviour of the robotic system.
Using as much as detail as possible, describe the behaviour of the system.

Specify behavioural requirements.,
Using the behavioural description, identify specific *measures of performance* (*performance benchmarks*.

Identify Perceptual Requirements.
Use the behavioural description and behavioural requirements to identify specific aspects of the robot's environment (including its own state) that must be perceived. Consider:

- Are there any *physical objects* that need to be perceived? e.g the soccer ball;
- What *attributes of those physical objects* need to be perceived? e.g. the location of the soccer ball;
- What aspects of the *robot's physical state* need to be perceived? e.g. fallen over, low fuel, etc.
- What *events* needs to be perceived? i.e. are there any significant changes in the state of the environment (and robot) that require an appropriate action? e.g. a change in the trajectory of a moving soccer ball.

Identify locomotion requirements (movement and action). Locomotion requirements arise from behavioural requirements. e.g. how is the robot required to move around the environment? For example:

- *Directional requirements.* What are the directional requirements? Does the robot only have to move backwards and forwards? Turning? Strafing? etc.
- *Speed requirements.* How quickly should each type of movement be capable of being executed? e.g. straight-line speed? turning and rotational speed?
- *Stability requirements.* How stable does the robot need to be move while moving?
- *Special actions.* What actions (other than movement) are required by the robot? e.g. the ability to grip an object? the ability to get up after a fall (legged robots), the ability to kick a ball, the ability to climb steps?
- *Terrain.* Is the robot's movement requirements related to different types of terrain? What is the nature of the terrain the robot is required to move over? (or move through, in the case of autonomous ships, planes, helicopters, submarines, etc).

Identify Communication Requirements.
Identify the communication needs of the robotic system. For example, what communication capabilities are required for the robot(s) to communicate with:

- Developers? (i.e. for debugging or reporting)
- Users? e.g. the security robot needs to report the presence of intruders to a security guard.
- Other robots and software agents? e.g. soccer playing robots might tell their teammates where they intend to kick the ball.

Identify Environmental Requirements.
In what conditions is the robotic system expected to operate? Under water? On the moon? In changing light? In controlled light? Flat terrain? Uneven terrain? etc

Bibliography

[1] P. Agre. *Computational Theories of Interaction and Agency*, chapter Computational research on interaction and agency, pages 1–52. MIT Press, Cambridge, MA, 1996.

[2] P. Agre and S. Rosenschein, editors. *Computational Theories of Interaction and Agency.* MIT Press, Cambridge, MA, 1996.

[3] J. Albus. The NIST Real-time Control System (RCS): an approach to intelligent systems research. *Journal of Experimental and Theoretical Artificial Intelligence*, 9:157–174, 1997.

[4] J. Albus and A. Barbera. RCS: A cognitive architecture for intelligent multi-agent systems. *Annual Reviews in Control*, 29(1):89–99, 2005.

[5] M. Anderson. Embodied cognition: A field guide. *Artificial Intelligence*, 149:91–130, 2003.

[6] R. Arkin. *Behavior-Based Robotics.* MIT Press, Cambridge, MA, 1998.

[7] J-C. Baillie. Grounding symbols in perception with two interacting autonomous robots. In L. Berthouze, H. Kozima, C. Prince, G. Sandini, G. Stojanov, G. Metta, and C. Balkenius, editors, *Proceedings of the 4th International Workshop on Epigenetic Robotics: Modeling Cognitive Development in Robotic System*, pages 107–110. Lund University Cognitive Studies 117, 2004.

[8] L. Barsalou. Ad-hoc categories. *Memory and Cognition*, 11(3):211–227, 1983.

[9] L. Barsalou. Perceptual symbol systems. *Behavioral and Brain Sciences*, 22:577–609, 1999.

[10] L. Barsalou, W. Yeh, B. Luka, K. Olseth, K. Mix, and L. Wu. Concepts and meaning. In K. Beals, G. Cooke, D. Kathman, K. McCullough, S. Kita, and D. Testen, editors, *Chicago Linguistics Society 29: Papers from the parasession on conceptual representations*, pages 23–61. University of Chicago: Chicago Linguistics Society, 1993.

[11] M. Bickhard. Representational content in humans and machines. *Journal of Experimental and Theoretical Artificial Intelligence*, 5:285–333, 1993.

[12] D. Blank, D. Kumar, and L. Meeden. A developmental approach to anchoring. Computer science technical report, Bryn Mawr College, 2002.

[13] D. Blank, D. Kumar, and L. Meeden. A developmental approach to intelligence. In Sumali J. Conlon, editor, *Proceedings of the Thirteenth Annual Midwest Artificial Intelligence and Cognitive Science Society Conference*, 2002.

[14] D. Blank, D. Kumar, L. Meeden, and J. Marshall. Bringing up robot: Fundamental mechanisms for creating a self-motivated, self-organizing architecture. *Cybernetics and Systems*, 36(2):125–150, 2005.

[15] A. Borghi. Object concepts and embodiment: Why sensorimotor and cognitive processes cannot be separated. *La nuova critica*, 49(50):90–107, 2007.

[16] K. Bowyer and P. Phillips. Overview of work in empirical evaluation of computer vision algorithms. In K. W. Bowyer and P. J. Phillips, editors, *Empirical Evaluation Techniques in Computer Vision*, IEEE Comp Press, CA, USA, 1998.

[17] G. Box. *Robustness in Statistics*, chapter Robustness in the strategy of scientific model building. Academic Press, New York, 1979.

[18] S. Brennan. *Social and cognitive psychological approaches to interpersonal communication*, chapter The Grounding Problem in Conversations With and Through Computers, pages 201–225. Lawrence Erlbaum, Hillsdale, NJ, 1998.

[19] J. Briscoe, editor. *Linguistic evolution through language acquisition: formal and computational models*. Cambridge University Press, Cambridge, 2002.

[20] R. A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2:1423, April 1986.

[21] R. A. Brooks. Elephants don't play chess. *Robotics and Autonomous Systems*, 6:3–15, 1990.

[22] R. A. Brooks. Intelligence without reason. In *Proceedings of 12th Int. Joint Conf. on Artificial Intelligence*, page 569595, Sydney, Australia, August 1991.

[23] R. A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47:139–160, 1991.

[24] R. A. Brooks. New approaches to robotics. *Science*, 253:1227–1232, 1991.

[25] J. Bryson. *Intelligence by Design: Principles of Modularity and Coordination for Engineering Complex Adaptive Agents*. PhD thesis, MIT, 2001.

[26] C. Burgess and K. Lund. Modelling parsing constraints with high-dimensional context space. *Language and Cognitive Processes*, 12:177210, 1997.

[27] J. Cahn and S. Brennan. A psychological model of grounding and repair in dialog. In *Proceedings of AAAI Fall Symposium on Psychological Models of Communication in Collaborative Systems*, pages 25–33, North Falmouth, MA, 1999. American Association for Artificial Intelligence.

[28] A. Cangelosi. Symbol grounding in connectionist and adaptive agent models. In S.B. Cooper, B. Lowe, and L. Torenvliet, editors, *New Computational Paradigms: Proceedings of the First Conference on Computability in Europe, CiE 2005*, volume LNCS 3526, pages 69–74, Berlin, 2005. Heidelberg: Springer-Verlag.

[29] A. Cangelosi. The grounding and sharing of symbols. *Pragmatics and Cognition*, 14(2):275–285, 2006.

[30] A. Cangelosi, A. Greco, and S. Harnad. From robotic toil to symbolic theft: Grounding transfer from entry-level to higher-level categories. *Connection Science*, 12(2):143–162, 2000.

[31] A. Cangelosi, A. Greco, and S. Harnad. Symbol grounding and the symbolic theft hypothesis. In A. Cangelosi and D. Parisi, editors, *Simulating the Evolution of Language*, chapter 9, pages 191–210. Springer Verlag, London, 2002.

[32] A. Cangelosi and S. Harnard. The adaptive advantage of symbolic theft over sensorimotor toil: Grounding language in perceptual categories. *Evolution of Communication*, 4(1):117–142, 2001.

[33] A. Cangelosi, E. Hourdakis, and V. Tikhanoff. Language acquisition and symbol grounding transfer with neural networks and cognitive robots. In *2006 IEEE World Congress on Computational Intelligence*, pages 2885–2891. IJCNN, IEEE Press, 2006.

[34] A. Cangelosi and D. Parisi, editors. *Simulating the Evolution of Language*. Springer, London, 2002.

[35] A. Cangelosi and T. Riga. An embodied model for sensorimotor grounding and grounding transfer: Experiments with epigenetic robots. *Cognitive Science*, 30(4):673–689, 2006.

[36] D. Chalmers. *The Symbolic and Connectionist Paradigms: Closing the Gap*, chapter Subsymbolic Computation and the Chinese Room. Lawrence Erlbaum, 1992.

[37] A. Chella, H. Dindo, and I. Infantino. Anchoring by imitation learning in conceptual spaces. In *AI*IA 2005: Advances in Artificial Intellgience*, pages 495–506. Springer, 2005.

[38] A. Chella, M. Frixione, and S. Gaglio. Conceptual spaces for computer vision representations. *Artificia Intelligence Review*, 16(2):137–152, 2001.

[39] A. Chella, M. Frixione, and S. Gaglio. Anchoring symbols to conceptual spaces: the case of dynamic scenarios. *Robotics and Autonomous Systems*, 43(2):175–188, 2003.

[40] A. Chella, M. Frixione, and R. Pirrone. Conceptual representations of actions for autonomous robots. *Robotics and Autonomous Systems*, 34:251–263, 2001.

[41] R. Chrisley and T. Ziemke. Embodiment.

[42] M. Christiansen and N. Chater. Symbol grounding - the emperor's new theory of meaning? In *Proceedings of the 15th Annual Conference of the Cognitive Science Society*, pages 155–160, June 1993.

[43] H. Clark and S. Brennan. *Perspectives on socially shared cognition*, chapter Grounding in communication, pages 127–149. APA, Washington, DC, 1991.

[44] H. Clark and E. Schaefer. Contributing to discourse. *Cognitive Science*, 13:259–294, 1989.

[45] P. Cohen, T. Oates, and C. Beal. Robots that learn meanings. In *First Joint Conference of Autonomous Agents and Multiagent Systems*, 2002.

[46] P. Cohen, T. Oates, C. Beal, and N. Adams. Contentful mental states for robot baby. In *Eighteenth national conference on Artificial intelligence*, pages 126–131, Menlo Park, CA, USA, 2002. American Association for Artificial Intelligence.

[47] A. Collins and E. Loftus. A spreading activation theory of semantic memory. *Psychological Review*, 82:407–428, 1975.

[48] S. Coradeschi and A. Saffiotti. Anchoring symbols to vision data by fuzzy logic. In A. Hunter and S. Parsons, editors, *Symbolic and Quantitative Approaches to Reasoning and Uncertainty*, number 1638 in LNCS, pages 104–115. Springer-Verlag, 1999. Online at http://www.aass.oru.se/~asaffio/.

[49] S. Coradeschi and A. Saffiotti. Anchoring symbols to sensor data: Preliminary report. In *AAAI/IAAI*, pages 129–135, 2000.

[50] S. Coradeschi and A. Saffiotti. Perceptual anchoring of symbols for action. In *Proceedings of the 17th IJCAI Conference*, pages 407–412, Seattle, WA, 2001. Online at http://www.aass.oru.se/~asaffio/.

[51] S. Coradeschi and A. Saffiotti. An introduction to the anchoring problem. *Robotics and Autonomous Systems*, 43(2-3):85–96, 2003. Special issue on perceptual anchoring. Online at http://www.aass.oru.se/Agora/RAS02/.

[52] K. Coventry and S. Garrod. Saying, seeing and acting. the psychological semantics of spatial prepositions. In *Essays in Cognitive Psychology*. Lawrence Erlbaum Associates, 2004.

[53] P. Davidsson. Toward a general solution to the symbol grounding problem: Combining machine learning and computer vision. *AAAI Fall Symposium Series, Machine Learning in Computer Vision: What, Why and How?*, pages 157–161, 1993.

[54] G. Dorffner and E. Prem. Connectionism, symbol grounding, and autonomous agents. 1993.

[55] H. Dreyfus. *What computers still can't do: a critique of artificial reason*. The MIT Press, London, England, 3rd edition, 1993.

[56] L. Floridi. Open problems in the philosophy of information. *Metaphilosophy*, 35:554–582, 2004.

[57] P. Gärdenfors. *Conceptual Spaces: The Geometry of Thought*. A Bradford Book, MIT, London, 2000.

[58] P. Gärdenfors. *How Homo Became Sapiens: On the Evolution of Thinking*. Oxford University Press, London, 2003.

[59] A. Glenberg, D. Havas, R. Becker, and M. Rinck. *The grounding of cognition: The role of perception and action in memory, language, and thinking*, chapter Grounding Language in Bodily States: The Case for Emotion. Cambridge University Press, Cambridge, 2005.

[60] A. Glenberg and M. Kaschak. Grounding language in action. *Psychonomic Bulletin and Review*, 9:558–565, 2002.

[61] A. Glenberg and D. Robertson. Indexical understanding of instructions. *Discourse Processes*, 28:1–26, 1999.

[62] A. Glenberg and D. Robertson. Symbol grounding and meaning: A comparison of high-dimensional and embodied theories of meaning. *Journal of Memory and Language*, 43:379–401, 2000.

[63] S. Goonatilake and S. Khebbal, editors. *Intelligent Hybrid Systems*. Wiley, 1995.

[64] P. Gorniak and D. Roy. Probabilistic grounding of situated speech using plan recognition and reference resolution. In *Seventh International Conference on Multimodal Interfaces*, 2005.

[65] A. Greco, A. Cangelosi, and S. Harnad. A connectionist model for categorical perception and symbol grounding. In *Proceedings of the 15th An nual Workshop of the European Society for the Study of Cognitive Systems*, 1998.

[66] J. Halpern and D. Koller. Representation dependence in probabilistic inference. In Chris Mellish, editor, *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1853–1860, San Francisco, 1995. Morgan Kaufmann.

[67] S. Harnad. The symbol grounding problem. *Physica*, 42:335–346, 1990.

[68] S. Harnad. Grounding symbols in the analog world with neural nets. *Think*, 2(1):12–18, 1993.

[69] S. Harnad. Problems, problems: The frame problem as a symptom of the symbol grounding problem. *Psycoloquy*, 4(34), 1993.

[70] S. Harnad. Symbol grounding is an empirical problem: Neural nets are just a candidate component., 1993.

[71] S. Harnad. *The "artificial life" route to "artificial intelligence." Building Situated Embodied Agents*, chapter Grounding Symbolic Capacity in Robotic Capacity, pages 276–286. Lawrence Erlbaum, New Haven, 1995.

[72] S. Harnad. The symbol grounding problem. In *Encyclopedia of Cognitive Science*. Nature Publishing Group/Macmillan, London, 2003.

[73] S. Harnad. *Handbook of Categorization in Cognitive Science*, chapter To cognize is to categorize: Cognition is categorization. Elsevier, Amsterdam, 2005.

[74] C. Ho. Some phenomena of problem decomposition strategy for design thinking: Differences between novices and experts. *Design Studies*, 22(1):27–45, 2001.

[75] C. Hudelot, N. Maillot, and M. Thonnat. Symbol grounding for semantic image interpretation: From image data to semantics. In *Proceedings of International Workshop on Semantic Knowledge in Computer Vision, ICCV 2005*, 2005.

[76] N. Jennings. Agent-based computing: Promise and perils. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 1429 – 1436, San Francisco, 1999. Morgan Kaufmann Publishers.

[77] N. Jennings. On agent-based software engineering. *Artificial Intelligence*, 117:277296, 2000.

[78] N. Jennings and M. Wooldridge. Agent-Oriented Software Engineering. In Francisco J. Garijo and Magnus Boman, editors, *Proceedings of the 9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World : Multi-Agent System Engineering (MAAMAW-99)*, volume 1647, pages 1–7. Springer-Verlag: Heidelberg, Germany, 30– 2 1999.

[79] D. Jurafsky and J. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics and Speech Recognition.* Prentice Hall, 2000.

[80] C. Kennedy. A conceptual foundation for autonomous learning in unforeseen situations. In *Proceedings of the IEEE International Symposium on Intelligent Control (ISIC/CIRA/ISIS'98)*, Gaithersburg, Maryland, September 1998.

[81] C. Kennedy. Reducing indifference: Steps towards autonomous agents with human concerns. In *Proceedings of the Symposium on AI, Ethics and (Quasi-) Human Rights*, Birmingham, April 2000. Society for the Study of Artificial Intelligence and the Simulation of Behaviour (AISB'00).

[82] S. Kirby. Natural language from artificial life. *Artificial Life*, 8:185215, 2002.

[83] T. Kohonen. *Self Organizing Maps.* Springer, N.Y., 2001.

[84] T. Landauer, P. Foltz, and D. Laham. Introduction to latent semantic analysis. *Discourse Processes*, 25:259–284, 1998.

[85] C. Larman. *Agile and Iterative Development: A Manager's Guide.* Addison Wesley Professional, 2003.

[86] D. Law and R. Mikkaulainen. Grounding robotic control with genetic neural networks. *Tech. Rep., Univ. of Austin, Texas*, AI94223, 1994.

[87] D. Lenat. Cyc: A large-scale investment in knowledge infrastructure. *Communications of the ACM*, 38(11):33–38, 1995.

[88] L. Liikkanen. Exploring problem decomposition in conceptual design. Master's thesis, Department of Psychology, Universtiy of Helsinki, 2005.

[89] M. Risler M. Loetzsch and M. Jungel. Xabsl - a pragmatic approach to behavior engineering. In *In Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2006)*, pages 5124–5129, Beijing, 20016.

[90] B. MacLennan. Grounding analog computers. *Think*, 2(1):48–52, 1993.

[91] G. Marcus. *The Algebraic Mind : Integrating Connectionism and Cognitive Science Learning, Development, and Conceptual Change.* MIT Press, Cambridge, Mass, 2001.

[92] M. Mayo. Symbol grounding and its implication for artificial intelligence. In *Twenty-Sixth Australian Computer Science Conference*, pages 55–60, 2003.

[93] J. McCarthy and P. Hayes. *Machine Intelligence*, chapter Some Philosophical Problems from the Standpoint of Artificial Intelligence, pages 463–502. Edinburgh University Press, Edinburgh, 1969.

[94] M. Mitchell. *An Introduction to Genetic Algorithms.* MIT Press, Cambridge, MA, 1996.

[95] A. Monk. *HCI models, theories and frameworks: towards a multidisciplinary science*, chapter Common ground in electronically mediated communication: Clark's theory of language use, pages 265–289. Morgan Kaufmann, San Francisco, 2003.

[96] R. Nakisa and K. Plunkett. Evolution of a rapidly learned representation for speech. *Language and Cognitive Processes*, 13(1):105–127, 1998.

[97] A. Newell and H. Simon. Computer science as empirical enquiry: Symbols and search. *Communications of the ACM*, 19, 3:113–126 year=1976,.

[98] K. Pastra. Viewing vision-language integration as a double-grounding case. In *Proceedings of the American Association of Artificial Intelligence (AAAI) Fall Symposium on "Achieving Human-Level Intelligence through Integrated Systems and Research"*, Washington D.C., USA, 2004

[99] K. Pastra. *Vision Language Integration: a Double-Grounding Case.* PhD thesis, Department of Computer Science, University of Sheffield, U.K., 2004.

[100] C. S. Peirce. *Collected Papers of Charles Sanders Peirce*, volume 1-6. Harvard University Press, Cambridge, MA, 1931-1935.

[101] R. Pfeifer and G. Gmez. Interacting with the real world: design principles for intelligent systems. *Artificial Life and Robotics*, 9(1):1–6, 2005.

[102] R. Pfeifer and P. Verschure. *The Artificial Life Route to Artificial Intelligence*, chapter The Challenge of Autonomous Agents: Pitfalls and How to Avoid Them, pages 237–263. Lawrence Erlbaum, Hillsdale, New Jersey, 1995.

[103] E. Prem. Dynamic symbol grounding, state construction and the problem of teleology. In J. Mira and F. Sandoval, editors, *From Natural to Artificial Neural Computation. International*

Workshop on Artificial Neural Networks. Proceedings, pages 619–26. Springer-Verlag, Berlin, Germany, 1995.

[104] C. Prince. Theory grounding in embodied artificially intelligent systems. In *Proceedings of the First International Workshop on Epigenetic Robotics: Modeling Cognitive Development in Robotic Systems*, Lund, Sweden, 2001.

[105] H. Putnam. Meaning and reference. *The Journal of Philosophy*, 70(19):699–711, 1973.

[106] T. Riga, A. Cangelosi, and A. Greco. Symbol grounding transfer with hybrid self-organizing/supervised neural networks. In *IJCNN04 International Joint Conference on Neural Networks*, Budapest, July 2004.

[107] D. Roy. Grounding words in perception and action: computational insights. *Trends in Cognitive Sciences*, 9(8):389–396, 2005.

[108] D. Roy. Semiotic schemas: A framework for grounding language in action and perception. *Artificial Intelligence*, 167(1-2):170–205, 2005.

[109] D. Roy. A computational model of three facets of meaning. In M. de Vega, A. Glenberg, and A. Graesser, editors, *Proceedings of the Garachico Workshop: Symbols, Embodiment and Meaning*, December 2006.

[110] D. Roy and E. Reiter. Connecting language to the world. *Artificial Intelligence*, 167(1-2):1–12, September 2005.

[111] N. Sales and R. Evans. An approach to solving the symbol grounding problem: Neural networks for object naming and retrieval. In *Proc. CMC-95*, 1995.

[112] J. R. Searle. Minds, brains, and programs. *Behavioral and Brain Sciences*, 3:417–457, 1980.

[113] M. Shanahan. Perception as abduction: Turning sensor data into meaningful representation. *Cognitive Science*, 29(1):103–134, 2005.

[114] H. Simon. *The sciences of the artificial.* MIT Press, Cambridge, MA, third edition edition, 1996.

[115] R. Spiegel and I. McLaren. Recurrent neural networks and symbol grounding. In *Proceedings. IJCNN '01.*, volume 1, pages 320-325, Washington, DC, USA, 2001.

[116] C. Stanton and M-A Williams. Grounding robot sensory and symbolic information using the semantic web. In *RoboCup 2003: Robot Soccer World Cup VII*, Lecture Notes in Computer Science, pages 757–764. Springer, 2003.

[117] L. Steels. When are robots intelligent autonomous agents? *Journal of Robotics and Autonomous Systems*, 15:3–9, 1995.

[118] L. Steels. Perceptually grounded meaning creation. In M. Tokoro, editor, *Proceedings of the International Conference on Multi-Agent Systems*, Cambridge, MA, 1996. MIT Press.

[119] L. Steels. Language games for autonomous robots. *IEEE Intelligent systems*, pages 17–22, October 2001.

[120] L. Steels. Evolving grounded communication for robots. *Trends in Cognitive Science*, 7(7):308–312, July 2003.

[121] L. Steels. The symbol grounding problem is solved, so what's next? In M. De Vega, G. Glennberg, and G. Graesser, editors, *Symbols, embodiment and meaning.* Academic Press, New Haven, 2007.

[122] L. Steels and F. Kaplan. *Linguistic evolution through language acquisition: formal and computational models*, chapter Bootstrapping Grounded Word Semantics, pages 53–74. Cambridge University Press, 2002.

[123] L. Steels and P. Vogt. Grounding adaptive language games in robotic agents. In I. Harvey and P. Husbands, editors, *Advances in Artificial Life. Proceedings of the Fourth European Conference on Artificial Life*, Cambridge, MA, 1997. MIT Press.

[124] R. Sun. Symbol grounding: A new look at an old idea. *Philosophical Psychology*, 13(2):149–172, 2000.

[125] R. Sun. *Duality of the Mind.* Lawrence Erlbaum Associates, 2002.

[126] R. Sun and F. Alexandre, editors. *Connectionist-Symbolic Integration: From Unified to Hybrid Approaches.* Lawrence Erlbaum Associates, 1997.

[127] S. Swarup, K. Lakkaraju, S. Ray, and L. Gasser. Symbol grounding through cumulative learning. In P. Vogt et al., editor, *Symbol Grounding and Beyond: Proceedings of the Third International Workshop on the Emergence and Evolution of Linguistic Communication*, pages 180–191. Springer, 2006.

[128] M. Taddeo and L. Floridi. Solving the symbol grounding problem: a critical review of fifteen years of research. *Journal of Experimental and Theoretical Artificial Intelligence*, 2005.

[129] J. Taylor and S. Burgess. Steve austin versus the symbol grounding problem. In J. Weckert and Y. Al-Saggaf, editors, *Selected Papers from the Computers and Philosophy Conference (CAP2003)*, volume 37, pages 21–25, Canberra, Australia, 2004.

[130] D. Traum. *A Computational Theory of Grounding in Natural Language Conversation.* PhD thesis, Computer Science Dept., U. Rochester, December 1994.

[131] A. M. Turing. Computing machinery and intelligence. *Mind*, 59:433–460, 1950.

[132] P. Vogt. *Lexicon Grounding on Mobile Robots.* PhD thesis, Vrije Universiteit Brussel, 2000.

[133] P. Vogt. The physical symbol grounding problem. *Cognitive Systems Research Journal*, 3(3):429–457, 2002.

[134] P. Vogt. *Artificial Cognition Systems*, chapter Language evolution and robotics: Issues in symbol grounding and language acquisition. 2006.

[135] P. Vogt and F. Divina. Social symbol grounding and language evolution. *Interaction Studies*, 8(1):31–52, 2007.

[136] G. Wasson. *The Design of Representation Systems for Autonomous Agents.* PhD thesis, Computer Science Department, University of Virginia, August 1999.

[137] G. Weiß. Agent orientation in software engineering. *Knowl. Eng. Rev.*, 16(4):349–373, 2001.

[138] J. Weng. Developmental robotics: Theory and experiments. *International Journal of Humanoid Robotics*, 1(2):199–236, 2004.

[139] J. Weng and W. S. Hwang. From neural networks to the brain: Autonomous mental development. *IEEE Computational Intelligence Magazine*, 1(3):15–31, August 2006.

[140] J. Weng, J. McClelland, A. Pentland, O. Sporns, I. Stockman, M. Sur, and E. Thelen. Autonomous mental development by robots and animals. *Science*, 291(5504):599–600, Jan 2000.

[141] S. Wermter, C. Weber, M. Elshaw, V. Gallese, and F. Pulvermller. *Biomimetic Neural Learning for Intelligent Robots*, chapter Grounding Neural Robot Language in Action, pages 162–181. Lecture Notes in Computer Science. Springer, Berlin, 2005.

[142] S. Whitehead and D. Ballard. Learning to perceive and act by trial and error. *Machine Learning*, 7(1):45–83, July 1991.

[143] M-A. Williams, J. McCarthy, P. Gärdenfors, A. Karol, and C. Stanton. A framework for evaluating groundedness of representations in systems: from brains in vats to mobile robots. In *IJCAI 2005*, 2005.

[144] R. Zhao and W. Grosky. *Distributed Multimedia Databases: Techniques and Applications*, chapter Bridging the Semantic Gap in Image Retrieval, pages 14–36. Idea Group Publishing, Hershey, Pennsylvania, 2001.

[145] T. Ziemke. *Understanding Representation in the Cognitive Sciences*, chapter Rethinking Grounding, pages 177–190. Plenum Press, New York, 1999.