Design of Representation Systems for Autonomous Agents

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

at the

University of Virginia

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy (Computer Science)

by

Glenn Scott Wasson

APPROVAL SHEET

This dissertation is submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy (Computer Science)

_____

Glenn S. Wasson

This dissertation has been read and approved by the Examining Committee:

_____

Worthy N. Martin (Thesis Advisor)

_____

Randy Pausch (Committee Chairman)

_____

James C. French

_____

Gabe Robins

_____

Bennett I. Bertenthal

_____

David M. Kortenkamp

Accepted for the School of Engineering and Applied Science:

_____

Dean Richard W. Miksad

School of Engineering and Applied Science

June 1999

Abstract

This work proposes a design methodology for creating representation systems for robots that operate in dynamic, uncertain domains. In order to be efficient and effective in these domains, robot control must be achieved through a tight coupling of sensors and effectors and this constrains the design of representation.

My methodology attempts to guide the designer in creating representations that are effective for robots using these control systems. The methodology assists the designer in analyzing the robot's task, capabilities and environment to answer the questions of what to represent, how to structure that representation and how to keep that representation consistent with a changing environment.

The efficacy and validity of the methodology is documented through several agent designs that were successfully carried through to implementation. The key feature of these agents is the action-oriented portion of their architectures and the systems of representation they employ.

Acknowledgments

There are a large number of people who have contributed to this work in different ways and it would be hard to thank them all. So, a heartfelt thank you to all of them (you know who you are) and special thanks to the following people.

First, thanks to my parents and my sister for expressing a continual belief that I would (eventually) finish. Thanks to all the various student members of the vision group that have come through in my tenure here. Most particularly to Frank Brill, Gabe Ferrer and Jim Gunderson for helping to start the Robotics Group. My advisor, Worthy Martin provided invaluable assistance in leading me by what, at times, seemed to be an inscrutable master plan, but did, in fact, get me to this point. I also need to thank Tom Olsen, whose advising in the early days kept me interested enough to stay on for a Ph.D.

Finally, the road through graduate school is seldom travelled alone and I thank the cast of characters that is sigbeer for being (perhaps too) willing to provide diversions from drudgery. Also, I thank Aaron Cass, Anand Natrajan and John Jones for, at various times, providing the means and the mechanisms for outstanding occasions.

Table of Contents

List of Figures

List of Tables

Chapter 1 *Introduction*

A fundamental question for the designers of physically embodied AI systems is how those systems should decide what to do. The answer to this question is affected by the choices of how often new decisions are made and what information is considered when making them. These choices, in turn, put constraints on the types of computation used in the decision making process. This thesis addresses the question of what information to consider by providing a methodology for developing symbolic models of the environment that are efficient and effective for use by agents that must quickly and continually make decisions in dynamic environments.

1.1. Changing Views on Action Selection

"Action selection" refers to an agent's procedure for deciding what to do, i.e. choosing actions. Over time, many action selection mechanisms have been designed, some of which have lead to a rethinking of what it means for an agent to "decide what to do". In the 1970s, action selection was done via deliberative thinking [58], i.e. planning. Planning involves using a symbolic model of the agent's goals, capabilities and environment to generate a series of actions that meet the agent's goals. This plan is then carried out by an executive that simply performs the actions and (therefore) achieves the goals. While it is possible to generate optimal plans with this approach, planning systems were fairly unsuccessful in dynamic, real-world domains because they made unrealistic assumptions about their models

of the environment. The information considered in the action selection process, i.e. the world model, was assumed to be complete, accurate and static (except for changes made through the agent's own actions). Neither the speed with which these agents generated plans, nor the frequency with which they undertook this effort effected the correctness of their selected actions. This was fortunate because inferencing over a collection of facts (the world model) proved to be computationally complex [20] and slow in practice.

In the mid 1980's, Brooks [17] challenged both the world model assumptions of planners and the sense-plan-act model of robot architectures (e.g. [58]) with his subsumption architecture. In this architecture, there is no internal model of the world and thus there can be no time consuming deliberation process. This changes the notion of action selection from deciding upon a series of actions, to making a momentary decision on what to do *now*. Since the agent stores no information about the world, it can only decide what to do based on its current sensor values. It doesn't make sense to select a series of actions to take, because there is no information about past or future world states on which to base that selection. Instead, "actions" taken in the subsumption architecture are simple, short-lived effector commands that are executed whenever particular patterns are detected on the agent's sensors. Each action takes very little time to execute and so the agent operates in a tight loop, selecting and executing actions based on current sensor data.

The action selection mechanisms of these "reactive" agents differed from those of their predecessors because they decided what to do much more frequently and they considered far less information when doing it. Since these agents stored no world model, they could only use information about the part of the environment within their sensor range at a particular moment. This approach has the advantage that, since an agent continually senses the

world and re-decides what action to take, it can quickly detect and respond to changes in a dynamic environment. However, because no information about the world is stored, the agents are fairly myopic. The agents proved difficult to design (except for simple tasks) because all information relevant to determining the next effector command cannot always be determined by the agent's sensors at any time.

The success of subsumption and similar reactive systems [17][48][50] pointed up the problems with planning's world model assumptions when operating in dynamic domains. An agent's knowledge of the world is rarely complete and the world changes due to factors beyond the agent's control. Realization of inherent problems with incomplete knowledge, e.g. the frame problem [51], and Brooks' assertion that human intelligence could be achieved without representation [15], lead to the conclusion that an internal representation of the world impeded action selection in these reactive agents. Not only would the complexity of a useful world model, it was said, make it expensive to build, it would be expensive to sort through the data to decide what to do. By the time the agent selected an action, the world would inevitably have changed and so the selected action would be wrong. By avoiding the temptation to have a representation of the world, the design could avoid moving back to the sense-plan-act paradigm.

This thinking concluded that representation, in and of itself, was an obstacle to performance. Brill [14] has shown that this need not be the case. Representation does not inevitably lead to planning and "reactive style" action selection mechanisms can benefit from representation. The key to have a representation that the agent does not need to spend a long time searching and that does not require the agent to use elaborate inferencing (planning) to deduce the correct action. This thesis concentrates on designing such systems of repre-

sentation for autonomous agents operating in dynamic domains.

1.2. Representation Design

In this thesis, I investigate the *design* of *representation systems* for the *action oriented* portions of *agent architectures* and put forward a design methodology. My thesis is:

> This methodology can be used to produce systems of representation, for the action oriented components of an agent's architecture, that are efficient and effective for use in dynamic domains.

In the remainder of this section, I give basic definitions of these terms that will be elaborated in later chapters.

1.2.1 Design

This thesis proposes a form of a task-oriented design for creating representation for agent architectures. Task oriented means that the representations will be specialized to the task(s) and capabilities of the agent under consideration (this is similar to the general software design process discussed in [86]). The tasks of interest to this thesis (see section 1.3) take place in dynamic environments and so the agents are assumed to be selecting actions in the rapid sense-act cycle typical of reactive systems. Of course, many agent architectures today have multiple "action" selection mechanisms that decide on actions at various granularities, e.g. one system may decide that the agent's action should be to "drive to the store", while another system would decide on the particular motor commands to steer the car through unpredictable traffic. Section 1.2.4 discusses the types of action selection mechanisms that are applicable to this work.

An agent's capabilities strongly influence the design process. Capabilities are anything the designer is given to work with at the start of the design process. They may include hard-

ware (what sensors are available? how accurate are they? are there wheels? wings?) and software (sensor processing routines, device drivers, manufacturer provided libraries, etc.). Capabilities define the designer's starting point and determine both the basic actions that the agent is capable of and the entities that can be represented (and thus perceived and manipulated). This methodology has the designer examine tasks and capabilities to create representation that allows agents to be more efficient and effective at their tasks than they would be without representation.

1.2.2 Representation Systems

By "representation", I mean data structures that hold some information about the agent's environment. A representation system, then, is a collection of these data structures describing all or part of the environment in sufficient detail that the agent can *effectively* and *efficiently* complete its tasks. In chapter 3, the reader will see that much effort is devoted to representing what is required to make the agent effective, while not representing too much so that the agent can remain efficient.

It is important to distinguish between the "explicit" representation considered by this thesis and the "implicit" representation used by many reactive agents. Explicit representation refers to actual memory structures that store properties of the world. "Implicit" representation is often referred to by the catch-phrase "the world is its own best model" [16]. In other words, no properties of the world need to be stored in memory because if the agent needs to know them, it merely needs to observe the world and compute them. In a dynamic, physical environment with real sensors, implicit representation can be computationally expensive and/or impractical since not all of the environment can be perceived at any one time (see section 1.3.1). Hereafter, the word representation refers to the explicit variety.

1.2.3 Representation of Objects

If representation describes the agent's environment, what about the environment is represented? Fundamentally, I believe objects should be represented. Each unit of representation should store data about an object in the environment. An object is any "suitably distinct" physical entity[1] that the agent can act upon (or act with respect to, or use, or manipulate, etc.). Holding off on the definition of "suitably distinct" for the moment, objects need to be what is represented because actions take place on (or with respect to) objects. The purpose of a system of representation is to assist the agent in action selection and so data that is important for action, i.e. objects, should be represented.

At some level, an object is defined by the actions that can be taken on it. A chair is something that I can sit on and any object that I can sit on can be said to be a chair. The actions that can be taken on an object will also depend on the acting agent. Different agents will have different capabilities and thus different possible actions. For example, a robot that can plug itself into the wall would be built to understand that a wall socket is an object, while a robot that does not have this capability may treat the socket as part of the wall.

This gets back to the notion of "suitably distinct". An object must be distinct from its surroundings, so for example, arbitrary points in space, or indistinguishable portions of the carpet can not be treated as objects. What can be (or will be) distinguished will obviously depend on the agent's perceptual capabilities, but also on the agent's task. For example, if an agent wanted to write a note, it would need a writing implement. If the agent had perceptual routines to detect pencils, a pencil could be used as the writing implement. The agent could act on (write with) the pencil because it had the perceptual capabilities to treat

_____

1. I say physical entity to clarify that I mean corporeal objects and not programming constructs.

6

the pencil as a distinct object. Now suppose the agent makes a mistake and wants to erase something. It could use the eraser on the end of the pencil. Previously, the eraser had not been a separate object from the pencil because it was not involved in the writing action. Even though the pencil detection routine may have needed the eraser to be present as part of the sensory signature of the pencil, the agent need not give the eraser its own representation unless it needs to erase. In other words, the task determines what constitutes an object.

At this point, it is worth mentioning that an object-based representation system is a different kind of representation system than those used by many other robots, e.g. [42][53][55]. These robots have a representation of space (usually "free space"), but no notion of how the occupied space segments into important objects.

1.2.4 Action Oriented Architecture

Stateless, reactive agents, while robust at certain tasks, proved difficult to design. The lack of representation makes it difficult to scale such systems because the agent's current state can not be used to reduce the complexity of examining the environment for a large number of perceptual stimuli [75]. Also, while planning may be inefficient for action selection in certain tasks, it is useful when time permits the consideration of alternate courses of action and the cost of taking the wrong action is high.

In response to the limitations of reactive systems, many hybrid architectures were developed [10][29][69][71][3][5][22]. These hybrid architectures combine a traditional planner with an executive. The executive may itself be divided into multiple components, all of which I refer to as the "action oriented" portion of the architecture because they control the agent's interaction with the world. Although all parts of an architecture exert some control

on the agent's actions, the action oriented portion is the portion for which the timeliness of deciding "what to do" is most critical. That is, any portion of the architecture that operates in a tight "action selection loop" (in the sense-act paradigm) is in the action oriented portion of the architecture. This leaves out any portion of the architecture that needs to do inference, such as a planner.

The representation systems that result from the use of the methodology in this thesis are specifically targeted toward this portion of the agent architecture because of the speed with which it must react to events in the world. This speed is what makes useful representation for this portion of the architecture different from representation used by planners. In order to be efficient and effective, these representation systems must provide useful information to the agent without providing too much information. The agent must update its representation to correspond to the current state of the world and consult that representation to determine its next action within a certain time frame. That time frame must be such that the representation corresponds closely enough to the changing environment that the selected action is appropriate.

1.3. Applicability of Designs

The representation systems developed by this methodology are useful for agents operating in domains possessing the characteristics described in section 1.3.1. These agents also have fundamental similarity in the action oriented portions of their architectures, as described in section 1.3.2. Finally, perception in an important agent capability and its impact on representation design is discussed is section 1.3.3.

1.3.1 Domain Characteristics

There are a number of characteristics that make an agent's domain, i.e. task, capabilities and environment, amenable to the representation systems designed by this methodology. These characteristics are:

1.3.1.1 The Environment is Dynamic

A dynamic environment is one in which properties of the environment that are important to the agent's task change over time. This means that current sensor data is more valid that previous sensor data.

1.3.1.2 The Task Contains Different Aspects

Different aspects means that different portions of the agent's task can be handled by different portions of the agent's architecture. When the task requires rapid interaction with the environment, the agent can handle it with one mechanism, but when consideration of alternatives or sequencing of actions must be performed, other mechanisms are available. This means that the task must involve different "types" of computation. In must not focus entirely on real-time performance, but must provide some reward for deliberative thinking. However, since the environment is dynamic, the agent cannot reduce its reaction time too much with planning.

1.3.1.3 Knowledge of Large-Scale Space is Required

Large-scale space is an environment in which "spatial structure is at a significantly larger scale than the sensory horizon of the observer" [42]. An agent in such an environment will, at some time, not be able to perceive all important aspects of the environment. This agent can benefit from some form of memory (i.e. representation) to remember important, previ-

ously perceived data. Memory can also be used to hold expectations about what the environment will be like. These may come from long-ago explorations or from a human operator. Note that large-scale space is not disjoint from what this work refers to as local-space. Local-space is the area about which the agent's sensors and the representation in the action-oriented portion of its architecture currently give it information. The existence of large-scale space means there will be multiple local-spaces.

1.3.1.4 The Task Can be Decomposed

This means that the agent's task can be decomposed into a hierarchy of tasks and subtasks where the agent's overall goals are achieved by tasks at the top and the agents capabilities form the basis of the tasks at the bottom [86]. Of course, any task can be decomposed into subtasks, but any decomposition past the given capabilities is not of interest (the fact that a given library may translate a "move forward" command into motor voltages is beneath the concern of the designer because the library abstracts that away). Different tasks should view the world at different levels of abstraction. "Part and whole" is a common abstraction where the agent will treat some object as "a whole" in one task, but as a collection of parts in another task. For example, a task whose goal is to pick up a pot from the stove may represent the pot as a single entity. However, if the task's decomposition (and thus the agent's capabilities) caused the agent to actually pick up the pot by the handles, individual handles may be represented in the subtasks.

1.3.1.5 The Entire Plan Cannot be Made A Priori (or Replans Occur)

This characteristic means that the agent does not have complete knowledge of its environment and so classical planning is ineffective. This is not to say that the agent has no knowledge and must obtain all information through exploration. In fact, since the environ-

ment is dynamic, it is likely that the agent will not be able to generate a complete plan that can be blindly executed, even with complete knowledge, because the world state will change from when the plan was generated. If sufficiently abstract plans [66] can be crafted for the domain, the environment must be able to change such that replanning needs to occur. Note that while this thesis is not concerned with planning, either in how plans are generated, or in the organization of the planner's representation, it is concerned with the use of plans. The domain's environment will be sufficiently complex that it will be difficult for a traditional state-space planner (with its previously discussed world model assumptions) to operate effectively. This means some intelligence must be embedded in the action oriented portion of the architecture, i.e. it needs action selection machinery of its own, to execute the plans.

1.3.1.6 The Environment is Suitably Quantifiable with Coarse Metrics

This means that crude maps of the environment are effective for the agent's tasks. While all environments are quantifiable to some degree, the important idea is that the agent does not need precise metric information about its environment to get around. The agent's sensors can make up for inaccurate map information during task execution.

1.3.2 Agent Architecture

The methodology is designed for agents that have perception/action (PA) systems. Perception/action systems operate in a tight-loop, coupling sensation to action. These control loops are effective because they avoid a lengthy action selection process and I believe they are practically necessary for agents operating in dynamic environments. Simple transforms of the agent's sensor values select among possible actions. An action itself is not an unin-

terruptible process, as the agent is continually deciding whether to continue the current course of action or switch to a new one. The action of picking up an object may involve sending dozens of control commands to the effector motors over time. As long as the sensors indicate that picking up the object is what the agent should be doing, it will continue that action.

I assume (as do others, e.g. [5][10][29][50]) that a perception/action system is made up of small control programs, often called skills [10] or behaviors [50]. These programs watch the sensors and respond when appropriate. Often multiple skills will be active at the same time and agent's overall behavior is the combination of their output. Skills define "situated" control rules for the agent to accomplish some goal. Situated means that the skills expect to be operating in some context and interpret the agent's sensor data accordingly. Recall the agent with the goal of plugging itself into a wall socket. This agent's skill to plug itself in may interpret a short sonar reading as the approaching wall with the socket. A different skill navigating the agent across the room may interpret the same sonar reading as an obstacle to be avoided.

Skills in this thesis are similar to skills in traditional reactive agents, except they are too complex to be purely stateless and so they use representation. These skills can be thought of as sets of (situation, action) rules, where "situation" is some world state specified by both current sensor values and data stored in the skill's representation and "action" is the effector control command(s) to execute. The skill continually samples the environment to determine which "situation" is currently applicable and executes its "action".

The representation of these skill are "variables" that are associated with objects in the environment at run time [7], such as a juggling skill that has a number of ball variables (rep-

resentation) that must be associated with some juggle-able objects. Making this association is referred to as "binding" the representation. It doesn't matter if the objects are rubber balls or baseballs, as long as the agent can juggle them. The PA system's skills then use the current sensor values *and* the information contained in the representation as the context in which they select their actions.

An important aspect of skills is that they are not designed to be fool-proof, but rather to recognize when they fail. This is referred to as cognizant failure [27] and it allows a skill to contain a small set of rules for interaction with the environment. Rather than try and handle all exceptional cases (a seemingly impossible task), the skill merely reports that it has failed and another skill (or portion of the architecture) tries to correct the problem. The principles of cognizant failure and situated-ness allow skills to operate extremely fast since they need not deal with the full complexity of the world. Instead they can make simplifying assumptions about the world (to streamline their actions) and if the world moves outside of those assumptions, they merely need to recognize this fact and report failure.

Even though this design methodology is concerned with agents that have PA systems, the domain characteristics show that the agents must have other components to their architectures as well. Specifically, this methodology is appropriate for agents with layered architectures. Some layers don't deal directly with sensors and effectors, but are still in the action oriented portion of the architecture because they do no planning. The agents developed in chapters 4 - 6 have three layered architectures, as do many current agent architectures, e.g. [10][29]. However, this thesis is not concerned with architectural design (except in that the agents developed by this methodology must have an architecture and so I do discuss how they were designed), but rather with the design of systems of representation to be

used by an architecture. In this sense, any architecture that interacts with the world through a PA system can be used.

In this thesis, the PA layer is referred to as the "lowest" layer because it is directly connected to the actual environment. Other "higher" layers are connected only to the layers above and below them. Typically, the "highest" layer is some sort of planner. It is worth noting that most other existing robot architectures have two or three layers (PA layer and planner or PA layer, sequencer and planner). There has been no compelling case given in the literature for more layers. Some architectures have used more layers, [71][3][26] for example, but these have either never been implemented for any real-world task that makes use of these extra layers [71][26], divide the PA layer tasks into extra layers [3], or are not designed to control mobile robots [3].

1.3.3 Agent Perception

Representation of a dynamic environment necessarily means updating the representation as the world changes. This makes perception an important design consideration for any system of representation. This thesis is mainly concerned with vision-based agents, i.e. agent that derive most of their information about the world from vision. The issues explored in the design methodology are applicable to all sensor modalities, but are discussed in terms of vision because vision presents more computational problems than other sensors commonly used by robots.

1.4. Overview

This section provides an overview of the representation design methodology and the remainder of this thesis. The methodology is designed to create hierarchical systems of rep-

resentation to be used by agents acting in dynamic, uncertain domains. The methodology itself is organized as a series of questions for the designer. Each question is designed to highlight a certain aspect of agent design that has implications for the representations created. The methodology is not an algorithm and so there is still some art involved in agent creation. This is particularly true with regard to the methodology's first step, decomposing the agent's task into subtasks (as with any applicable software engineering technique, e.g. [11][84]). The methodology also advocates an iterative design process, where the designer modifies their answers to previous methodology questions, if those answers are found to produce "unworkable" representation systems. "Unworkable" means that the agent can not be made to operate efficiently and/or effectively in its domain. This point could be reached during the design phase or when building a prototype (chapters 4 - 6 contain several examples).

Once a task decomposition has been created, the individual tasks themselves are analyzed to determine what information about the environment is important and how that information can be extracted from the environment. This information can be stored in the agent's representation, but this involves a trade-off between the usefulness of the information and the cost of verifying it in a dynamic world. The methodology advocates making this trade-off based on how that information is used, how easy it is to compute that information and how many skills/behaviors use that information.

After the methodology has been laid out, three different agent designs are presented in chapters 4, 5 and 6. Each of these agents performs a different task and uses one or more systems of representation designed by the methodology of chapter 3.

Evaluating any design methodology is difficult and in this case even a comparative anal-

ysis is hard because few other methodologies address the design of representation systems for the action oriented portion of an agent architecture. In chapter 7, I discuss how some popular architecture design methodologies might address the task of the agent in chapter 6. I develop 5 criteria to evaluate the efficiency and effectiveness of representation systems and show how the other design methodologies can fail to create representation systems that meet all or most of those criteria.

The main contribution of this thesis is a structured approach to design of representation for autonomous robots. This and other contributions and conclusions are discussed in chapter 8. Today, there is no debate about whether a robot should be totally reactive, or totally deliberative, it needs elements of both. Similarly, there is no debate over whether robots should have representation or not, representation can be useful though it comes with some liabilities. This thesis explores how representations can be made both efficient and effective by extending reactive systems to use a small amount of task dependent representation and addressing the issues of what to represent, what structure that representation should take, and how the data stored in that representations can be kept consistent with changes in a dynamic world.

Chapter 2   *Related Work*

In this section I analyze previous work in both agent representation systems and agent design methodologies. Few agents are designed without some methodological guidance. However, most current methodologies do not explicitly address the problem of representation design and so the representation systems of the corresponding implemented agents themselves must be examined.

2.1. Perception and Representation

The main goal of this work is the design of representations of the environment that are effective for the action oriented portion of an agent architecture. Action in complex, dynamic worlds relies a great deal on perception to provide timely information about the environment. As such, an agent's representation must be closely tied to its perception. In other words, representation must be continually modified based on perceptual information. Many researchers have developed agents that use representation in their perceptual system.

The Pengi system of Agre and Chapman [2] plays a video game (Pengo) in which the simulated agent (a penguin) does battle with some enemies (bees) in a rectilinear maze of ice blocks. The agent uses data structures known as markers [76]. These structures are associated with particular elements of the video game that are currently of importance to the agent (when running away from a bee, you need to look ahead for the-block-that-is-blocking-my-route to determine if you can escape). These markers are similar to the representa-

tion used by the agents of chapters 4 - 6 in that they store both the position and task function of objects in the environment. Task function is important because the same ice block could be the-block-that-is-blocking-my-route in one task and the-projectile-to-kick-at-the-bee-near-me in another. Later, Chapman's Sonja system [19] played a more complex video game with fewer constraints on the agent's actions.

In both Pengi and Sonja, markers are used to bridge the gap between early vision and action. Markers hold locations of important entities in the environment and based on spatial locations of markers, the agent can determine its next action. However, these markers are only a limited form of representation because they only hold locations within the agent's two-dimensional omniscient (overhead) view. No memory is used to store important locations outside the current view or to allow markers to persist between perception/action cycles. In addition, neither system addresses the perceptual problems of early vision. While Sonja does compute early visual properties to determine locations for markers (Pengi directly accesses the video game's internal data structures), it does not handle problems with physical sensors, such as a limited field of view, first-person perspective, occlusion and the unconstrained nature of the vision problem.

Kuniyoshi et. al. [45] have developed an agent that also uses data structures, which they call markers, to hold positions of task-dependent points in space. This agent operates in the real world and uses stereo and optical flow to update positions of markers. However, in [65], Riekki and Kuniyoshi state that markers must be associated with perceivable features and so they cannot move outside the field of view. In addition, a single marker in this system can store positions of multiple points in space (markers can specify paths for example) and as we'll discuss in chapter 7, this can make them difficult to update.

Brill [14] has created a simulated agent that operates in a 3-D, first person domain. This agent represents important task dependent objects in its environment with markers that are similar to those used by Pengi. These markers can, however, represent information outside the current field of view. This expanded area, called the "effective field of view" [13], can be used by a set of behaviors to choose the agent's next action. The term "effective field of view" denotes the area of the environment that the agent can access in its decision making process. It includes the area currently within range of the agent's sensors and certain, recent, task-dependent percepts that are stored in the agent's internal representation. This stored data is treated as if it were another sensor and so the agent can continue to operate in a "reactive" fashion, even though it may be selecting actions based on stored data [13]. Brill's agent uses a system of representation that is similar to portions of the ones used by the agents in chapters 4 - 6. However, I have expanded the concept of markers to make them effective for inter-layer communication by defining the concept of marker instantiation and by separating concepts that were entwined in Brill's system. For example, the perceptual properties of objects associated with Brill's markers were determined by the object's role in the task (so all food was red and all red objects were food). Brill's work addressed the issue of whether representation could improve the performance of a perception/action system while this work addresses the issue of the design of representation systems.

Horswill [32][33] has created agents that associate unary predicates with visual trackers (to determine the truth value of some predicate, look at the position of the tracker). These trackers can be inside or outside of the field of view (and are updated correspondingly by odometry or vision). However, while his agents avoid the scaling problems inherent in an Agre and Chapman like system (see section 2.6) by limiting the number of trackers,

Horswill provides no theory or methodology for the design of a system to control their allocation. In chapter 3, I discuss the design of multi-layered representation systems with this goal in mind.

Ballard [8] discusses agents with animated (foveated) vision systems. These agents store the coordinates of important objects that have previously been foveated by their visual systems. Ballard advocates storing the coordinates of these objects in object-centered coordinates, that is, relative to other objects. In this way, the agent only needs to maintain a transform from its current foveal position to *an* object since the stored object-to-object transforms can be used to transfer the fovea to any other important entity. While this may be more efficient in some cases than the ego-centric coordinate system proposed in chapter 3 (since every ego-centric coordinate must be updated when the agent moves), Ballard's approach is fragile if objects can move relative to each other. This thesis assumes that agents designed by this methodology will be operating in dynamic environments where the object-centered approach will be too brittle and so advocates paying the computational price for an ego-centric representation.

2.2. Representations for Navigation

Maps of an agent's environment are a common form of spatial representation. Since many tasks for autonomous robots (including those in this thesis) require navigation, maps are quite useful. Grid based maps, called occupancy or certainty grids, have been implemented using sonar [12] [55] and stereo vision [54]. These representations describe the free-space surrounding the robot and so are most suited for navigation and obstacle avoidance. Since these techniques register only occupancy, their representation is not suitable for a general representation of important properties of objects in the environment. Kuipers

[42][43][44] has developed maps for indoor and outdoor "large-scale" environments. His TOUR model defines a semantic hierachy of information where <view, action> pairs are accumulated to form topological and then metric maps. A <view, action> pair means that doing "action" should put the agent in a position where its sensors perceive "view".

Both occupancy grids and the TOUR model's maps (indeed all map representations), represents space and not objects. This means that computational effort must be expended to "extract" object data from the map. Typical tasks for autonomous agents, including navigation, are with respect to some object (e.g. "pick up the book", "go to the store", "hit the ball") and therefore will need object data. A perception/action system should not use map representation as, for any large map, it will be prohibitively expensive to extract object information. However, other layers of the architecture can (and the agents in this thesis do) use maps. Kuipers work is complementary to mine in that even though maps are an inappropriate representation for a PA layer to use in choosing actions, the PA layer can be used to build a map using the TOUR model. The <view, action> pair are the right "granularity" for a PA layer to generate because they correspond to the phases of the PA layer's execution loop, i.e. perception (view) and action. The synthesis process that builds the maps could be performed by another layer of the architecture that receives these tuples from the PA layer.

Other researchers have also used map representations. Kosaka and Kak [41] have developed an agent that navigates using CAD models of its environment, visually matching model features to perceived features. CAD representations are difficult to obtain (someone has to create the model) and brittle. The brittleness is due to the fact that CAD models model the static properties of the environment, those portions that are thought not to change. However, the agents of interest here exist in domains where dynamic (not static) features

of the environment are important. Miller [53] presents a mapping representation that is based on regions. Each region defines the number of degrees of freedom about which the robot can determine its position based on local sensory information. In an open area, the robot can determine nothing about its position or orientation, while in a dense area of objects, the robot may be able to totally orient itself. Mataric [50] uses a topological map built as a distributed collection of "behaviors". Activation energy spreads outward from the node at which the robot is located, activating nearby behavior-nodes to begin looking for their associated sensory signatures in the sonar input. The main advantage of her technique is that the map actively maintains itself, adding or removing links between nodes as the agent travels or fails to travel between perceptually distinct areas of the environment. However, these maps are non-metric and so cannot always be used to make efficient path plans (just topologically short ones).

All of these representations are useful for self-localization, i.e. determining one's position with respect to a map, but are not a good representation for general tasks. This thesis provides a means of breaking down large-scale spatial knowledge, such as that contained in maps, and making it available to the PA layer as computationally affordable chunks of local-space information.

2.3. Agent Architecture Design Methodologies

There are many agent design methodologies in existence today. Most are concerned with dividing the agent's tasks among various components of the agent's architecture based on the nature of the tasks. Although most scarcely address the design of representation systems (Brooks [17] being the exception), they do each advocate the use of one or more representation systems.

Early work in the design of agent architectures for dynamic environments was done by Brooks [15][16][17]. He proposed eliminating all representation because it would invariably become inconsistent with the state of the world and therefore lead the agent to make incorrect decisions. While this approach has been rejected as heavy-handed by most researchers (including the Representation subgroup of the 1995 AAAI Spring Symposium: "Lessons Learned from Implemented Software Architectures for Physical Agents"), it did cause a re-evaluation of the role and structure of representation in the perception/action components of agent architectures [14]. Brooks himself used internal state, i.e. representation, in most of his robots [17].

The RAP system [28] has been used in a number of autonomous agents with multi-tiered architectures, most often as part of the 3T architecture [10]. RAPs use lisp-style predicate structures to describe the state of the world. While these have certain advantages (for systems written in lisp) such as the ability to perform unification and their ability to be used as a fact database for a classical planning system, they are fairly unstructured for perception/action systems. Unstructured means there is no limit on the number or content of these predicates. The RAP system was not designed to operate in tight perception/action loops with the environment and so it is not surprising that its representation system is inefficient for such purposes. The original RAP work [27] used a simulated perception system that wrote time-stamped sensor predicates to RAP memory. This required the RAP system to search through its list of facts to determine which were current and required some form of garbage collection. Such searches present an unacceptable time sink for agents in dynamic environments and are the result of trying to make a PA system operate like a planner.

The 3T architecture [10] uses a layer below the RAP system called the skill layer. This

layer sends "event" messages to the RAP system over a low-bandwidth channel, allowing RAP memory to contain smaller numbers of "higher level" facts. For example, when pouring a cup of coffee, instead of storing visual state used to accurately track the coffee pot while aligning it with the cup, the RAP memory can simply hold the fact that the pot "is aligned" with the cup (allowing the RAP engine to tell the skill layer to begin the pouring task). While the accumulation of perception and action information into events reduces the number of predicates and hence the search process of the RAP layer, this dissertation argues that any representation system can be made more efficient if it can be given appropriate structure. Chapter 6 describes an agent that uses structured representation to encode "higher level" properties of the world (such as those stored by the RAP system) without the need to perform unification to derive them from a database of information.

It should be noted, however, that this work does not dismiss the RAP system's sequencing capabilities or processing engine. In fact, the representation design methodology presented here is complementary to such a system, since RAPs do not address the design of representation and this work does not address the structuring of agent processing capabilities[1].

As mentioned previously, the 3T architecture has at its lowest layer, a collection of independent, parallel behaviors called skills. These skills are typically arbitrary C code and can pass arbitrary data structures between themselves and the RAP system. The 3T design philosophy is concerned with deciding which tasks should be handled by which layers of the agent architecture. The design of the processes that carry out those tasks is left to the

---

1. I do address the structure of an agent architecture in that I believe it should be multi-tiered, as with many other architectures [10][29][5][71]. I also designed sequencing layers for the agents in chapters 4 and 6, but I do not assert that my sequencing layers are superior to the RAP system, merely that they perform similar functions.

agent programmer. This thesis' methodology proposes that arbitrary data structures are the wrong representation for PA systems because they can force the PA system to do unacceptable amounts of work to sort through the represented data. This slows the PA system's decision making process. In addition, arbitrary amounts of data can take arbitrary amounts of time to verify. The PA system can not check if arbitrary representation is consistent with the world state and remain responsive.

Gat's ATLANTIS [29] architecture has representation in all its layers. Skills in ATLANTIS's perception/action layer are written in a language called ALFA [30], which allows internal state variables. These variables, however, are simple data types (integers, floating points, etc.) used to implement loops and such, and are not effective to model the environment. Of course, such data types can be used to encode information about the environment, but since neither ALFA nor ATLANTIS place any limit or structure on the representation, the representation has the same deficiency as the arbitrary data structure representation allowed by 3T. Gat's sequencer layer, which sits above the PA layer, is modeled after Firby's RAP system and so uses similar memory structures (with the same benefits and limitations).

An alternative architectural layout is proposed by Simmon's TCA architecture [69]. His work is concerned with interleaving planning, sensing and execution through TCA's plan representation, a fundamentally different type of representation than that addressed by this thesis. Although TCA's modules appear to hold internal state, that state's construction is left to the module designer and thus suffers from the potential for arbitrary data structures.

The architecture of Noreils and Chatila [59] is somewhat similar to 3T in its division of tasks between components of the architecture. While they also allow arbitrary data struc-

tures, they do use a technique similar to my representation instantiation (see chapters 3 through 6). Beliefs about the world are expressed to sensory modules that then try and fit world data to them. The Supervenience architecture [71] has many layers, each using the LISP clause as the fundamental "unit" of information. The layers share a common memory system and have the same problem as Firby's RAP memory, i.e. it becomes expensive to both find and "forget" (throw away) information among all the predicates.

Albus's RCS architecture developed at NIST [3] was designed to control many types of mechanical devices, not just mobile robots. It defines a general representation structure used by all levels of an architecture in which higher layers have progressively more powerful inferencing capabilities. Unfortunately, this allows planners and PA layers to use the same representation structures. I believe that the techniques available to planners allow them to use representational structures that are ineffective for perception/action systems. I recommend creating separate (but linked) representations of the world for planners and PA layers to avoid the temptation of using inference in the PA layer itself.

Arkin's AuRA [5] architecture separates PA layer and planner functionality, but is mainly concerned with navigation at the PA layer. Its representation is geared toward potential fields representing travel paths for the agent. Connell's SSS architecture [22] has only limited representation of the environment's geometry in its symbolic layer and none in its subsumption or servo layers. This architecture has only been applied to navigation and so the lack of representation was reasonable. As stated previously, I believe stateless systems are not reasonable for general tasks.

CIRCA [56] is an architecture concerned with making real-time guarantees about its performance. The authors do not address the design of efficient representation systems, but

much like the RAP system, there is no reason why a well designed representation could not be used in this architecture.

Certain extensions of the Soar architecture, such as Robo-Soar [46] or Air-Soar [62], represent cognitive architectures that control agents that interact with the world. Soar is a cognitive architecture that can pursue a number of hierarchical goals simultaneously. The key feature of the Soar architecture that makes it different than other multi-tiered architectures is that each layer in Soar has the same control structure, it just operates in a different problem space [46]. Since I am interested in representation design, it is beyond the scope of this thesis to address the advantages and disadvantages of this architectural structure vs. other layered systems. Instead, I address the use of representation in this architecture.

Soar has a two level memory system, where "working memory" contain knowledge about the agent's environment and "long term memory" contains productions that contain the control information that the agent uses to select and reason about actions. This thesis is interested in designing representation whose function is akin to "working memory". There are a number of problems with the memory system in these versions of Soar, such as LISP-predicate style representation and a uniform representation at all layers. These are discussed in more detail in chapter 7.

Uniform representation is used at all layers in Soar because all layers are essentially production systems. The problem is that this means that the information used for long-term planning and reactive in a dynamic environment is contained in the same structure. This can tempt a designer to store information that is useful for planning in a layer concerned with reacting. Without trying to constrain what is contained in the representation of the layers of the Soar systems that interact with the world, the Soar methodology allows these lay-

ers to bog down trying to search through and maintain that information.

2.4. Psychological Representation

There has been considerable work by psychologists on the form and function of representation used by the action oriented portions of the human cognitive architecture. That is, how do humans represent information that is not for the purpose of symbolic, cognitive thought (such as planning), but rather for sensori-motor system use when carrying out actions? Although some studies show that people will not use memory when that option is available [7], none have proposed that humans have no representational system in the manner of Brooks [17]. Particularly in the visual domain, representation plays an important role. Aloimonos has shown how certain ill-posed vision problems (e.g. shape from motion) become well posed when the agent can use multiple camera views with an understanding of the motions between them [4]. This implies some memory of previous images, which itself implies some representation to hold those memories.

Shimon Ullman proposes a model of the "intermediate" human visual system based on small programs called visual routines [76]. An important concept used by these programs is the ability to remember, or "mark", portions of the image that have already been analyzed. He proposes a "marking map" that holds the location of portions of the scene described by the incremental representations built during scene analysis.

Pylyshyn and Storm [63] propose the FINST model where a limited number of "reference tokens" can be bound to visual features that they then track. Associating a FINST with a visual feature is a prerequisite to further processing involving that feature. The FINST model is an object-based model of attention, as opposed to a location-based model, in that FINSTs do not point to particular locations in space, but to particular visual features (and

continue to do so as the features move). It is different from other attentional models in that it supposes that attention can be directed to multiple places in parallel. This position is further supported by Yantis [85] whose experiments show that people can track multiple independent elements by imagining them as the vertices of a "virtual polygon". While the previous works have been concerned with targets in the visual field, Attneave and Farrar [6] show that subjects can remember and track the locations of elements that were initially within the visual field, but have since moved out of view. The parallel, object-based model of attention, where targets can be in or out of the visual field, is consistent with the representation systems designed by this thesis.

Other researchers have concentrated on how representations are used for action. Ballard et. al. [7] describe how cognition on the scale of actions (approximately 1/3 of a second) consist of small "programs" with "variables" that are bound at action time. These "variables" are bound to objects, parts of objects, or visual features of appropriate world aspects that will be manipulated by the action program.

This thesis advocates the creation of linked layers of representation where the represented information gets more complex (requires more computation to determine) as the layers that use it get further from the PA layer. Psychology researchers have investigated similar links between representations used by different bodily systems. Feldman [25] investigates the connection between the representation used by the visual-motor system and more cognitive representations that might be used for other tasks. He describes a hierarchical representation system with four levels. The first is the representation of space within the current fovea and the second synthesizes various foveal fixations to (ego-centrically) represent the "stable world". This system, Feldman says, primarily interacts with the "environmental

frame" that encompasses the other two levels and represents space in a non-body coordinate system.

Although not strictly dealing with representation, the two-handed rod manipulation experiments of Guiard and Ferrand [31] show the usefulness of multiple representations. People manipulate rod handled tools (like a rake or a broom) by having one hand perform coarse motion and the other fine motion. The fine motion hand operates in a coordinate system set by the coarse motion hand. While it seems plausible that people can represent a rod handled tool as a single conceptual entity, if they need to manipulate it, representations of the rod in different coordinate systems are needed. In the deictic program paradigm of Ballard [7] variables for right and left hand positions would be bound to portions of the rod, probably with one hand's position defined in terms of the other's.

2.5. Other Design Methodologies

Since designing the software for a robot architecture is an exercise in software engineering, methodologies in this field must be examined. There are two principal places in which current software engineering design strategies can be applied: the decomposition of the agent's goals and the design of an agent's representation. While the latter is the primary concern of this thesis, the former is a necessary step in the methodology of chapter 3. No current software engineering methodology addresses the design of representation systems for autonomous robots, so I address the use of the methodologies in decomposing the agent's task and how that may lead to a system of representation.

Structured programming [84] decomposes the problem using a top-down approach. The problem is divided into sub-problems, each of which are further divided until some basic level of implementability is reached. The methodology presented here applies this ap-

proach, as well as a bottom-up approach, to decompose the agent's goals into a series of tasks. Since structured programming is a tool for general software design, it has little to say about the implementation of the various tasks in the decomposition and so is open to all the problems of using arbitrary data structures for representation.

Object oriented design [11] espouses a philosophy of decomposing software not along functional lines, but into a collection of cooperating entities, called objects, that encapsulate both data and the functions that operate on that data. Object oriented design does not seem to map to the problem of decomposing the agent's tasks as well as structured programming. Since the decomposition is not by function, the designer has to do something like creating "goal objects" in which the agent's capabilities represent functions that move the world state (data) toward some goal. These goal entities would then have sub-goal entities that moved portions of the parent goal's world state toward a goal. While other possible interpretations of the "objects" in the agent's system are possible, a pure functional decomposition is what is needed in the end. The designer has to reason how to use the agent's capabilities to achieve its goals and the definition of "objects" here seems to get in the way.

However, unlike other methodologies, the object oriented approach has much to offer in the design of representation systems. The representation systems developed in the following chapters are "object based" where object refers to a relevant aspect of the world (often a physical object). These representations encode both an object's relevance to a particular task and how a particular action will be carried out on that object. For example, if the agent's task were to pick up a soda can, the representation would indicate that some object was a soda can, and as such, picking it up means positioning the effector(s) in a certain way. Note that the action of picking up a dumbbell would require different effector positions and

so a representation that indicated an object was a dumbbell would have to encode those. This combination of function and data in a single structure is similar to the object-oriented philosophy. In fact, the encapsulation/information hiding aspect of the object-oriented model is particularly appealing because it maps well to this thesis' concept of creating "roles" in tasks that can be "filled" by various entities in the environment (see section 3.4). The roles form equivalence classes of objects that can be used by the agent to perform some function. The details of the selected object can be (more or less) hidden from the agent's action selector. The object-oriented paradigm's abstraction of class hierarchies is also useful, in the design of levels of representation. If different levels are used by different layers of an agent architecture, different abstractions of the environment can be placed in different levels so that the different layers can operate on data at the right "granularity" for their decision making processes.

The object-oriented model does not, however, point out the issues that most effect the structures that make up the agent's representation. The methodology presented here directs the designer in considering the structural impact of issues of perception, action and communication.

Communication among the "functional units" (be they objects or functions) in a software design is another important issue. Both data-driven [23] and event-based programming [67] have been proposed to model communication between units. Since communication within the agent architecture is of concern to this thesis, these paradigms can be of use.

Data-driven programming structures a program's "units" into functional blocks where data flows between them along defined pathways. Units are defined in terms of their inputs and outputs with the internal design left to the designer (possibly using another methodol-

ogy). Programs become data-flow graphs that define order of execution by data availability and have no notion of a system wide program counter. Modeling communicating skill networks this way is appealing because any natural parallelism is exploited. However, data-flow graphs themselves do not say anything about how the data should be structured, or what data should be communicated. My design methodology addresses both these questions.

Another communication paradigm is the event based paradigm. In this paradigm, computational units register with other units to receive notification when certain "events" take place. This programming paradigm is often used in GUI systems to represent such asynchronous events as mouse or key clicks [67]. In this paradigm, behaviors may communicate by having the agent's perceptual system generate events in response to certain changes in world state. Those changes would be captured in specific data structures (representation) for the event. Any behavior registered to receive notification of a particular event would receive this representation when that event was generated. The agents developed in chapters 4 - 6 have behaviors that communicate in a manner similar to a blackboard system [47] and events could be used as the means of informing behaviors that new data is available on the blackboard. However, this is an implementation detail and does not address the format of the data as this thesis does.

2.6. Scaling Problems

Many researchers have investigated what I will term as the "scaling problem" for action oriented architectures. That is, can these architectures be made to perform complex tasks? The architectures of Brooks [17] and Agre and Chapman [2] differ in their use of state, but both perform no planning and follow no plans. While following no pre-defined instruction

sequence may make sense in some dynamic environments, researchers have questioned the limits of the tasks that such agents can perform.

Kirsh [39] refutes Brooks' claim that one can substitute control for representation in an agent architecture. In order for this to be true, there must always be enough perceivable (in view and not occluded) cues in the environment that an agent can select an action that will have no adverse consequences down the road. This is the only way that an agent can (ultimately) get away with not considering alternate courses of action, i.e. planning. Kirsh believes that reasoning (planning) is required for complex tasks and planning requires an abstract, symbolic model of the world. Therefore the representations that Brooks seeks to eliminate cannot be removed from the agent architecture entirely.

Tsotsos [75] also argues that the pure Brooksian approach to agent construction cannot scale to human intelligence, by showing that visual search, a common activity in intelligent agents, is NP complete without a target (actually, without an "explicit" target, meaning the agent doesn't have a description of its target that can be used to simplify the search process). This "unbounded" visual search involves grouping random collections of pixels and analyzing them to see if one of the agent's stimuli is present (and so the agent should execute its response). Interestingly, Tsotsos also shows that with a small amount of state (a description of the target being sought), visual search is linear. A perceptual description would have to be stored somewhere, i.e. in some representation. The representation designed by this methodology provides targeted visual search.

Pengi [2] has some limited state in its visual system that can limit the complexity of the search task. Agre and Chapman refer to systems with this limited state as deictic, or pointing, because the state "points to" (holds) only currently relevant aspects of the game. How-

ever, as pointed out by Ballard [7], the Pengi architecture is a cognitive one and not a visual one. That is, the advantage of such an architecture is the simplicity of the process of selecting the current action when behavioral "variables" (markers) can be bound to different objects in the game over time. The problem is not with what the agent stores in its internal state, but with what it might have to store at any given moment. The agent must search the entire (perceivable portion of the) environment, all the time, for all the stimuli to which the agent has a response. Even after a marker is bound, it might need to be rebound to a different object, e.g. when the-bee-closest-to-me changes to a different bee. Agre and Chapman had access to their game's internal data structures, which contained a unique labeling of each object in the game. This allowed them to switch their markers to new targets at little computational expense. However, for systems that must deal with physical sensors, the more complex the domain, the more important aspects there will be to register and thus the more computational burden there will be on the perception system.

Maes [48] creates a behavior-based system with a similar scaling problem. In her architecture, behaviors are connected to other behaviors whose preconditions they achieve. Sensors put activation energy into the behavior network based on precondition predicates they detect as true and when a behavior has all its preconditions activated, it will run (possibly activating other behaviors). For a complex task, there can be huge numbers of predicates that the agent needs to monitor for and since there is no behavior hierarchy, there is no means of limiting the network to considering only those predicates needed by the current task. In other words, the agent must always look out for stimuli related to all tasks that it might ever want to execute because at any time the requisite predicates may become true causing a switch in the agent's activity. Bryson [18] argues that the complexity of creating,

debugging and tuning such a fully parallel, behavior-based system (as opposed to one with a control hierarchy) outweighs the benefits in reaction time that parallel systems may provide. In her study, the tuning of such a system was so difficult that it could not be made to perform as well as a system with a behavior hierarchy, at the same task.

Neither the stateless nor the purely deictic behavior-based approach scales. As we'll see in succeeding chapters, my agents attempt to avoid this scaling problem by only looking for aspects of the environment that are important to their current task, even though they lose some ability to recognize serendipitous circumstances because of this.

A related piece of psychological work is presented by Patalano and Seifert [61]. They show that humans can encode known future goals or goals that they have been sidetracked from by associating them with objects required to complete them. If a subject sees some item needed to complete a goal other than the one they're currently pursuing, they may grab it since they'll need it soon. This is interesting because it shows the people can recognize objects as useful, even when they are not related to their current activity. While this seems to leave humans open to the same scaling problem, the experiments show that people do not recognize more and more objects as useful as they get more and more goals. They seem to strike a balance between the number of pending goals and the items they recognize as important to achieving them. When there are too many goals, people will focus on one at a time and fail to recognize important objects for pending goals. The methodology proposed in this thesis does not preclude examining the environment for aspects that may be useful in the future. It does however, caution the designer that the computational cost of too much "look ahead" can ruin the reaction time of the agent.

Chapter 3   A *Representation Design Methodology*

This chapter proposes a design methodology for autonomous agents operating in dynamic environments. The main contribution of this methodology is a structured method of analyzing an agent's task and capabilities to guide the development of the agent's representation system. Agent designs based on this methodology are intended to interact with the world in a rapid, stimulus-response fashion. That is, the time delay between perception and action must be as small as possible. The component of the agent architecture dealing with sensors and effectors is called the perception/action (PA) system and is conceptualized as running in a "tight" loop using sensor data to control effectors and effectors to manipulate objects and direct sensors. Due to the computational complexity, the PA system must have almost no explicit inferencing capabilities to remain responsive. Agents designed via this methodology are also assumed to operate in dynamic domains, i.e. environmental change causes perceptions to become outdated.

It must be understood that this is a methodology and not an algorithm for agent software design. As such, the methodology cannot generate *provably* optimal agents along any lines. In fact, there are no tasks that necessitate the use of this methodology. However, this methodology can be a powerful tool to focus a designer's attention on certain aspects of "agentness" that are of critical importance to the creation of autonomous systems, particularly the representation used within the control structure. I will argue that this methodology can help

designers create agents that are efficient and effective at tasks in the domains of concern

defined below.

3.1. Task Domains

As discussed in section 1.3.1, the task domains of interest in this thesis share a number

of characteristics. They are summarized in table 3.1.

Table 3.1: Summary of Domain Characteristics

1	The environment is dynamic	Current sensor data is more valid than previous sensor data.
2	The task contains different aspects.	The different facets of the task can be handled by "independent" architectural components within the agent.
3	Knowledge of large-scale space [43] is required.	The agent cannot perceive all important aspects of the environment a priori or at any one time.
4	The task can be decomposed.	The task can be described by a hierarchy of tasks and subtasks and allows important environmental entities to be viewed at different levels of abstraction, e.g. part and whole.
5	The entire plan cannot be made a priori, or replans do occur.	The task requires the agent to have on-going "higher level" control.
6	The environment is suitable quantifiable with coarse metrics.	The agent can use crude maps effectively.

Domains with these characteristics are well suited to agents with perception/action sys-

tems. The methodology emphasizes aspects of the design process that are important in such

domains.

3.2. A Design Methodology

This section outlines a design methodology for autonomous agents. The steps of the

methodology take the form of a series of questions that the designer must answer. For each

step, I briefly describe the questions and illustrate them with a running example of an agent that is designed to "walk the dog". I then discuss the rationale behind the questions and summarize the contributions of the step to the design process.

At the start of the design process, certain facts about the agent, its task, and its environment must be known. For the task and the environment, a high-level specification must exist along with the availability of more detailed information if it becomes necessary. For the agent, the designer needs a description of its capabilities. In the example task, the agent must walk a dog along a planned route to the park (see figure 1) and then play fetch with
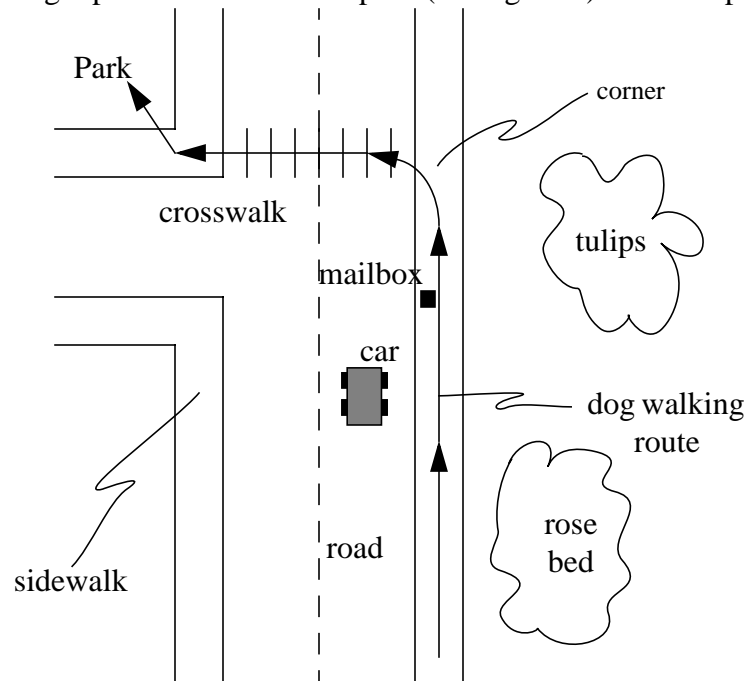
Figure 1. Dog Walking Environment

the dog. For exposition purposes, figure 1 represents a number of constraints on the task. The agent must follow the indicated route, must stay on the sidewalk, must cross the street where indicated, etc. A complete list of constraints appears in appendix C. The agent's environment consists of a road, sidewalk, crosswalk, and dog. There are also several flower beds along the route to the park and possibly cars in the street. The important agent capa-

bilities include navigation abilities, a leash effector that can be reeled in to keep the dog

within various distances, a pressure sensor to detect which direction the dog is pulling the

leash, a vision system that can identify important landmarks, curbs, tulips, roses, the dog,

the leash, the ball, the mailbox and cars, as well as a reloadable ball launching system so

the agent can play fetch with the dog.

Throughout this chapter, I will refer to many agents and their tasks (including the "walk the dog" agent). The task, environment and capabilities of these agents will be summarized in insets like task specification 1.

Task:	Walk a dog to the park and play fetch with it. The dog must be kept safe from cars and out of the flowerbeds along the route.
Environment:	The park and the surroundings of the route there (street, sidewalk, cars, flowerbeds, crosswalk, mailbox). Flower beds can contain either tulips or roses.
Agent perception capabilities:	Landmark detection system to identify "corner", "park" and "crosswalk", obstacle detector for navigation, sidewalk edge detector, tulip and rose bed detector, dog and leash detector, dog tired-ness sensor, proprioceptive sensors to detect own motion.
Agent action capabilities:	Leash effector with direction sensor (allowing agent to keep dog close by, out of the street and moving forward), reloadable ball launch system (allowing agent to throw ball and get it back from dog), navigation ability (allowing agent to move toward objects and across the street).

Task Specification 1. Walk-the-dog

This thesis broadly defines capabilities as anything that the designer

has to work with at the beginning of the design process. Capabilities can be anything from

detailed hardware descriptions like a stepper-motor to full behaviors like "navigate to a

landmark". For the "walk the dog" example, agent capabilities are mostly given as behav-

iors (e.g., a navigation system that can maneuver to landmarks, instead of wheels and a

camera), but for other agents, bottom-up thinking is applied to turn capability descriptions

into behaviors (see chapters 4 - 6).

Finally, I use hyphenated phrases, e.g., walk-the-dog, as proper nouns to aid the reader

in understanding the goal of a task. While these phrases appear to make certain steps of the methodology trivial, they are merely names that could be stripped of their semantic content by using titles like task-37. However, it is easier to understand that the goal of the walk-the-dog task is to have the dog walked, as opposed to the goal of task-37 being to have the dog walked.

3.3. Task Decomposition

The first step in the methodology involves creating a hierarchical decomposition of the agent's tasks[1]. The agent will have certain capabilities (both hardware and software) that are given to the designer. These can be thought of as the agent's "primitive" skills, i.e. the fundamental processes from which all of the agent's actions are built (see section 1.3.2). If the task decomposition is thought of as a tree with the original task specification at the root, these skills are the leaves. The job of the designer is to bridge the gap between the task specification and the skills. When creating the decomposition, it is important to consider the following:

What are the agent's primitive skills? The answer to this question depends on the agent's capabilities and the degree to which each can be viewed as a "black box". Although the design begins by considering the skills, ultimately creating a task decomposition must be both a top-down and bottom-up process. The designer must decide on the agent's skills while breaking down the overall task into a sequence of subtasks that achieve the agent's goals. This brings up several more important questions.

Which tasks can be decomposed into sequential subtasks? Which can be decomposed into

---

1. Task decomposition is a major part of general requirements analysis and it remains mostly an art [66][84]. This methodology provides a structure in which to practice that art.

parallel subtasks? Interior nodes of the decomposition tree will be broken into subtasks whose overall outcome is meant to complete the parent task. In this way, the leaf skills can be combined to achieve a greater range of behavior than that of the individual skills.

3.3.1 Example Task Decomposition

In figure 1, we can see that the dog must be taken along a certain route to the park (remember task specification 1 and figure 1 represent the given task constraints). First the dog must be walked along the sidewalk to the street corner with the crosswalk. Then the agent and dog must cross the street on the crosswalk and go into the park. Finally, the task specification says the agent should play fetch with the dog. We can derive further specifications for the steps on the way to the park by examining the map in figure 1. For example, to keep the dog on the sidewalk, the dog must be kept out of the street and out of the flowerbeds. At the crosswalk, before crossing the street, the agent should look for oncoming cars.

A few more details can be discerned by careful thought about the processes involved. When walking the dog, the agent moves and the dog is meant to follow. If the dog does not follow, the agent must tug on the leash. Since the leash is the agent's only mechanism for controlling the dog, the leash must be manipulated to keep the dog "safe", i.e. on the sidewalk, and to keep the dog moving forward. A good strategy for controlling the dog might be to keep it nearby and just tug the leash when the dog's heading needs to be changed, but in the end this is up to the designer.

When the agent is in the park, it needs to play fetch with the dog by repeating the cycle of throwing the ball and waiting for the dog to retrieve it. However, the game should come to an end when the dog has had enough exercise and so the agent needs to periodically check if the dog is beginning to tire.

Figure 2 shows a hierarchical task decomposition of the walk-the-dog task as described above. Ovals represent tasks and subtasks, while lines represent the "is subtask of" relationship between the higher oval (the parent task) and the lower oval (a subtask). Each of the subtasks in figure 2a, is further decomposed into additional subtasks in figures 2b-f.

The overall task of walking the dog is broken into five tasks that follow the pre-planned route and give the dog some exercise. Each of these consists of a set of leaf tasks that can be implemented based on the agent capabilities from task specification 1. For example, keep-dog-moving-forward, keep-dog-out-of-street, keep-dog-close-by and reel-in-leash-near-flowers are all leash control skills that monitor the dog's position and tighten or loosen the leash at various times. Keep-dog-out-of-street monitors the positions of the dog and the street and tugs on the leash if they become too close to one another. Keep-dog-nearby allows the dog a certain amount of leash depending upon the agent's location along its route.

Walk-toward-corner, walk-toward-park and move-across-when-clear are all agent navigation skills. Watch-for-flowers, watch-for-cars-at-crosswalk and wait-for-dog-to-return are perceptual skills based on known characteristics of the agent's vision system. Throw-ball-to-open-area and get-ball-when-dog-returns are ball handling skills that make use of the agent's reloadable ball launching system. Is-dog-tired is a perceptual skill that determines if the dog is tired of playing fetch. Of course, most tasks make use of multiple capabilities, such as throw-ball-to-open-area, which uses the vision system to detect a clear place for the dog to run and the ball launching system to throw the ball. For this discussion, the implementation of these tasks and the reasons for this specific decomposition are unimportant. What is important is that the leaf nodes all represent primitive actions based on the agent's capabilities and the actions overall effect produces the right qualitative behavior.
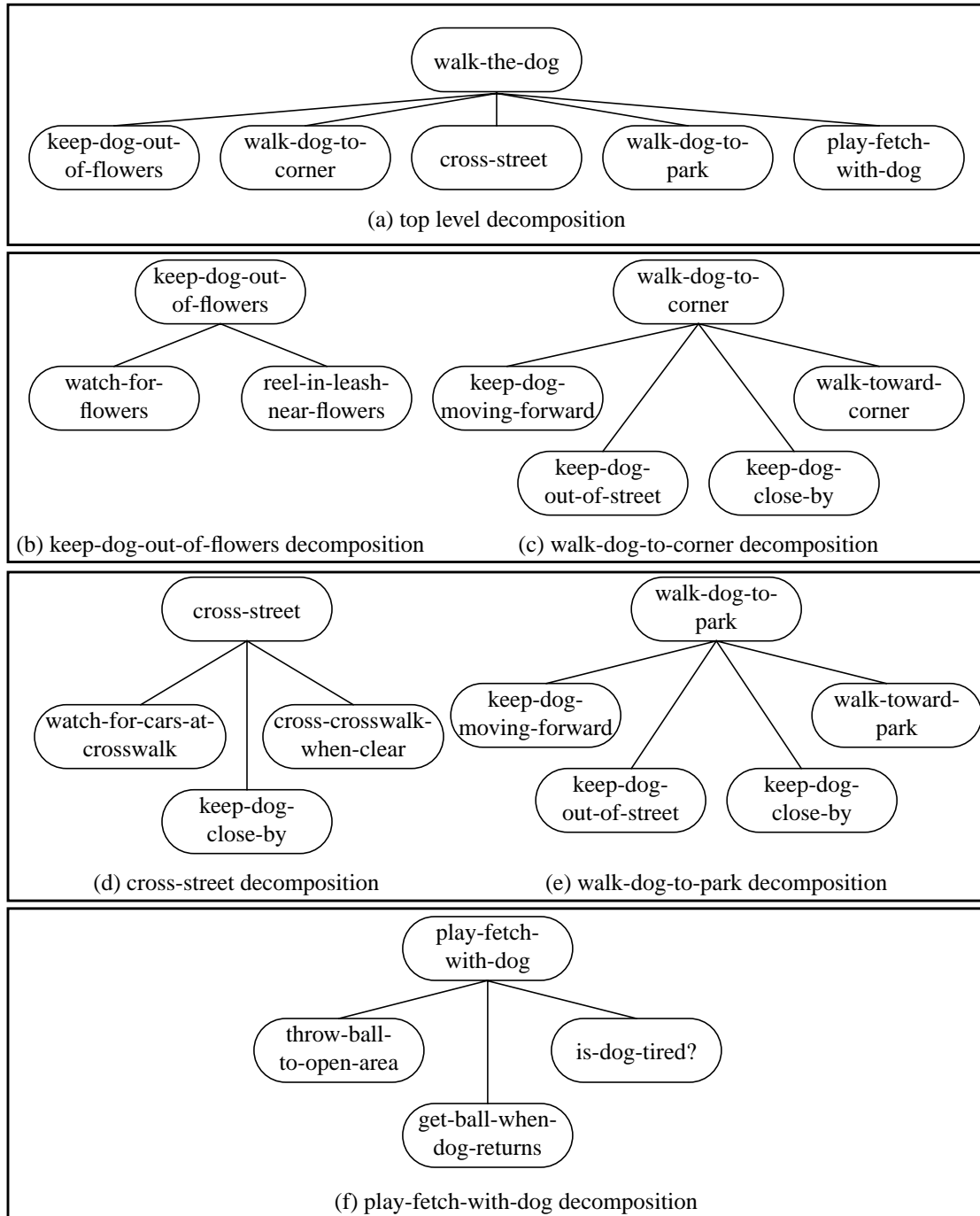
Figure 2. Walk-the-dog Decomposition Hierarchy

(a) top level subtask flow

(b) keep-dog-out-of-flowers subtask flow

(c) walk-dog-to-corner subtask flow

(d) cross-street subtask flow

(f) play-fetch-with-dog subtask flow

(e) walk-dog-to-park subtask flow
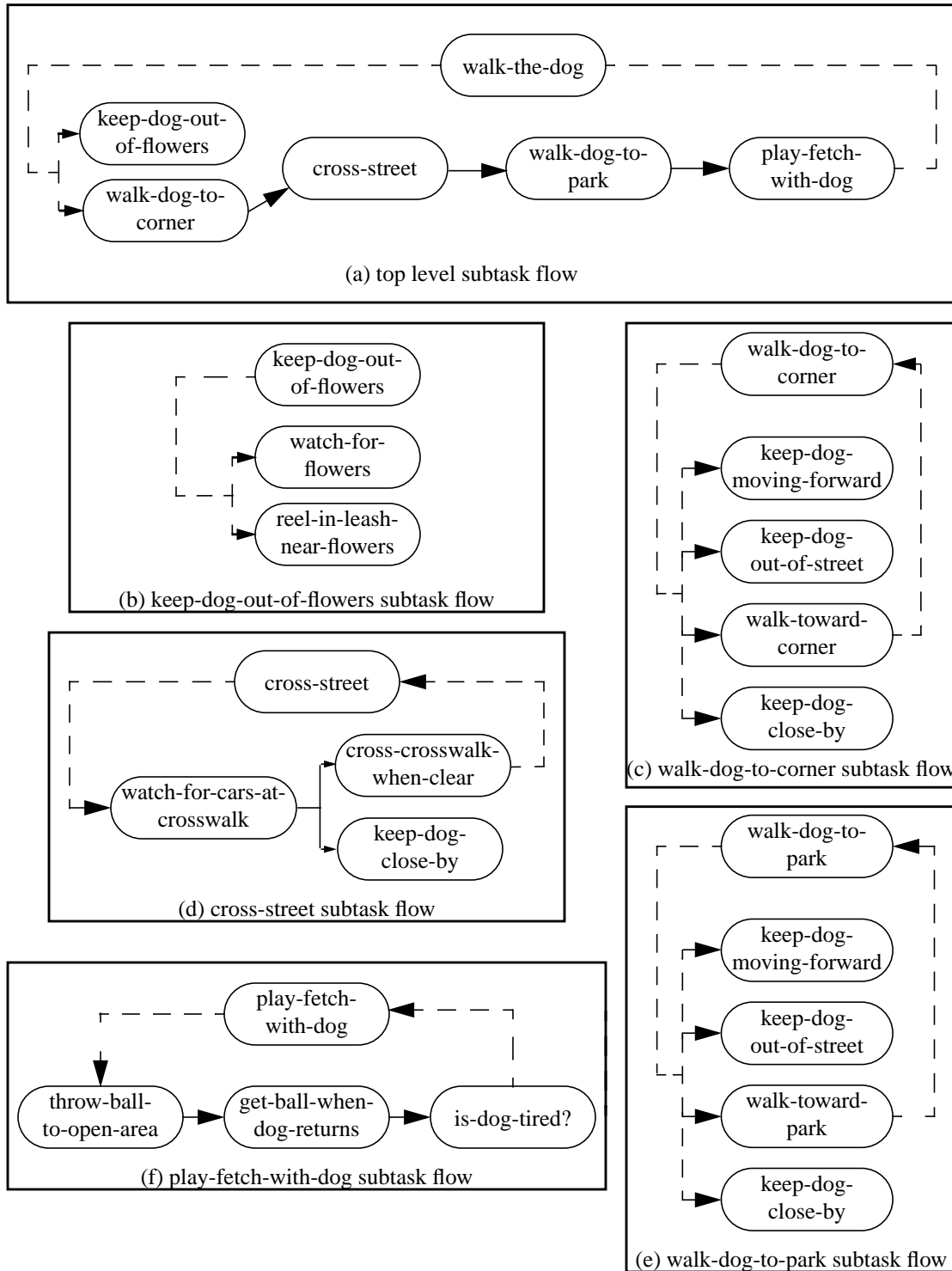
Figure 3. Walk-the-dog Subtask Flow Diagrams

Figures 3a-f show the "flow of control" during task execution, i.e. the order in which sub-tasks are executed to achieve parent task goals. Subtask ovals that are stacked above one another indicate that these subtasks are executed in parallel. A solid arrow from one subtask to another indicates that when the left subtask completes, control transfers to the right sub-task. An arrow from a higher oval to a lower one indicates that the parent task executes one or more of its child subtasks. A dashed arrow from a subtask back to a parent task indicates that the child has achieved its goal and therefore the control flows back to the parent task. Figure 3a shows the basic flow of the walk-the-dog task as it has been described (go to the corner, cross the street, go to the park and play fetch with the dog). Figures 3c and 3e show that a number of subtasks are executed in parallel when the agent walks the dog to some location. Three subtasks control the dog to keep it safe (out of the street and nearby) and keep it headed forward. The final subtask moves the agent toward its destination. The route to the corner is lined with flower beds and so the agent must keep the dog out of them while walking to the corner. Figure 3b shows the 2 parallel subtasks that monitor for flowers and reel in the leash whenever they are detected. Figure 3b has no control flow back to the par-ent indicating that the subtasks will continue to execute until some event (external to this task) causes the agent to stop executing the parent task (and thereby stop executing its sub-tasks). In this case, the agent will continue to run the subtasks of keep-dog-out-of-flowers until walk-dog-to-corner completes. Figure 3d shows how the agent crosses the street. First the agent checks for oncoming cars, then if there are none it crosses the crosswalk keeping the dog close by. The play-fetch-with-dog task of figure 3f shows a series of tasks that play fetch with the dog. The agent throws the ball, waits for the dog to return with the ball and checks if the dog is tired. While the last subtask tests the dog's status, there is no obvious

change in control flow based on this information. However, the tiredness of the dog is passed up to the play-fetch-with-dog task and that task determines whether to follow the dashed arrow of figure 3f (to throw-ball-to-open-area) or the dashed arrow of figure 3a (to walk-the-dog).

3.3.2 Decomposition Rationale

The purpose of this step in the methodology is to have the designer think about the relationship between the agent's task and the agent's capabilities. A hierarchical task decomposition can provide a form of information hiding [60] to some tasks. The tasks at any one layer of the hierarchy do not need to understand the internal workings of the tasks at the other layers, only the mechanism to communicate with them (see Section 3.7). The designer needs to create this hierarchy from both the top-down and the bottom-up. That is, the designer must think about what services a task at a certain layer of the decomposition should have available from the layer below. At the same time, the designer must consider how the goal-achieving behaviors of multiple tasks, at a certain layer, can be combined to present an abstraction at a layer above. The tasks at the lowest layer of the decomposition (referred to as "behaviors") will come most directly from the agent capabilities, while the tasks at the highest layers will rely on the abstractions of tasks from all the layers below.

The creation of the task decomposition begins by designing the agent's skills. What exactly constitutes a skill will depend on the agent's given capabilities and the abstraction they present, i.e. how much they are a "black box". For exam-

Task:	pour coffee from a pot in a coffee machine into a cup
Environment:	coffee machine with coffee pot in machine, coffee cup
Agent:	pair of 6-DOF arms with attached grippers, visual sensor to detect pot/cup alignment

Task Specification 2. Pour-a-cup-of-coffee

ple, an agent with an arm and gripper effector has the capability to *grasp* objects, i.e. close

the gripper "fingers" around an object, but pre-made software libraries might allow the

same agent to *pick up* objects (involving moving the arm to the object, positioning the grip-

per, grasping and then lifting the object). From the agent's point-of-view, this "pick up" be-

havior is a black box. As a further example, consider the agent capabilities described in task

specification 2. Given the agent hardware description, the designer might create behaviors

to grasp objects like a cup or coffee pot, pull such a pot from a coffee machine, align a pot

with a cup, and tilt a pot to pour coffee.

Task:	drive car to the store
Environ-ment:	car on road with other cars, lane markings, street signs, traffic lights, start in driveway
Agent:	grasping/pulling effector, obstacle/landmark/lane detector, steering skill, color detector

Task Specification 3. Drive-to-the-store

Once the agent's basic behaviors have been designed, the designer must decide both how skills can be combined to achieve various task goals (bottom-up) and how various task goals can be decomposed into goals that can be achieved by

other tasks (top-down). This can be done by considering the effects of executing various

tasks (starting with the skills) in parallel or in sequence, as well as, how tasks can be broken

down into sequences of subtask or groups of parallel-executing subtasks[2]. For example,

pouring a cup of coffee (see task specification 2) consists of the sequence of grasping the

coffee pot's handle, removing it from the machine, aligning it over the cup, and tilting it to

the proper angle. Other tasks are composed of subtasks that execute in parallel, such that

the desired behavior "emerges" [16]. For example, the driving task in task specification 3

2. The reader should note that the discussion of serial and parallel execution of tasks does not mean that the task decomposition is an and/or tree [64]. Task decompositions used in this methodology do not have "or" branches, though a particular subtask may select among various choices when deciding what to do.

might consist of steering to stay in your lane, avoiding other cars and watching for the next location where you need to turn. These activities must be done in parallel in order to drive safely and we consider them to all be part of normal driving. The designer should also look to create behaviors bottom-up, by executing groups of more primitive behaviors in parallel. For example, if the designer had watch-for-lane-changes and check-for-brake-lights behaviors, he might execute them in parallel to create the avoid-other-cars behavior. It is worth pointing out that all subtasks will ultimately decompose to some kind of hardware-triggered transitions in the agent's electronics. However, such details are well below the "level of abstraction" that is important at design time and the decomposition should stop at the level of the agent capabilities.

The reader may ask, "since all agent behaviors ultimately arise from the execution of a collection of the agent's skills, why not decompose the agent's task into a 'flat' ordering of these skills?" While no argument can be made that a hierarchical decomposition is necessary for any task, as we will discuss later, it allows the designer to create an agent architecture that separates the agent's deliberating and acting concerns to allow efficient and effective operation in dynamic domains.

For any given task, there are multiple feasible decompositions. Sometimes a particular decomposition will represent different possible orderings of steps in the agent's task, or choices among alternatives available to the agent. For example, the pour-a-cup-of-coffee agent can align the cup with the pot or the pot with the cup. In this case, it probably doesn't matter and so the designer can just pick one. However, a car driving agent may be able to take any of several routes and might want to select among them based on weather, traffic conditions, etc. In this case, the designer will want to think of the various decompositions

as being part of a larger tree, with a runtime decision as to which branch of the tree will have its subtasks executed. The designer can think of this larger tree as an and/or tree, where each node is a complete hierarchical decomposition of the type advocated in this methodology (that is, without decision points in the tree itself, but possibly within the nodes).

The designer should be aware that creating the task decomposition is an iterative process with the rest of the methodology. The answers to the questions in the remainder of the methodology are dependent on the task decomposition and so the designer can arrive at a situation where some other question cannot be satisfactorily resolved without changing the decomposition. For example, the designer may at first believe that a task will be used in two different parts of the decomposition. Later in the perception question (section 3.6), the designer discovers that the two situations in which the task is to be used, occur in suitably different parts of the environment so that the perception needs to be different in each case. This would involve changing the decomposition to have two different (situated) tasks instead of the original one.

Once the designer has a task decomposition, task flow diagrams should be created. This process should be assisted by the fact that when decomposing a task into subtasks, the designer is likely to have imagined an ordering of those subtasks.

For the most part, creating the decomposition and task flow diagrams is part of the art of agent design and this thesis has little specific to suggest. Software engineering provides a number of methodologies for decomposing tasks into smaller conceptual units, e.g. [11][23][67][84]. Some of the more popular methodologies are discussed in chapter 2.

3.3.3 Decomposition Summary

In this step the designer is to create a hierarchical decomposition of the agent's overall task. This begins by deciding on the agent's most primitive behaviors. Next the designer considers how the agent's "top-level" goals can be broken into subgoals, and how the behaviors can be combined to achieve those subgoals. This step takes advantage of domain characteristics 2 and 4 (see sections 1.3.1.2 and 1.3.1.4) by exploiting the innate hierarchical structure of the task so that (in future steps) specialized representations for these tasks can be created.

A hierarchical decomposition is useful in two ways. First, the hierarchy gives shape to the final control structure of the agent (see Section 3.8). Second, tasks can hide operational details from their parent task [60] so that the parent task need not spend its time considering too many fine details.

3.4. Identify Task Roles

What are the task roles? After creating the task decomposition, the next step is to identify the *roles* in each task. Task roles are the aspects of the environment that effect the outcome of the task. Typically, they are the objects that the agent acts upon during the task (e.g. it's "the hat" when the agent's task is to "put on the hat").

What entities can fulfill those roles? Once the task roles are determined, the designer must decide what entities in the agent's environment can fulfill those roles.

3.4.1 Example Task Roles

Consider the environment shown in figure 1. The designer must first consider the entities in the environment that are important when walking the dog to the corner. First, the agent

needs to know about "the corner" in order to identify and navigate toward it. Obviously the agent also needs to have a dog. The agent also needs to be conscious of the flowers along the way and of the location of the street (or sidewalk edge) with respect to the dog. These are the primary entities (corner, dog, street and flowers) needed to get to the corner in accordance with the task specification.

Next the designer must decide what aspects of the environment are important when crossing the street. The agent needs to be conscious of oncoming cars, it needs to identify the crosswalk (to get across in the legal manner) and, of course, it needs the dog. After crossing the street, the task of walking the dog to the park is easier than walking the dog to the corner because there are no flowers along that part of the route. So, the agent need only be concerned with the location of the dog, the street and the park. Finally, when playing fetch with the dog, the agent must be able to detect open areas to throw the ball to, as well as detect the dog and the ball to retrieve the ball when the dog returns. Obviously, the dog is the important entity for determining if the dog is tired.

I have used various phrases such as, "know about", "have the concept of" or "be concerned with" to designate the aspects of the environment that are important to the agent during a specific task. Describing a task with such phrases is a key to the designer that the associated aspect is a role in the task. That is, these phrases signify that some object in the environment possesses qualities that the agent must sense and respond to. There is a dog role in all the tasks because the agent must monitor the dog's location and redirect the dog if necessary. If the agent has the perceptual capabilities to designate some portion of its environment as "the dog", it can use that portion in actions with the task role "dog". For convenience, I have repeated the decomposition of figure 2, with the roles in bold, in figure 4.
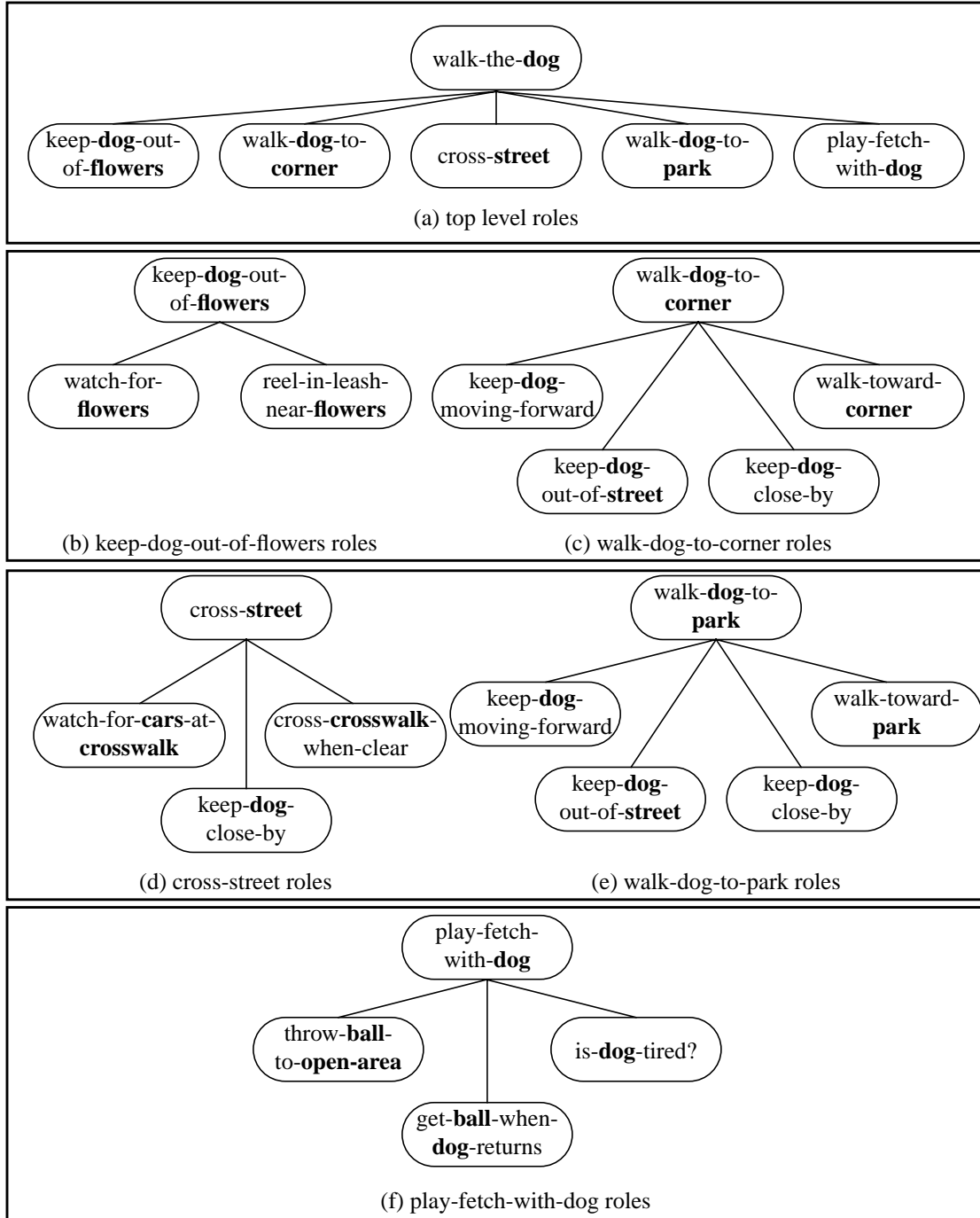
Figure 4. Walk-the-dog Task Roles

I caution the reader that while the hyphenated task name phrases of figures 2 and 3 appear to make the selection of roles obvious, the designer will not have these well-crafted names when developing the decomposition. One method that allows the designer to identify task roles and create English language task names is to describe the task's goals in words. Often the nouns will be roles in the task. Skill in doing this will allow agent designers to create the sort of semantically meaningful task names used throughout this thesis.

However, even with a goal describing task name, all roles may not be obvious and chapters 4 - 6 contain many examples. Often, some roles arise from the interaction of tasks (which is partly an implementation issue). For example, the agent in chapter 4 has a task that detects obstacles and so this task has a role for an obstacle. However, another task the agent's wheels also has a role for an obstacle because the task steers the agent around them. This second obstacle role was needed because detect-obstacles only finds obstacles, and expects some other task to steer around them.

The entities that can fulfill the task roles can also be determined from the environment (in the present example, the information about the environment is encapsulated in figure 1). First, there is the **dog** role. The task specification provides no details about our dog, except that the agent has a dog detector. So, the **dog** role will be filled by whatever entity in the environment is identified by this detector[3]. The **corner** must be that corner on the right side of street at the crosswalk. The **park** must be the park on the other side of the street. While walking the dog to the corner, the **flowers** can be either the tulips or the roses. The **street** will be anything beyond the edge of the sidewalk.

Note that from here on, I use boldface to denote the task role and plain text to denote ac-

---

3. This "dog detector" is given as an agent capability to simplify the discussion here, but it is an over-simplification of a complex problem.

tual entities in the world. So, **dog** represents *the* dog (not necessarily explicitly as discussed in section 1.2.2) that the agent is walking and while that dog may be just one of several dogs in the environment, its the important one.

The designer must choose the important portion of the street to fill the **street** role in the cross-street task. One solution is for the **street** to represent the area in between the two curbs (edges of the sidewalks) on either side of the street at the crosswalk. The **car** role can be fulfilled by any large moving object in the street. In this case, the role name is slightly misleading since the **car** does not have to be played by a car, but could be played by a truck, a bus, or a grand piano rolling down the street toward the agent and its dog. Anything that is large, in the street and moving toward the agent should be avoided.

The ball used to play fetch with the dog can be whichever ball the agent carries with it to the park, but the agent should make sure the dog returns with the same ball that was thrown (and this is done by the get-ball-when-dog-returns task). This means that ball will play the role **ball** in the ball throwing and reloading tasks. The open-area in the park can be any space which is clear of objects that the dog should avoid (more on how this is determined in Section 3.6.1). This open-area will then play the **open-area** role.

3.4.2 Task Roles Rationale

Roles are task specific descriptions of environmental aspects that influence the agent's behavior, i.e. what the agent must sense and act upon. In a dynamic environment the agent cannot afford to be concerned with anything else [8]. This is why task roles are so important. They force the designer to pick out what is important in a particular task and allow the agent to concentrate its resources on just those entities [35]. This is not just a matter of efficiency, but of effectiveness. It is generally impossible for the agent to build a complete

model of the world, which it keeps up-to-date in a reasonably dynamic environment[4] [49]. There is simply too much information available to the agent for the agent to extract it all and then decide what to use. The agent must concern itself with only the environmental aspects relevant to its current task. This allows the agent to be designed to accomplish each of its tasks using tight perception/action loops that involve determining information about object(s) associated with task role(s) and choosing corresponding effector commands.

The pour-a-cup-of-coffee task of task specification 2 requires some object to play the role of *the* coffee pot and some object to play the role of *the* coffee cup. Other environmental aspects are not important. Cups other than the cup that is getting filled are not important and neither are any of the myriad of other data which may be available to the agent's sensors (e.g. the contents of the kitchen cabinets or the color of the floor).

At this point it should be made clear that a role's "description of environmental aspects" does not necessarily refer to perceptual qualities, like red, or round, but possibly to qualities that only have meaning within the current task. For example, a navigation task might have to steer around obstacles while heading for a goal. Since obstacles and the goal determine the agent's route, there should be **obstacle** and **goal** roles. Any entity fulfills the **obstacle** role when it is positioned between the agent and its goal, regardless of its perceptual properties. For another task that same entity may play a different role or none at all. Of course,

---

4. I define a "reasonably" dynamic environment in a somewhat circular manner. That is, it is an environment where changes occur at a rate that precludes the agent from acting quickly enough to accomplish its goals if it tries to build a complete world model. This definition is clearer when viewed from the perspective of how slowly such an environment would have to change for modern computers to keep up. As an example, consider the robot Spot discussed in chapter 5. Spot uses 100% of a 200MHz Pentium MMX to monitor an area of approximately 50 pixels square in a 256x256 pixel image. Spot can compute stereo and motion information about this area approximately 10 times a second. This allows tracking of the object in this space at a slow walk. If Spot were to perform his computations on the entire image, the process would be approximately 25 times slower. As computers get faster, this factor will be reduced, but the agent will always be able to handle a more dynamic environment by ignoring unimportant aspects of it.

perceptual information can be important as well. If the agent wanted some entity to play the role of **sandpaper**, it might select any object with the perceptual quality of a rough surface. So, the description of a task role may involve both perceptual information and task dependent information.

The designer must understand that roles are more than just descriptions of properties that entities in the environment must have, they contain semantic information. This semantic information allows tasks to be flexible because it tells the task how it should act with respect to certain entities without providing specific descriptions of those entities. The **obstacle** role above tells the designer something about the associated entity beyond a perceptual description, i.e. it is in the agent's path and should be maneuvered around.

The question of what are the roles in a task is deeply entwined with the question of the entities that can fulfill those roles. I use the term "bound" to denote a role that has been associated with an object in the environment. That role is said to be bound to that object and that object is said to fulfill that role. Sometimes the task will only achieve the "correct" result if its role(s) is bound to a specific entity (drive-my-car is only done correctly if **my-car** is associated with the car owned by the agent). Other times, there can be multiple entities that can fulfill a task role. For example, the **flowers** role in the walk-the-dog example could be played by tulips or roses. For the pour-a-cup-of-coffee task, the coffee cup could be a mug, a tea cup or a styrofoam cup. Task roles need not even be associated with single objects. Typically, these objects must bear some spatial relationship to each other. For example, a task's **bowl-of-apples** role might be fulfilled by a bowl object with at least one apple inside of it. Roles may also represent more nebulous entities, such as "the lawn" in mow-the-lawn.

It is also not necessary for the designer to exhaustively specify every entity that could fulfill a task role, rather a description of a class of entities can be used. For example, the **car** role in the watch-for-cars-at-crosswalk subtask of walk-the-dog can be bound to any entity fitting the "large and moving toward the agent" description. In the actual implementation of an agent's perception system, a given perceptual description will always allow "recognition" of any one of a set of entities. This is because these routines will necessarily contain descriptions of perceptual qualities to search for and not unique object identifiers. The number of entities that match a given description will depend on the capabilities of the agent, the implementation of the agent's perceptual system and the amount of noise in the perceptual process.

The designer must also take note of when one role depends on another. This means that any information the agent gets about the entity associated with one role effects the information the agent has about the entities associated with other dependent roles. This often occurs between a role in a parent task and roles in one or more of its child subtasks. For example, the **street** in cross-street depends on the **crosswalk** in cross-crosswalk-when-clear and watch-for-cars-at-crosswalk because the important part of the street is defined by the curbs on either side of the crosswalk. So position information about the curbs (and hence the crosswalk) effects the agent's notion of the position of the street. This primarily occurs when one task uses more abstract properties of some entity than the other tasks, yet each task has a role that should be associated with that entity (or some portion of that entity). For example, assume that the drive-to-the-store task had a **car** role for the car to be driven and the pre-drive inspection contained a check-tire subtask with a **tire** role. There would be a dependence between this role and the **car** role because if any tire is flat, the car is un-

drivable. So, the agent may store the fact that the car is "undrivable" in the **car** role without any indication of why. The fact that this tire was flat would be stored in the **tire** role. Note that a roles can be dependent on more than one other role, such as if the check-tire task had separate roles for each of the car's tires. The **car** would then be dependent on all 4 **tire** roles. Also, the dependency need not be symmetric, e.g. the "undrivability" of the car depends on the state of the tires, but the state of the tires is not determined just because the car is undrivable (the car could be out of gas). Identifying dependencies is important in representation design because the representation must be structured to communicate information between dependent roles. I return to this issue in section 3.7.

Since roles are very context dependent (i.e. their meaning is derived from the agent's current "situated" activity - as defined in section 1.3.2), tasks can be made simple and reusable. If the agent needs to walk to a number of destinations, it can keep executing the walk-to-destination task and just change the entity associate with the **destination**. By using the available task context, designers can create tasks that need not perform inferencing about the environment (see Section 3.8 for a discussion of inferencing by tasks and when it must be avoided). Rather, because the semantic meaning of an object associated with a role is pre-defined, the task's actions can be pre-defined also. The semantics of roles allow the designer to make situational control rules that are flexible. The roles can be bound to different entities, but the task will treat them the same.

In general, there is a trade-off between the flexibility of a task to bind its roles to many different entities and the "situatedness" of that task. The more restrictive a task is about the specific objects to which it will bind its roles, the more likely it is that the designer can optimize a task because it only has to operate on those entities. However, the more entities

that a task can work with, i.e. bind to its roles, the easier it is to reuse that task in different situations. This is particularly important for agents with planners because the tasks that the agent can execute will determine the "operators" that the planner considers. The fewer there are, the easier it is to plan, and of course, fewer tasks need to be implemented.

When deciding on task roles, it may be easiest for the designer to start with the task roles for the leaf nodes in the decomposition hierarchy. As we will see later, the leaf subtasks tend to be executed by the most responsive layer of the agent's architecture, which places strict time constraints on the amount of processing that can be done. The computationally straightforward nature of the leaf subtasks means that the task roles are usually easy to identify. These subtasks can often be aptly described by simple verb-noun phrases like grasp *the coffee mug* or open *the door*.

3.4.3 Summary

The designer must identify task roles because the agent must focus its resources on only the aspects of the environment that are important to the current task. Designing agents this way addresses domain characteristic 1 (see section 1.3.1.1) because it allows the designer to concentrate the agent's processing when the dynamic environment makes it impossible to monitor everything. Identification of the objects in the environment that can fulfill the roles is important because information about those objects will structure the remaining steps in the methodology.

3.5. Representation of Task Roles

What role bindings are shared between tasks? Recall that a task role/entity binding refers to the agent having selected an entity in the environment to play a specific role in its current

task. Thereafter, the task's actions will take place on that object. For example, the pour-a-cup-of-coffee task has a **cup** role. The agent selects a particular cup to use (from the set of all cups that fit the task role's perceptual description), it is considered bound to the **cup** task role and that entity will be filled with coffee. Sharing refers to roles in different tasks being associated with the same object. It also refers to dependent roles that hold different information or a different abstraction of the same entity. The designer must also remember that tasks across levels of the decomposition hierarchy may share roles.

What information about the entity bound to a task role is needed for the task?

For what roles would it be useful to develop an explicit representation? Explicitly representing a task role means that some data structure exists to hold information (from the previous question) about the entity bound to that task role. See section 1.2.2 for a discussion of implicit vs. explicit representation.

How often should the task role information be verified? That is, how often should the agent check that its stored information is still valid? Throughout the remainder of the dissertation, I use the term "maintenance" to refer to the active process of verifying the validity of a piece of information (by allocating perceptual and computational resources to the task of determining that information). Maintenance of information is in contrast to "storage" of information, which refers to passively holding some data in memory. I sometimes refer to deciding how often a task role's information should be verified as deciding on the rate of maintenance for that role.

3.5.1 Example Task Representation

In determining what role bindings are shared between tasks, we need to examine the task environment and our decomposition. The **dog** role occurs in almost all the tasks of walk-

the-dog, not surprisingly, because the dog is central to the agent's purpose. All these tasks should have their **dog** role bound to the same dog and so that role binding is shared. By the route shown in figure 1, we can see that the agent needs to pay attention to only one of the **corner**, **crosswalk** or **park** roles at a time, in order to accomplish the walk-dog-to-corner, cross-street and walk-dog-to-park tasks, respectively. So these roles are not shared between those tasks. However, these roles are shared within the various "subtrees" of their particular task's decomposition (see figure 2) and so we must consider those subtrees in more detail.

Walk-dog-to-corner consists of controlling the path of the dog with its leash while navigating to the landmark named **corner**. At the same time, the agent must keep a lookout for flowers and possibly do additional leash manipulation to keep the dog away from them. The corner in walk-dog-to-corner should be the same physical corner as in walk-toward-corner, so **corner** is shared between the parent and child tasks. Besides **dog** and **corner**, the important roles in the tasks that are active at this time are the **flowers** and the **street**. The **flowers** are shared between the subtask that creates the role binding (watch-for-flowers) and the leash control skill (reel-in-leash-near-flowers) because the leash control skill reacts when **flowers** is bound by pulling the dog away. Obviously, both subtasks need to refer to the same flowers. The **street** in keep-dog-out-of-street will be bound to different parts of the street as the agent walks along. It should correspond to the portion of the street within range of the dog, given the current leash length. These roles (**flowers** and **street**) are not shared with any other subtasks because other subtasks do not try to control the dog with respect to the flower beds (they keeping the dog moving forward, etc.)[5].

---

5. The reader may ask why keep-dog-out-of-street isn't broken into watch-for-street and reel-in-leash-near-street, in the same manner as keep-dog-out-of-flowers? It is an arbitrary decision I am making here mainly for purposes of discussion.

The **street** in cross-street is different than the **street** in keep-dog-out-of-street. Cross-street's **street** is bound to the crosswalk at the corner, i.e. the area between the two curbs. This **crosswalk** is shared between the watch-for-cars-at-crosswalk and cross-crosswalk-when-clear tasks because the agent should cross the same crosswalk that it has just determined to be free of **cars**.

Walk-dog-to-park is similar to walk-dog-to-corner in that the **park** role is shared between the parent and the walk-toward-park subtask. The **street** is, again, not shared with the other subtasks because the other subtasks are concerned with different aspects of the dog's motion.

The final task in walk-the-dog is play-fetch-with-dog. The **dog** is again a shared role between this task and its subtasks, as is the **ball**. The agent must throw the ball and retrieve it from the dog (and it ought to be the same ball in both cases).

Throughout this chapter I have mentioned that the position of entities bound to task roles is the information that the agent's tasks need. This is because the agent's tasks involve moving itself and the dog along a certain path and not getting too close to particular points (e.g. flowers). Therefore, position is the information that should be computed for task roles.

Now the designer must consider which of these shared roles should have some explicit form of representation. Since the **dog** role is shared among most of the tasks and subtasks in this domain, I argue that it should be represented. The dog's position is important for all the top level tasks. Since the dog will be moving all around the agent while the agent is monitoring entities bound to other task roles, such as the **street** and the **flowers**, having a **dog** representation will allow the agent to remember the dog's position even when the dog cannot be seen [14]. Information indicating the dog's position must be shared by all the

subtasks of walk-dog-to-corner and walk-dog-to-park. Providing an explicit representation of the dog's position not only allows for sharing among parallel subtasks, but for passing of **dog** information between the subtasks.

In addition to the **dog** role, the **corner** role and the **park** role in the walk-dog-to-corner and walk-dog-to-park tasks need to have some form of representation. The position of the entities fulfilling these two task roles are used to drive the agent's locomotion system. Again, since the agent will at times be monitoring the **dog**, the **street** or the **flowers**, it will not always be directly perceiving the **corner** or the **park**. The "limited field-of-view" argument [14] for representation applies here and is discussed in section 3.5.2. Basically, the agent cannot perceive both the street and the flowers at the same time because they are on opposite sides of the sidewalk and the agent's vision system only covers a limited area. However, the agent cannot forget about the existence of one of those entities, just because it is looking at the other.

The case for representation of the **flowers** and the **street** in the keep-dog-out-of-flowers and keep-dog-out-of-street tasks is similar, since the positions of these entities are used in leash effector control. Detecting these entities and tracking their positions facilitates the computation of geometric properties (such as proximity) between the entities and the dog. While this is consistent with psychological literature, e.g. [19][63][76], the most compelling reason to represent these entities is the limited field-of-view argument [14].

The **street** in the cross-street task may not need representation. As stated in section 3.4.1, the **street** is actually determined by the positions of the curbs on either side of the crosswalk area. For the watch-for-cars-at-crosswalk task, the agent needs to be able to see the entire crosswalk (i.e. both curbs, the area in between and some of the street before the crosswalk)

at the same time in order to make sure no cars are coming. Since all important entities are in view for the duration of the task, they need not be represented explicitly. However, for the cross-crosswalk-when-clear task, it may be useful to have a representation for the curb on the other side of the street (the side where the park is) because the agent can use this to servo on. Since this task runs in parallel with keep-dog-close-by, the agent may have to divert its attention to the dog and no longer be able to perceive the curb (again, the limited field-of-view argument). So, the agent need not represent **street** because no information about the street is needed by later tasks. All that needs to be represented to get the agent across the street is the crosswalk, which will be associated with the curb on the opposite side from the agent.

It may seem that the **ball** used in play-fetch-with-dog also does not need representation, as long as the dog actually fetches it each time. This is because throw-ball-to-open-area assumes the ball is in the agent's ball throwing effector and get-ball-when-dog-returns always puts it there. Since the dog presumably carries the ball in its mouth, knowing the dog's position is sufficient to find the ball during get-ball-when-dog-returns. If the dog fetches the ball, the agent need not be concerned with where the ball lands after it is thrown and so none of the play-fetch-with-dog subtasks need to store any information about the ball. However, if the agent does not begin the walk-the-dog task with the ball in its launching effector, but say in a carrying pouch, a ball representation might be useful to store whether the ball was in one of the known fixed locations (pouch or launcher) or "thrown". The agent need not represent the **open-area** because the dog will go there and get the ball. Once the ball is thrown, the agent does not need to know any more information about that area (and in fact, a new open-area will be selected on each loop of the fetch task, so information stored about

any particular open-area will not be useful for long).

Now the designer must consider the rate of representation verification. Since the agent uses the positions of entities for effector control, the agent needs to verify the positions at a rate as close as possible to the rate of effector control. This rate is needed so that changes in important environmental entities will be detected and responded to as quickly as practical. Consider the dog's position, which as mentioned above, is the information that must be known about the entity associated with the role called **dog**. The agent executing the keep-dog-out-of-street task might, for example, pull on the leash with strength proportional to the distance between the entities associated with the **dog** and the **street** representations until those entities are far enough apart.

Keep-dog-moving-forward might involve monitoring the dog's position and pulling on the leash until the dog is in front of the agent. The effectiveness of the agent's leash control depends on effective monitoring of the dog's position. Since the positions of the entities associated with **corner** and **park** are also used as feedback in effector control loops, this information should also be verified at the same rate that these loops issue effector commands. Verification of the ball's position can be done much less often since the position is either one of two fixed locations, or "thrown" and the transitions between these states take place at well-known times (since they occur only by the agent's actions).

If the agent has the computational and perceptual resources to verify the information in all representations at the rate of effector control, then the verification design is done. However, most agents have only a limited perceptual field and so, regardless of computational resources, cannot verify information stored in all representations because they cannot perceive all relevant entities. Now the designer must make trade-offs between timeliness of in-

formation and the cost of shifting perceptual resources to different areas of the environment.

In the walk-dog-to-corner subtask there are several parallel activities. Recall from figure 1 that as the agent is walking toward the corner, it must keep the dog out of the street on one side and the flowers on the other. The agent has to switch between verifying the position of the entities associated with **corner**, **dog**, **street** and **flowers**. Since the position of the corner, street and flowers only change (with respect to the agent) due to the agent's own motion, the agent can estimate their locations with its proprioceptive sensors. A reasonable algorithm might be to update the position of the flowers when the dog is on the right side of the agent and the street when the dog is on the left side. Obviously the dog should be monitored in both cases.

Since the bindings of both the **street** and **flowers** will change as the agent moves, it is not necessary to monitor the associated entity's position when the dog is on the opposite side of the sidewalk (since whatever entity is currently bound to the role will likely be unimportant the next time the dog crosses over there). The agent must also periodically check its position relative to the corner. This can be done opportunistically, when the dog walks in front of the agent (when both the dog and corner are in the field of view) or based on the accuracy of the proprioceptive sensors (when the dead-reckoning is known to become inaccurate).

For the cross-street task, the agent can begin by trying to bind **car** to any large object moving toward the crosswalk. After a suitable length of time, if **car** has not been bound, the agent can stop trying to bind it and begin moving across the street. If the agent has a representation for the curb on the other side of the street from the corner (the other side of

the crosswalk) then there is a trade-off between monitoring the dog and watching the servo point used to cross the street. If not, the agent might just move forward one street width and watch the dog.

Walk-dog-to-park has the same trade-off between verifying information about the **dog** and the **park** as between the **dog** and the **corner** in walk-dog-to-corner. Again, the agent can update the park's position opportunistically or when dead-reckoning becomes inaccurate. During play-fetch-with-dog, the agent does not need to monitor the position of the ball since the dog will retrieve it. In fact, the agent may not even monitor the dog since it is going to come back with the ball. In this case, the agent just needs to perform a search in its local area to determine when the dog has returned.

3.5.2 Representation Rationale

Task roles are variables that become associated with (bound to) entities in the world, much like variables are bound to values in a program. For some roles it is useful to "save" this association rather than recompute it whenever it is needed. Saving the association refers to having an internal representation, i.e. a data structure, that stores information about the entity bound to the task role. Representation can be a powerful tool for an agent designer because it can be used to share data between tasks and provide perceptual continuity over both time and distance [13]. We examine the sharing of roles between tasks, since when multiple tasks have roles that should be associated with the same object, it makes sense for them to share a single role (and its binding) instead of each maintaining their own binding. Role sharing is a principle reason for having an explicit[6] representation for a role. I will now discuss the advantages and disadvantages of representation in detail.

The creation and maintenance of role/entity bindings is a large part of the computational

load of the sensor/effector control component of an agent's architecture (recall the discussion of PA systems in section 1.3.2). The designer can reduce this load by taking advantage of multiple tasks that use the same binding (so each subtask need not go through the binding process). Often parallel subtasks in an emergent behavior scheme [16] will each effect the same entity. The pour-a-cup-of-coffee task requires that the pot be aligned over the cup and tilted at the correct angle. The tilting and aligning tasks occur in parallel and each refers to the same pot.

Task role/entity bindings are also shared between sequential tasks. The pour-a-cup-of-coffee agent must grasp the coffee pot's handle, remove the pot from the machine and then do the alignment/pouring task. The **pot** role should be bound to the same object in each of these for the overall task to make sense. After the first task makes a binding, that binding should be used by the subsequent tasks.

One caution to the reader is that this thesis uses consistent names for all roles that are shared between tasks. In other words, when two tasks share a role the role has the same name in each task. However, the designer must often do some abstraction to see when multiple tasks have roles that can be filled by the same entity (and hence those tasks can share the binding). Imagine that the skills for the pour-a-cup-of-coffee agent are being designed by different teams. If the skill that removes the vessel with the coffee from the coffee machine represents that coffee container with a role named **pot**, while the skill that pours from the vessel into the cup represents the coffee container with a role named **kettle**, the designer must make sure that **pot** and **kettle** are associated with the same object. As another exam-

---

6. Note that even the implicit roles will need to be taken into account in the coding of the agent's various behaviors. That is, the implicitly represented roles will still be use by the agent to decide on its actions. A task will calculate some information about the entity fulfilling an implicit role, it just won't store it.

ple, imagine that the pour-a-cup-of-coffee agent designers are borrowing a skill from a previously created agent that poured a glass of juice. This juice pouring agent had a skill called pour-from-pitcher-to-glass that had **pitcher** and **glass** roles. The pour-a-cup-of-coffee agent designers might use this skill by sharing the binding of the **pot** role in remove-pot-from-machine with the **pitcher** role in pour-from-pitcher-to-glass[7].

The decision of what information about a particular entity is important for the task obviously depends on the needs of the task. In the walk-the-dog example, positions of entities are important because all the agent's actions are based on the proximity of various objects (e.g. How close is the dog to the street? How close is the dog to me? Is there a car coming at me?). In fact, I argue that position is *the* fundamental piece of data that needs to be stored (and kept up-to-date) for every represented entity in the domains of interest to this thesis. This is because, as stated in section 1.3.1.1, agents designed by this methodology are meant to act in dynamic domains where current sensor data is more relevant than previous sensor data. The tasks addressed by this thesis involve robots navigating around and reacting to changes in their environment. In order to determine anything about the environment, an agent first needs to know where to look, i.e. the position to direct perceptual resources, and then other properties can be determined. This is not to say that some tasks may not need other data about entities. For example, in the drive-to-the-store task of task specification 3, when the agent is stopped at a traffic light, it must monitor the color of the signal. The agent's brake effector control process may monitor the color of the object associated with the **traffic-signal** role and release the brakes when it turns green.

Once the designer has decided what information about an object associated with a role is

---

7. This is analogous to software reuse [74].

needed by the task, role representation must be considered. Role representation has several

benefits. First, representing a task role allows the agent to deal with a limited field-of-view

[13]. The limited field-of-view argument for representation says that important entities that

can be outside the agent's sensory field during a task should be represented so that the agent

doesn't just forget about them. For example, the walk-the-dog agent's vision system has a

limited field of view and so cannot perceive the roses and the street simultaneously. By rep-

resenting the position of the flowers and the street, no matter which one the agent is looking

at, it can react (based on the stored data) if the dog suddenly bolts for the other.

Note that not all information that a task needs to know about an entity must be stored in

that entity's representation. The **traffic-signal** representation used by the drive-to-store

agent might just store the signal's position so the agent can find it again if the agent looks

away while waiting at the red light (to change the radio station perhaps). The signal color

may not be stored since once it turns green, the agent releases the brake and is no longer

concerned with the signal.

Another advantage of representation occurs when a role is shared. Role sharing comes in

two different forms. First, when multiple tasks act with respect to the same entity, repre-

sentation allows those tasks to share the role data structure (and hence the cost of creation

and maintenance). Another form occurs between the dependent roles identified in section

3.4. This form of role sharing refers to sharing information about the entity bound to a role

because it effects the information the agent has about entities bound to other roles. Repre-

sentation can be useful here because the tasks that manipulate the data structures of depen-

dent roles can reflect any new information in these roles representation with a minimum of

communication with other tasks. If the roles are not represented, each time a task T wants

information about an entity bound to a dependent role, it must gather data from the other tasks with roles on which task T's role depends. Depending on the architecture, this communication can be expensive. Some roles represent more abstract properties of the environment than one or more other roles on which they depend. For example, the **bowl-of-apples** role may store the fact that the entity associated with the **apple** role is inside the entity associated with the **bowl** role, while the **apple** and **bowl** roles themselves just store the positions of those entities. These dependent roles should be represented because these more abstract properties are often expensive to compute and so the agent would like to be able to store the result for some time.

The designer should not think that representing all task roles is beneficial. There are costs associated with representing and not representing. These issues are at the heart of the questions of what information about the bound entity should be represented and how often that information should be verified. Proponents of the reactive approach argue that the world is its own best model [15] and so nothing needs to be stored internally. Information can always be determined by consulting the world when necessary. Implicit in this argument is that the relevant entities can always be kept within sensor range, and if this is true for a particular task, the designer may consider not representing that task's roles. However, always computing a piece of information from the current sensor data requires the continued allocation of perceptual and computational resources. The more information stored in a representation, the more information the agent has access to when the associated entity is not in viewable, but the more expensive the maintenance. The agent must strike a balance between the amount of information stored and the rate at which it is verified since either can overburden the agent's finite computational power. In much the same way that deciding on

task roles caused the designer to focus the agent's tasks on only what was currently important, the selection of what entity information should be stored and how often it should be verified serves to focus the tasks even more.

The designer should note that sometimes the agent's capabilities obviate the need for representation of a role. For example, if the dog walked in the walk-the-dog task had a GPS transmitter in its collar and the agent had a GPS receiver, the dog might not need representation because it would effectively always be in sensor range.

Deciding how often information needs to be verified is also task dependent. There is a time/usefulness trade-off between a high rate of maintenance (and thus high usefulness of data) and computational load. Recall that the "rate of maintenance" refers to how often a piece of information is verified. So, the higher the rate of maintenance on a piece of data, the more useful that data is, but the more expensive it is in terms of perceptual and computational resources. Note that this maintenance need not be periodic. For some information, it may be sufficient to verify it whenever the agent can schedule that computation in with its other jobs.

Traditional reactive systems [17] place emphasis on usefulness of information and so pay the maximum computational price. However, this methodology advocates examining the agent's task and capabilities to determine acceptable rates of maintenance. In fact, there exists the possibility that certain pieces of data cannot be verified given the current sensory view. This is where the trade-off becomes a more complex decision. If all entities associated with represented task roles are always within the agent's perceptual field, then the rate of verification presents only a computational cost. However, if an entity can move out of the perceptual field (either through its own motion or the agent's) the agent must direct its

perception system to a different location in order to verify information about that entity. This means that other important entities may no longer be in view and so their information cannot be maintained at any computational cost.

Maintenance can be arbitrarily complicated, possibly involving planning. Going down-town to verify that the grocery store is still there is an extreme example. So, the designer must create a "maintenance scheme", i.e. a procedure for keeping representation verified at required rates, that balances task needs for up to date information, computational cost, and limited sensory horizons (areas perceivable by a sensor at any one time).

Two other factors that are important in representation maintenance are "opportunism" and "effector vs. non-effector control". "Opportunism" refers to the agent happening to get the entities associated with multiple task roles within its perceptual field simultaneously. The agent should be designed to take advantage of this, such as when the dog walks in front of the dog-walking agent and the agent can see both the corner and dog in the same view. Both representations can be updated at the same time.

"Effector vs. non-effector control" refers to whether or not a piece of information stored in a role is used by that role's task to directly control effectors or for some other purpose. The reel-in-leash-near-flowers task uses the position of the dog and the flowers to directly compute commands for the leash effector (how hard to pull and in what direction). Other pieces of information that the agent may compute about the entities associated with task roles may not be used by effector control tasks. Deciding if the pot has enough coffee will determine if the pour-a-cup-of-coffee agent should even proceed with the pouring action, but does not directly influence the motion of an agent effector. This sort of non-effector control data will typically need to be verified much less often than the effector control data

and so the agent designer should bias the maintenance process so that most of the agent's time is spent verifying the effector control data.

Since the majority of the agent's maintenance computation will be devoted to effector control data, the role representations used by the leaf tasks should store no non-effector control information. For these tasks, non-effector control data is only excess baggage whose maintenance will slow down the task's operation. In order to understand this, consider that all information stored in a role representation will need to be verified at some point. If it doesn't, then the information is unaffected by the dynamic nature of the environment and there is no benefit to explicitly representing it. Instead, the agent designer should design that knowledge into the agent's tasks with an implicit role. So, since all role data needs to be verified, and the rate of verification determines the usefulness [14] of the role information to its task, effector control tasks should not expend effort maintaining non-effector control information.

It is perhaps not obvious that representation should be deleted when a task completes, providing the represented role is not shared with another task. Once a representation is no longer useful, the agent should not maintain it, and so deleting these representations reduces the agent's maintenance work. Even when a role is shared between tasks, the designer may want to have the agent delete that role's representation if the sharing tasks are not executed closely in time. If the work required to recreate and re-bind the representation is less than the work to maintain the representation in between tasks, it makes sense to delete the representation.

3.5.3 Representation Summary

The designer must decide which task roles should be explicitly represented in the agent's

control program. The two main reasons to have an explicit representation are to provide information about important entities outside the agent's current perceptual field and to share data. These address domain characteristics 3 and 5 (see sections 1.3.1.3 and 1.3.1.5) because the former provides a symbolic model that other tasks can use to help control the agent and the latter provides a means to deal with large-scale space [43].

I have argued that a piece of information that must be stored in every role representation is the position of the associated entity, if for no other reason than to direct the agent's perception system to the entity to determine other information. This takes advantage of domain characteristic 6 because the agent can (at least coarsely) estimate positions. The designer must also decide on an appropriate representation maintenance scheme. There must be a balance between the usefulness of the stored information about each currently relevant entities and the amount of computation involved in establishing that information.

3.6. Perception

What information can the agent extract from the environment to recognize the entities that should be bound to the current task's roles? The agent's perceptual capabilities will form the baseline for what it can identify in the environment. Those capabilities and any task dependent information must be encoded into perceptual descriptions of task roles and associated recognition processes. The term "primitive recognizable", or PR, is used to denote any entity from the set of entities that the agent will match *directly* to a given perceptual description. "Directly" means that object recognition is based on only the agent's given perceptual capabilities and not on any combinations of, or relationships between, separately recognized entities.

How does the duration of the various role/entity bindings effect the perception system?

What level-of-detail (or resolution) is required in the information of the representation?

Resolution refers to the precision of the data stored in the representation.

3.6.1 Example Task Perception

For simplicity of exposition, I have specified that the dog walking agent has the perceptual capabilities to directly identify most of the entities that can fulfill its task roles (see task specification 1). The dog, the ball, the corner, the park, flowers, curbs and cars can all be directly recognized by the agent's perception system and so are PRs. The street, however, must be found in relationship to curbs. From figure 1, we can see that the keep-dog-out-of-street task can be accomplished by keeping the dog away from the closest curb, which can be identified by the agent's sidewalk edge detector. Also, flowers come in two types, roses and tulips (see figure 1) and so the agent must be able to recognize either of these. The watch-for-flowers task will involve looking for roses, then looking for tulips, then back to roses, etc. The perception system will need to handle multiple perceptual descriptions for the same role.

It is tempting to say the **open-area** in throw-ball-to-open-area can be recognized by the lack of any of the entities that the agent wishes to keep the dog away from, i.e. no flowers, curbs or cars. However, such "negative" recognition processes are problematic because even if the agent finds no object that conflicts with the **open-area** description, there may be other entities present. The agent does not want to throw the ball into an open pit just because there are no flowers, curbs or cars there. These negative processes are similar to PROLOG's negation as failure model [68] because when the agent can not determine that any known objects is in an area, the area is determined to be "open". These kinds of prob-

lems pervade perception systems on autonomous agents and I do not seek to eliminate them here. If an agent's perceptual capabilities do not allow it to identify necessary entities in the environment by another means, then the agent's capabilities are deficient and the best the designer can do is be aware of the issue.

Next, the designer must examine the duration of the role/entity bindings for the roles involved. Clearly the **dog** role has the longest binding duration (the length of the task). The **corner** and **park** remain bound for the length of the walk-dog-to-corner and walk-dog-to-park subtasks. However, the **street** and **flowers** bindings change as the agent moves down the street. The **crosswalk** in the subtasks of cross-street remains bound for the duration of those tasks as does the **ball** in the subtasks of play-fetch-with-dog.

The designer would like to take advantage of the fact that the **dog** binding lasts for so long by "foveating" the dog's position to speed up the perceptual computation that determines its position. "Foveating" refers to processing only a limited portion of the input image around the entity's (last known) position. Since the **dog** should always correspond to the same dog, the agent need not look over a large area for other potential dogs, once the initial binding is made. The same is true of **corner** and **park**. However, the fact that the environment is dynamic and all these entities cannot be kept in view simultaneously may cause this strategy to be modified slightly. When the agent looks away from the position of an entity for some time, it may have to examine a larger (or entire) portion of the perceptual stream when it returns to view that entity because the estimate of the entity's position may have become incorrect and an expanded search may be necessary.

The **street** and **flowers** roles cannot be foveated since those roles can potentially be rebound to other flowers or parts of the street as the agent and dog move. However, since the

crosswalk in the cross-street task remains bound to the same entity (the curb) for the entirety of the task, it can be foveated and tracked. The ball does not need to be foveated because the agent does not have to continually track its position (since its position is stored as "in pouch", "in launcher" or "thrown").

Finally, the designer must address the level-of-detail needed in the representation's information. For most of the task roles, the agent needs to know the position of the associated entity. When that role is a landmark (**corner**, **park**, or **crosswalk**) the accuracy of this position value can be less than the accuracy of the position stored in roles used for leash control (**dog**, **street**, **flowers**). This is because when starting toward any landmark, only the general distance and direction are useful. The agent might dead-reckon the position of a landmark, meaning the accuracy of the position estimate will decrease. Since coarse metrics are effective in this domain (as mentioned in Section 1.3.1.6), provided the agent periodically verifies its estimates with its vision system, it can still navigate to the landmarks. However, since the dog moves of its own accord and the **street** and **flowers** can be bound to different entities at any time, estimating their positions with proprioceptive sensors is risky. Of course, the agent will have to do just that if, say, the dog is not in the same view as the corner, but the **corner**'s location needs to be verified (such as when the agent believes its estimate of the corner's position can no longer be trusted based on its understanding of the limits of its dead-reckoning capabilities). Lastly, as mentioned above, the ball's position is stored as one of three states, which is a fairly course level-of-detail. This level is acceptable because of the agent's simple ball launching system.

3.6.2 Perception Rationale

This step seeks to impart some structure to the agent's perception system based on the

agent's task and capabilities. Ultimately, the agent's perception system is charged with the creation and maintenance of the bindings between task roles and entities in the environment. The computational demands of maintaining task role information in a dynamic environment means that the designer must carefully structure the perception system to be efficient. The first step in this process is to consider how the task specification and agent capabilities determine what information the agent can (and should) extract from the perceptual stream to recognize entities to be bound to task roles. In some ways, this is both a top-down and a bottom-up process like creating the task decomposition.

There are some roles for which perception must come top-down from descriptions of particular objects that can fulfill them and the agent must somehow use its capabilities to detect this description. In other words, the walk-the-dog agent cannot associate the **dog** role with the mailbox just because it can easily detect the mailbox. On the other hand, perception for roles such as **car** in watch-for-cars-at-crosswalk can be thought of in a bottom-up manner. Any capability the agent has can be combined to find "cars". If the agent has a motion detector, then its reasonable to use it to identify **car**s, and a "bumper detector" would work if there were no parked (non-dangerous) cars. Some iteration with the task decomposition and role selection steps may be necessary if the entities that can possible fulfill a role are not detectable by the agent. Ultimately, if the agent cannot identify objects that it must act on for its tasks, then the agent's capabilities are insufficient.

As another example of how task and capabilities determine what perceptual data should be used in the recognition process for a role, consider the whack-a-mole agent of task specification 4. This agent has the ability to detect and localize motion as well as determine color. So, the agent could identify a mole as a brown object protruding above the game surface.

Alternatively, the mole could be any fast moving object within the game area. Since the task specifies that the game contains only moles (all of which should be whacked), the later perceptual description could suffice, but not if the motion computation cannot proceed at the rate that the moles pop up and down. Note that in either case, each mole is a PR.

The designer may need several recognition processes to detect the entity that should be bound to a role, for a variety of reasons. Some task roles are not filled by single objects, such as the **bowl-of-apples** role that is fulfilled by a bowl and some

Task:	play whack-a-mole game ("mole" pops up from set of holes & agent must hit it before it goes down)
Environment:	game surface with 6 holes, brown moles below each hole
Agent:	positionable mallet effector to hit moles, perception system can detect motion and color

Task Specification 4. Whack-a-mole

apples with the correct spatial relationship between them. The designer will have to have thought about combinations and relationships among the objects that fulfill roles when identifying dependent roles in section 3.4. For example, consider the agent of task specification 5 whose task is to pick up a covered pot from the stove. The agent may have a specialized "pot with lid" detector that can identify the vessel and cover as a single object, or PR. On the other hand, if the agent can identify the pot and lid as separate PRs, the **pot-with-lid** might be identified as a **pot** with a **lid** above it. If the agent could determine the position of the pot's handles, then for the pick up task the agent could identify the pot by the location of its handles (since handle position is the necessary information for that task's effector control).

There are other times when multiple perceptual routines are needed because the perception of all objects, and particularly these compound objects ("objects" consisting of individually identified entities) is situation dependent. Consider a lamp and a lightbulb. The

agent may be able to identify each individually, but when the lightbulb is screwed into the lamp's socket, the "socketed" relationship cannot be perceived except from an overhead or underneath perspective. Similarly, a penny on top of the Empire State building would appear very small when perceived from the ground, if it could be seen at all.

Another important issue is that objects can change their perceptual characteristics when assembled. A knife and a wooden knife-set holder have one appearance when separate, but when the knife is put in

Task:	pick up a covered pot from the stove
Environ-ment:	stove with several pots, one to be picked up has 2 handles and a lid
Agent:	two grasping effectors, vision system

Task Specification 5. Pick-up-covered-pot

the holder, only the knife's handle is visible. A sugar cube dropped into a cup of water looses its cube shape. Of course, these two categories of perceptual difficulties are not so different. Both problems are the result of the fact that objects look different in different states. A "different state" may result from the object being at a different location (the penny), in a different spatial configuration (the light bulb), partially occluded (the knife), or being physically altered (the sugar cube). The effect of all this is that the agent may need multiple perceptual descriptions of an object to recognize that object in different situations.

Once the designer has decided how entities associated with task roles can be recognized, two necessary optimizations must be made by considering the effects of role/entity binding duration and required precision of stored information. These "optimizations" are vital to operate effectively in a dynamic environment.

We examine the duration of the role/entity binding because if the agent needs to monitor some entity that is bound to a task role for a long period of time, it may be advantageous to foveate the entity to detect changes in entity properties needed for tasks. For many agents,

e.g. [4][8], this means tracking the entity's position. Reducing the area of the perceptual field that must be searched, decreases the time needed to determine the needed entity information. This increases the potential update rate for the entity's representation, which for tracking means the entity can move faster relative to the agent (and the agent will not loose it). However, if a role is bound to several different entities during the duration of the task, foveation may be less useful as the agent will still need to examine a (larger) portion of the perceptual field for other entities that could be bound to the task role. For example, suppose the play-wack-a-mole task consists of a single subtask, hit-whackable-mole. The **whackable-mole** role will be rebound to a variety of different moles during the game. Limiting the **whackable-mole** search process to a single mole hole will only allow the agent to hit one of the moles (and thus get a low score). There may be other cases where the agent both foveates the currently bound entity and examines a larger portion of the perceptual field for other entities that should be bound to the task role. For instance, the agent may implement hit-whackable-mole by foveating the current **whackable-mole** and monitoring whether it is still out of its hole. As long as the mole is "up", the agent should continue the hit action against this particular mole. At the same time, the agent is processing the current visual field for moles that can be bound to the **whackable-mole** role when the current mole has been hit or retreats into its hole.

Determining if a binding duration is "long" depends on the execution time of various tasks. Long binding durations arise in two different situations. One is when a single task with a long execution time binds a role that stays bound to the same entity for a large portion of that task's duration. The second way a role can have a long binding duration is for several shorter, sequential tasks to share a role that they all expect to be bound to the same

entity. Of course, "a long execution time" and "a large portion" are still ill defined, but basically if a role is bound and the agent then does not need to look for other, more appropriate entities to bind the role to, foveation of the target can save processing time.

The last question in this section addresses the fact that some tasks need more precise information about the entities bound to roles than other tasks. Some information can be determined from multiple sensors. For example, the position of an object can be determined by vision or by sonar. Some sensors provide more coarse data and so there is less data to analyze. Other sensors are more precise and still others have a larger perceptual field. All sensors can only detect certain environmental properties. Since sonar only provides positional information, data from a typical robot sonar system is much coarser than visual data. However, it typically covers a larger perceptual field. Proprioceptive sensors provide only a small amount of information, but the size of their perceptual field is not a concern. The designer needs to examine precision requirements to create a representation maintenance scheme. That is, a designer must determine when different sensors can be used to determine information and develop some strategy for allocating sensor resources. In the walk-the-dog example, the agent used a combination of proprioceptive sensors and vision to maintain its representation. This was done because the agent cannot use its visual resource to acquire all task relevant entities all the time and the lower accuracy sustained while an entity was out of view was manageable.

Answers to this question, along with the designer's decision on the required rate of maintenance (see section 3.5) define four basic classes of role maintenance requirements.
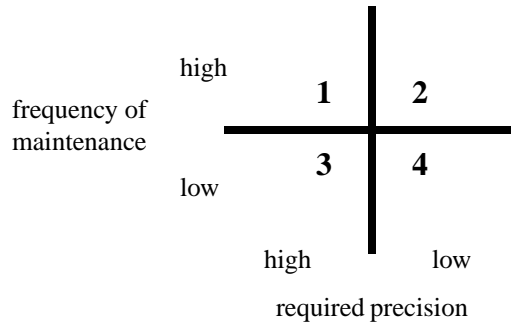
Figure 5. Categories of Role Maintenance Requirements

Figure 5 shows the space of role maintenance requirements divided into four quadrants based on precision and maintenance frequency. Exactly how a designer decides what constitutes "high" and "low" precision or maintenance frequency depends on the agent's task and will be relative to the needs of the various tasks the agent must perform. However, placing the needs of a particular role into one of these classes will help the designer create an appropriate representation maintenance scheme for the agent. Additionally, each quadrant can be further divided into the same four classes and the design decisions discussed below will apply to those finer divisions as well.

The four quadrants are as follows. Quadrant 1 contains roles that whose associated information needs to be verified frequently and whose information must be precise[8]. These roles put the most constraints on the agent's maintenance scheme and they require the largest amount of time for the high accuracy sensors. For example, the dog in walk-the-dog needs to have the camera trained on it as often as possible because its position needs to be verified frequently and the camera is the agent's most precise position-determining sensor. Quadrant 2 represents roles that need to be verified often, but do not require much precision in

_____

8. The information must always be *accurate*. That is, the agent will perform worse with inaccurate information than with correct information. The issue is, how exacting does that information have to be? When position is specified, in how small a region does the agent have to localize the entity.

the information they contain. Hopefully, the agent can just use another, less precise sensor to handle these roles as the most precise sensors will be tied up with maintaining the roles in quadrant 1. For example, the agent in chapter 6 avoids obstacles with sonar. Sonar provides coarse position estimates for obstacles (compared to the visual estimate of the position of the agent's destination provided by the agent's camera), but the avoidance task does not need very precise information anyway because it tries to circumvent obstacles and not squeeze through tights spaces. Roles in quadrant 3 are those that need to be updated infrequently, but need to contain precise information. Often these roles are ones that depend on other roles and hold an abstraction of the information the dependent(s) contain. For example, the **pot** role in the pour-a-cup-of-coffee task needs to store the pot's position in one task and the amount of coffee in the pot in another. The amount of coffee must be precise, to tell if another cup can be poured, but it need not be checked often (only before each new cup). Roles in this category will vie for time from the precise sensors with the roles in quadrant 1. The designer must create a scheme where by quadrant 3 role data can be occasionally computed, with the bulk of the time being spent determining quadrant 1 role data. Finally, quadrant 4 contains roles that neither need to be updated frequently, nor need to contain precise information. Although there may be tasks that have roles fitting into this category, the designer is advised to consider whether or not such roles really need (explicit) representation. Roles in this category may contain information that is needed by a scheduling or planning system and so while their information may need to be determined, it may not need to be held in a representation in the action-oriented portion of the agent's architecture. For example, a role that stores the answer to the question "is it raining?" might be in this quadrant if the agent believes that rain showers do not end quickly (and thus this data will not

need frequent updates since it will be true or false for some time) and if the agent is not concerned with how hard it is raining (so precision is not important). Instead of being stored in a representation, this type of information could be sent to a planning system and stored there, but this is beyond the scope of this work.

There is also a computational dimension to representation maintenance because computing arbitrary information can take arbitrary amounts of time. If the agent wants to verify some complex property of an entity at a high rate, with high precision, it must have the computational power to do so and the designer can do little to help if it does not. This brings up the question of effector vs. non-effector control information from section 3.5.2. The computational difficulty of determining some role information can effect all categories of roles. Roles in quadrants 1 and 2 would be effected by any computation that slowed down the rate at which the agent's resources were available to verify their data. Roles in quadrant 3 often store information that is expensive to compute (which is why they are updated infrequently) and the agent will need to schedule that computation so as not to interfere with other maintenance. Even roles in quadrant 4 can be problematic if their maintenance computation needs to be done in an environment where the loss of cycles to other role updates causes the agent to miss events. "Anytime algorithms" for determining the required properties of the environment allow the designer to make trade-offs between precision and computation time. The amount of time available can be allocated between the roles so that their information is at least of the desired precision relative to the other roles. Of course, scheduling of anytime computation is NP-complete [87] and this issue is not addressed by this thesis. In addition, compositions of system components with different anytime algorithms does not guarantee that the system as a whole behaves like an anytime algorithm [87].

3.6.3 Perception Summary

In general, the designer will want roles in the leaf tasks of the task decomposition to be fulfilled by PRs because these can more easily be maintained at rates commensurate with the effector control loops that implement these tasks. This may cause the designer to iterate between deciding which entities can fulfill a task role and which entities are PRs.

In this step, the designer must decide how the agent's perceptual capabilities can be used to recognize entities that can fulfill various task roles. For the represented roles of the leaf tasks of the decomposition hierarchy, it is important that the entities bound to the roles be PRs so that their representations can be maintained effectively. The duration of role/entity bindings can allow the perception system to "foveate" the area around a bound entity and thereby decrease processing time. Also, the level-of-detail required for particular data allows the agent to allocate its resources to where they are most needed. The designer may have to go back to a previous step of the methodology (perhaps to the task decomposition or the selection of task roles) and try a redesign if all the current representations cannot be maintained at the required precisions/rates. All these considerations are meant to address domain characteristic 1 (see section 1.3.1.1) since that characteristic places demands on the agent's computational power and thus its ability to react.

3.7. Communication

What information is important in inter-task relationships? In other words, what data, computed by one task, is needed by another? In Section 3.5, roles that are shared between tasks were identified and so role information is clearly be exchanged between such tasks. At this point it is important to also think about data other than the information stored in role

representation (from section 3.5) that tasks may need to exchange. The results of (the actions of) previous tasks or the progress of other tasks in the system are examples.

3.7.1 Example Task Communication

In the walk-the-dog task, there are several kinds of information that are communicated between tasks in this domain. First, there is the information about the environment that the roles store (much of which has been mentioned in various descriptions of the tasks). Tasks obviously communicate position information for the entities associated with their roles. This communication takes place between parallel executing tasks that all base their actions on that information (such as between the subtasks of walk-dog-to-corner and walk-dog-to-park) and between sequential tasks such as depicted in figure 3a.

When the first subtask of figure 3a (walk-dog-to-corner) is executing, it maintains the position of stored in **corner**. When the task is complete, this information is communicated to cross-street to help establish the initial position of the crosswalk. Cross-crosswalk-when-clear reports the final position of the crosswalk (stored in **crosswalk**) to cross-street upon completion so that this information can move on to walk-dog-to-park. There it is used to determine the agent's trajectory to the park. **Dog** is maintained by many tasks over time and when each completes executing, that role information provides the next task with the position to continue monitoring the dog.

Tasks also communicate both perceptual descriptions and binding information for the entities to be bound (or that have been bound) to their roles. In this domain, many roles can be bound to only one entity and so a perceptual descriptions of the entity could be built into the tasks. However, perceptual information is still communicated. Consider the **flowers** role. Watch-for-flowers will bind the **flowers** role when flowers are in sight, and a percep-

tual description of those flowers will be communicated to reel-in-leash-near-flowers. This is useful because, as shown in figure 1, the tulips are farther back from the road than the roses and so the dog can be given more leash near them. The watch-for-cars-at-crosswalk task needs to communicate the fact that the **car** role has not been bound to an object (after trying for some time). This triggers cross-crosswalk-when-clear. As another example, suppose the agent were programmed with descriptions of several dogs that could be walked. The walk-the-dog task would pass a description of the current dog to all its subtasks, which would in turn pass the description to the leaf tasks. In one of these, the binding between the dog and **dog** will be created and subsequent tasks can use the perceptual description to maintain the dog's position (although the agent may examine a smaller portion of its sensory stream).

In addition to passing information about the entities bound to task roles between tasks, the tasks exchange some notion of the confidence they have in that information. For example, if the keep-dog-out-of-street task has been observing the **dog**, the confidence in the position stored in that representation will be high. This may allow the watch-for-flowers or walk-toward-corner tasks to redirect the agent's vision system to try and verify the **flowers** or **corner** roles. Confidence is a function of the time since the agent has acquired the role's entity with its vision system and provides a means for implementing the representation update algorithm discussed in section 3.5.1.

Tasks also communicate some data to indicate a task's action's result. Most of the agent's tasks just need to report success or failure and the next task in the sequence needs to decide what to do with "failure". I have not specified what happens if any task "fails", except for the is-dog-tired task, for which failure of its test means it should continue playing fetch.

However, imagine an agent with more sophisticated abilities to deal with failure. Now action results can be used to determine which task to execute next. For example, the agent could walk the dog to the corner, but the dog could break free and run away. The agent should not continue across the street, it should try and get the dog back. Note that the position of the corner alone was not sufficient to determine the success or failure of the walk-dog-to-corner task. The agent reached the corner, but it no longer had the dog.

Progress reports are another communicated item that represents a task's estimate of how well it is proceeding toward its goal(s). This form of communication often occurs between tasks executing in parallel. For example, avoiding the mailbox on route to the corner (see figure 1) is accomplished by the walk-toward-corner task. When the agent is in the process of avoiding this obstacle, the task notifies keep-dog-out-of-street that the **street** should not be in view and watch-for-flowers that the **flowers** should be or vice versa (depending on the direction of avoidance). Both walk-dog-to-corner and walk-dog-to-park should tell keep-dog-moving-forward when they are avoiding obstacles so that the dog is not forced into an obstacle.

3.7.2 Communication Rationale

The purpose of this step is to help the designer structure the task role representation. The information communicated between tasks should be part of the role representation because it is either about the entities bound to the role or about actions taken on those entities. There is an intimate relationship between actions and entities because actions take place on objects in the world, not arbitrary points in space. Of course humans are capable of taking actions seemingly without any relation to objects in the world (e.g. grasp at the air), but these are hardly the sort of actions that allow robots to do useful work. Actions do not make sense

if they do not effect the physical world and roles are action-specific descriptions of the physical world (this stance is softened somewhat in succeeding chapters for situations where the agent's sensors cannot detect the needed entity). So, by examining the types of entity information that tasks exchange to support their actions, the designer can determine what data needs to be in the role representation[9].

The designer can determine a structure for the role by examining what is communicated between tasks. Obviously, the information about the entity associated with the role (from the questions of section 3.5) will be communicated because it is what tasks need to know to accomplish their action(s). In addition, there are four principle classes of role information that tasks use and may need to communicate. They are *perceptual descriptions*, *actions*, *confidence* and *progress*. These classes of role information can be summarized as follows.

Perceptual descriptions are routines used by associated recognition processes to examine the sensory stream for entities that possess the "described" characteristics. These entities can fulfill the role. Such descriptions are often exchanged between parent and child tasks. Tasks, particularly leaf tasks, are often made "reusable" by allowing them to successfully operate with a number of different entities bound to a role. The pour-a-cup-of-coffee agent needs not have separate skills to manipulate mugs and tea cups, if it can operate with the **cup** role bound to either. The parent task typically has the information to decide what entity in the environment should be acted on by the child task and can therefore create an appro-

---

9. Admittedly, there is an engineering issue here as well. Since most of the information that is communicated between tasks is role information, it is efficient to add information to the role representation that is perhaps not strictly relevant to the role, but rather to the actions being taken on the role. For example, action requests, results and progress are information about the internal state of some process executing an action, rather than information about some entity in the world. However, since roles only exist to support the execution of these acting processes, it is not such a leap to say that the roles should carry all information that need be communicated between tasks.

priate perceptual description and communicate it to the child. The child task will either bind the role (if it is a leaf task) or create a new description appropriate for its child task. Consider the pick-up-the-covered-pot agent of task specification 5. The pick-up-covered-pot task might pass a perceptual description of the **pot** to the first of its subtasks (grasp-the-pot) and that task may break down the pot description into PR descriptions, such as **handles**, which are passed to the grasp-pot-handles leaf task.

Since there is necessarily a set of entities that match a perceptual description, a child task may report information about the actual bound entity back to the parent. For example, the pour-a-cup-of-coffee agent may report whether the **cup** role was bound to mug or a tea cup, so that the parent task can determine if there is enough coffee in the pot for that type of vessel. In the walk-the-dog agent, reel-in-leash-near-flowers and watch-for-flowers communicate the actual binding to the **flowers** role (to the tulips or roses) to determine the amount of leash the dog can use.

Actions are role information about the action to be performed and the results of that action. A role's representation may or may not contain the "action to be performed" portion of the action information, depending on the implementation of a particular agent. For example, an agent may possess a skill that is capable of performing multiple actions on the roles that are passed to it. An agent with a grasping system might be able to hold a cup by the handle (to drink from it), by the cup itself (to wash it) or by the cup's base (to invert it into the dishwasher). Agents without such multi-purpose skills may not need such action information in the role representation. However, most skills require some form of parameterization based on the agent's current state (see chapters 4, 5 and 6 for many examples). Action parameterization can be complex, especially when coordinating multiple effectors.

For example, an agent with two arms that has to shovel snow may require separate parameters for actions to be executed by each arm effector so that snow could be thrown to the left or to the right.

For any action, the role representation should also contain the results of the action just performed so that the next task can determine the outcome of the previous task. While it may be possible for a task to determine the results of previous actions by examining the state of the world (perhaps be examining the entity data stored in roles), the results of an action are dependent on the context in which it is executed and that may not be adequately captured by the role representation. For example, an agent that tried to open a door and failed because the door was locked may want to report this fact to other tasks because by observing the environment, those tasks can only determine that the action failed (since the door is still closed), but not why. It may be helpful to have an action result stored in the role representation even when the information can be determined by inspecting the environment because it prevents other tasks from having to execute elaborate perceptual routines to infer information that is known to the acting task. In addition, as we'll discuss in Section 3.8, doing a lot of inference can hurt the responsiveness of certain tasks.

Confidence denotes the agent's belief that the information stored in the role's representation is still useful [14]. Confidence is valuable for action because it helps determine how to allocate perceptual resources (to verify information in which there is low confidence) and select courses of action (to operate on the entity in which there is the most confidence). Confidence can be defined in a number of ways. When the role's associated entity is out of the agent's field of view, the designer may decrease confidence because the entity can move on its own and even if it did not, proprioceptive sensors are typically not as accurate

as direct perception. Confidence can also be expressed as a measure of how well an entity matches a role's perceptual description. For example, the drive-to-the-store agent may have a role for a particular street where it has to make a left turn. As the agent approaches the street, it may be too far away to make out the street sign. So, the agent might have only 50% confidence that the upcoming street sign is the correct street. Confidence is not just whether a role is bound (though it can be) because in this case, the agent had bound the street-sign to a role, but it may end up not being the sign the agent really wanted. In chapters 4 - 6, we will see several examples of how confidence can be implemented for agents with different capabilities.

Finally, *progress* is a measure of how a task is proceeding. Progress can be used by other tasks to monitor the communicating task. Tasks may adjust their actions based on the progress of other tasks. For example, the drive-to-the-store agent may have one subtask that handles the navigation and another that watches for the store. If the navigation process has only have a general idea of how far it is to the store, but not the store's actual position, it can report progress toward the store to the monitoring task. The monitoring task can then devote increasing time to looking for the store, as the agent draws closer.

It is, of course, possible for a designer to create tasks that communicate information other than that identified above as necessary for task inter-relationship. However, this information is extraneous and must be some "internal" state of the task. This data can only be about the task's internal computations and not about important elements of the agent's environment, otherwise it would be stored in the task's role representations. Such "internal" data should not be communicated between tasks because it provides no semantic information about the environment. Significant inference or translation machinery would have to be

built to interpret the internal state data *from each task*. Instead of this, I argue that groups of tasks that communicate should express any important information about the environment or their actions in their role representations so they can communicate through a common role structure.

When creating role structures for an agent's various tasks, the designer must take into account the dependent roles identified in section 3.4. These roles contain information that depends on information stored in other roles. Role information must be communicated to tasks with dependent roles as part of the agent's inter-task communication. This means that representation structures must contain information in a form that a task with dependent roles can interpret in order to correctly update its roles. Task communication and appropriate role structures are the means for establishing epistemological links between dependent representations.

3.7.3 Communication Summary

This section discussed what information is communicated between tasks. Since tasks operate on roles, they should be communicating information about the entities bound to those roles, specifically the represented roles (since unrepresented roles are not needed outside of their task). This leads to the proposal that representation should be the channel by which tasks communicate (many other agent architectures, e.g. [10][65][71], have also used representation for communication).

This section outlined a structure for role representation. Roles have six basic components that I will call "Index", "Property", "Identify", "Action", "Confidence" and "Progress". These components are summarized in table 3.2.

Table 3.2: Representation Component Summary

Component Name	Component Contents
Index	A task dependent label that indicates this representation's role in the current task, i.e. its role name.
Property	Information about the bound entity that is stored in the representation (and hence is a small amount)[a].
Identify	A perceptual description used to detect the entity associated with the role in the sensory stream. Identify may contain the perceptual data used to make the initial role/entity binding and information about how to maintain the binding after it has been made.
Action	The action that the agent wishes to take on (or with respect to) the entity bound to this role. It also contains the results of the action on the role (when the action completes) and additional parameters needed by the action (beyond those in the Property component).
Confidence	A measure of the agent's belief that the information stored in the role representation is correct, i.e. it is a measure of usefulness [14].
Progress	Denotes how well the current, ongoing action on the entity associated with this role is proceeding.

a. For all the representations used by the walk-the-dog agent, this component could be called the "where" component because it stores the position of the bound entity. However, I use the term Property because different tasks (especially at different levels of the decomposition hierarchy) may store different information in their representations.

At this point, it is worth emphasizing that Index and Identify are independent of each other. That is, a single entity can play several roles (so the Identify component would be the same, but the Index component would change) and a single role can be fulfilled by several entities (so the Index component would remain fixed while the Identify component changed). An example of the former would be if the pour-a-cup-of-coffee agent had to throw away the cup after it was used. During the pour-a-cup-of-coffee task, the cup would be associated with the role, **cup**. However, for the throw-trash-away task, the same cup might be associated with **trash**. An example of the latter occurs in the drive-to-the-store task when different signs get assigned to the **street-sign** role for the read-street-signs task.

This role structure addresses domain characteristics 1, 2, 3, 5 and 6. The roles are designed to carry only the information used by the current task and thus be maintainable in a dynamic environment (1). The roles are used for communication between architectural components (2 and 5) and finally they hold quantitative data (6) about the environment beyond the agent's sensor horizon (3).

3.8. Architecture

How should the agent's tasks be laid out in its architecture? The designer must now create an architecture that uses and manipulates roles discussed in the previous sections.

3.8.1 Example Task Architecture

Since this thesis targets multi-layered architectures (see section 1.3.2), the walk-the-dog tasks must be divided into layers. The tasks must be partitioned such that they can effectively use the representations designed thus far. This means that tasks the control effectors and bind their roles to PRs should be placed in the most responsive layer, i.e. the PA layer. This includes all the leaf tasks of figures 2b - 2f. These tasks should be in the PA layer because the PA layer's tight perception/action loops can keep representations bound to PRs up-to-date.

The other tasks may or may not be in the PA layer. The more tasks that the PA layer needs to handle, the more likely it is to be overburdened and thus reduce the agent's reaction time. These tasks are better off in a second layer of the architecture where they can activate the PA layer tasks as needed.

3.8.2 Architecture Rationale

The previous steps of the methodology have guided the designer in choosing what to rep-

resent and the trade-offs associated with representation. The purpose of this thesis is to guide the design of representation, not architecture. However, agents need architectures and so this section discusses the impact that representation may have on the structure of an agent's architecture. An issue that is fundamental to both representation and architecture design is that of "cycle time". "Cycle time" is the time for a task to perceive the environment (including representation maintenance), select an action and execute it. Executing the action does not necessarily mean running it to completion, but rather issuing effector commands that move toward the task's goal within the current environment. The next time through the perception/action loop, the task will choose new effector commands and these may continue the previous "action" or they may start a new one.

Representation should be designed so that it can be maintained at a rate that is effective for the particular task using the representation. The impact that this has on architecture is that a task must be executed by an architectural component that has a rapid enough cycle time to maintain that task's representation at rates required by the tasks. This thesis assumes that the designer will be using a layered architecture and these architectures map nicely to the "role hierarchies" that tend to arise when dependent roles represent portions of the environment at different abstractions. This means that if a role is dependent on one or more roles in one layer, that role's task should be in a higher layer. The abstract property stored in that role will typically be expensive to compute and so would slow down the cycle time of the lower layer. Tasks that require high rates of representation maintenance should be executed by the PA layer, but this must be tempered by the computational requirement of that maintenance. Tasks that can get by with slower maintenance should probably be executed by other layers of the architecture because the more complex the task that the PA lay-

er has to be able to execute, the more likely the layer will be to slow down and ruin the responsiveness of other tasks.

The division of tasks into layers, will also depend on a number of other engineering issues, such as latency/bandwidth of inter-task communication, capabilities of layers of the target architecture, amount of task inference and computational power required. Communication is perhaps the issue most effected by representation. Tasks that execute in parallel and share a role (i.e. communicate role information) should be in the same layer of the architecture because intra-layer communication is generally at a higher bandwidth and a lower latency. Tasks at different levels of the hierarchy communicate information about the roles at different levels. For example, suppose $role_1$ depends on $role_2$ and $role_3$, which are in tasks at different layers of the architecture. There will necessarily be communication as part of the maintenance of $role_1$ and this communication represents the "link" between these roles. The designer will want to consider the frequency of the communication when placing the tasks in the architecture this way. Natrajan [57] discusses methods of automatically updating $role_1$ when either $role_2$ or $role_3$ is updated and such a scheme involves even more communication. The designer needs to consider how often role information is exchanged under their particular maintenance scheme. Often, roles in tasks at different levels of the decomposition can be at different layers of the architecture because the lower layers of the architecture maintain the important details needed for effector control, while the higher layers maintain other data that needs to be updated less frequently.

3.8.3 Architecture Summary

In this step, the agent designer must try and structure the tasks in the decomposition hierarchy into "layers" of the agent architecture. The first step was to divide the agent's tasks

into PA layer and non-PA layer tasks. The PA layer tasks would be executed by a component of the agent architecture that is responsive enough to events in a dynamic environment to appropriately maintain those tasks representation. The non-PA layer tasks may themselves be divided into layers based on a number of characteristics. Often the final structure of the non-PA portion of the architecture will be determined by implementation concerns (e.g., latency/bandwidth of inter-task communication and computational power required).

This methodology is concerned with the design of representation systems, not the design of agent architecture. Since each agent needs an architecture, this step approaches the question of architecture design from the perspective of representation's effect on it. However, the final structure of the architecture is mostly an engineering issue, not a contribution of this work.

3.9. Post Methodology

After completing the methodology questions, the designer will still have work to do. At this point, the designer should have a layered system based on the agent's capabilities and the task decomposition. The designer should know what roles exist in the agent's tasks and what entities in the agent's environment can fulfill those roles. Knowing which role/entity bindings are shared and how long these bindings last will help the designer decide what roles need representation in the agent's architecture. Answering questions about the information that the agent must store and maintain in these representations will help structure the agent's perceptual system. Understanding what information must be shared or communicated between tasks will help determine the flow of information within the system. Finally, based on the flow of information and the nature of the tasks, the designer will partition them into sets which can be executed by different "layers" of the agent's architecture.

At this point, the designer must implement the agent architecture such that it can effectively manipulate the system of representation that has been designed. The agent must use its perception abilities to maintain the information stored in the role representations and the agent architecture must have a means of exchanging task role binding information between tasks (at the same or at different layers).

3.10. Methodology Summary

This section repeats the important aspects of the methodology. The methodology is meant to answer three questions about agent representation (sections that most directly address those issues are given in parenthesis, though answers to the questions in those sections may depend on the answers to questions in previous sections).

What should be represented (see section 3.4)? How should that representation be structured (see section 3.7)? How should that representation be maintained (see sections 3.5 and 3.6)?

The design can answer all these questions by answering the questions of the methodology, which are repeated below.

- *What are the agent's primitive skills?*
- *Which tasks can be decomposed into sequential subtasks? Which can be decomposed into parallel subtasks?*
- *What are the task roles?*
- *What entities can fulfill those roles?*
- *What role bindings are shared between tasks?*
- *What information about the entity bound to a task role is needed for the task?*
- *For what roles would it be useful to develop an explicit representation?*
- *How often should the task role information be verified?*
- *What information can the agent extract from the environment to recognize the entities that should be bound to the current task's roles?*

- *How does the duration of the various role/entity bindings effect the perception system?*
- *What level-of-detail (or resolution) is required in the information of the representation?*
- *What information is important in inter-task relationships?*
- *How should the agent's tasks be laid out in its architecture?*

The final role structure developed by the methodology is also repeated in table 3.3 (see section 3.7 for the culmination of the arguments for it).

Table 3.3: Representation Component Summary Reprise

Component Name	Component Contents
Index	A task dependent label that indicates this representation's role in the current task, i.e. its role name.
Property	Information about the bound entity that is stored in the representation (and hence is a small amount)[a].
Identify	A perceptual description used to detect the entity associated with the role in the sensory stream. Identify may contain the perceptual data used to make the initial role/entity binding and information about how to maintain the binding after it has been made.
Action	The action that the agent wishes to take on (or with respect to) the entity bound to this role. It also contains the results of the action on the role (when the action completes) and additional parameters needed by the action (beyond those in the Property component).
Confidence	A measure of the agent's belief that the information stored in the role representation is correct, i.e. it is a measure of usefulness [14].
Progress	Denotes how well the current, ongoing action on the entity associated with this role is proceeding.

a. For all the representations used by the walk-the-dog agent, this component could be called the "where" component because it stores the position of the bound entity. However, I use the term Property because different tasks (especially at different levels of the decomposition hierarchy) may store different information in their representations.

This role structure addresses the domain characteristics laid out in section 1.3.1 because the roles are:

- designed to carry only the information used by the current task and thus be maintainable in a dynamic environment (section 1.3.1.1)

- specialized to the needs of particular tasks (section 1.3.1.4)

- used for communication within and between architectural components (sections 1.3.1.2 and 1.3.1.5)

- designed to hold quantitative data (section 1.3.1.6) about the environment beyond the agent's sensor horizon (section 1.3.1.3).

This chapter presented a design methodology for autonomous agent with representation systems that specialized to the tasks and capabilities of the agent. The methodology is a series of questions to guide the designer in creating representation systems that are efficient and effective for use in dynamic domains.

Chapter 4 *Applying the Methodology (Bruce)*

The first agent designed by the design methodology laid out in chapter 3 was Bruce [82][83]. Bruce's task is to play hide-and-seek in our laboratory against a human controlled opponent. Figure 6a shows Bruce and figure 6b shows the vehicles that he can play against (though only one at a time). The human hides the vehicle somewhere within the game area (see figure 7). Bruce will search the game area for the opponent and attempt to touch, i.e. tag, it upon sight. Once Bruce has spotted the opponent, the human may drive it away from Bruce. If Bruce looses his opponent during the chase, he should examine nearby objects to see if the opponent is hiding behind them. If Bruce does tag his opponent, he wins and the game is over.

(a) (b)
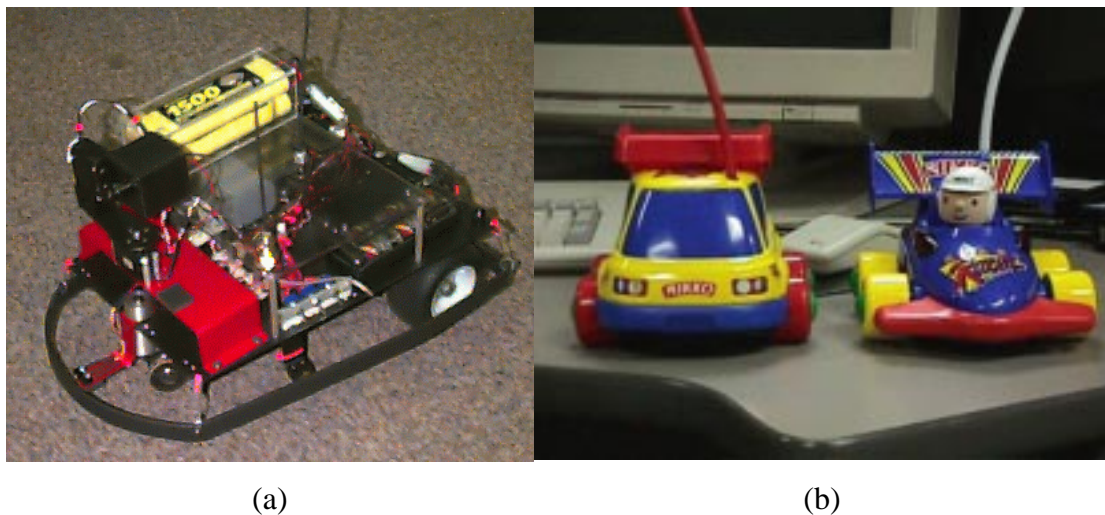
Figure 6. Bruce and Opponents

Bruce was built at UVA and is based on the RugWarrior board [38]. He possess two in-

dependent drive wheels with fairly coarse shaft encoders, a single color camera on a pan/tilt platform and a "bump skirt" capable of detecting impact on the front, left and right sides. Onboard processing is done by a pair of Motorola MC68HC11s. Bruce has a radio modem to communicate with his host workstation, a Sun Sparc 10, where most of the processing is done. Bruce also has a video transmitter that broadcasts the images from his camera to a Datacube MV200 for image processing.

In this section I examine how Bruce's software architecture was developed using the methodology presented in this thesis. Figure 7 shows a plan to search the game environment that Bruce must execute. The numbers indicate steps in the plan with the solid line representing the path to follow. The dashed lines indicate where and in which direction Bruce should turn his camera to look for his opponent. Bruce and his field of view are shown near the completion of step 3. The other shaded shapes denote objects in the game zone. By following the methodology's steps and answering its questions, a design to follow this plan and play hide-and-seek emerges.
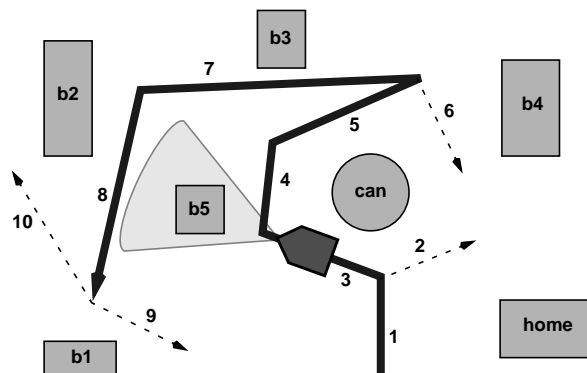
Figure 7. The Search Plan

4.1. Task Decomposition

What are the agent's primitive skills? The designer must now see what capabilities can be combined into "black box" skills that can be used in the bottom-up portion of the de-

composition process. Bruce's basic skills are navigation to a location and control of the camera's pan/tilt mount. The agent navigates to a location by combining motor and perceptual capabilities. By servo-ing toward the position of location-specific perceptual characteristics detected in its camera, Bruce can move throughout his environment. By varying these characteristics, the agent can move to a variety of static objects or chase the opponent.

Which tasks can be decomposed into sequential subtasks? Which can be decomposed into parallel subtasks? As mentioned previously, playing hide-and-seek consists of searching for the opponent, chasing the opponent (to tag it) and possibly performing another search behind nearby objects if Bruce looses the opponent during a chase. These three tasks are shown as subtasks of the play-hide-and-seek task in figure 8 and the control flow between them, i.e. the order in which they execute, is shown in figure 9a. Note that in figure 4, arrows denote flow control with dashed arrows indicating control flow between parent and child tasks.

In order to search, the agent needs to go between landmarks looking for the opponent and avoiding obstacles. In order to view the entire game area, Bruce will need to point his camera at certain areas of the environment, but he need not navigate to them. So, the search task
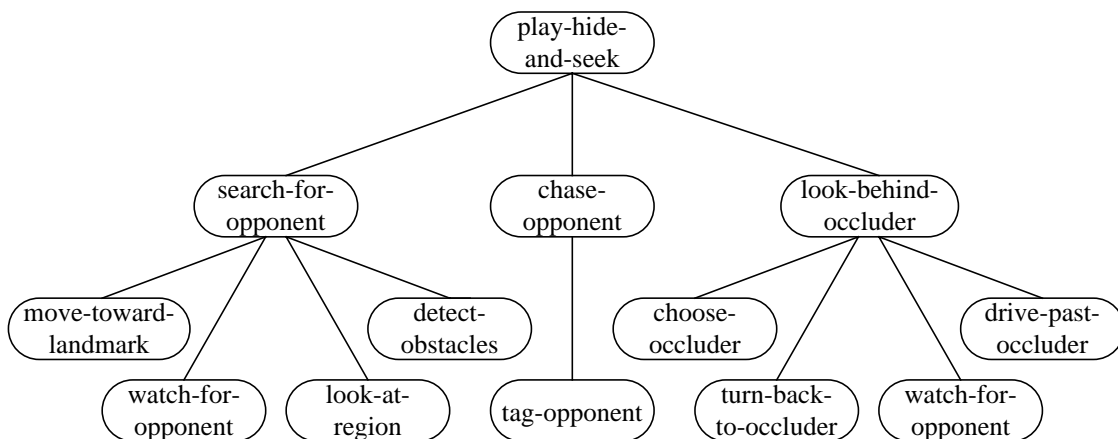
Figure 8. Hide-and-Seek Task Decomposition
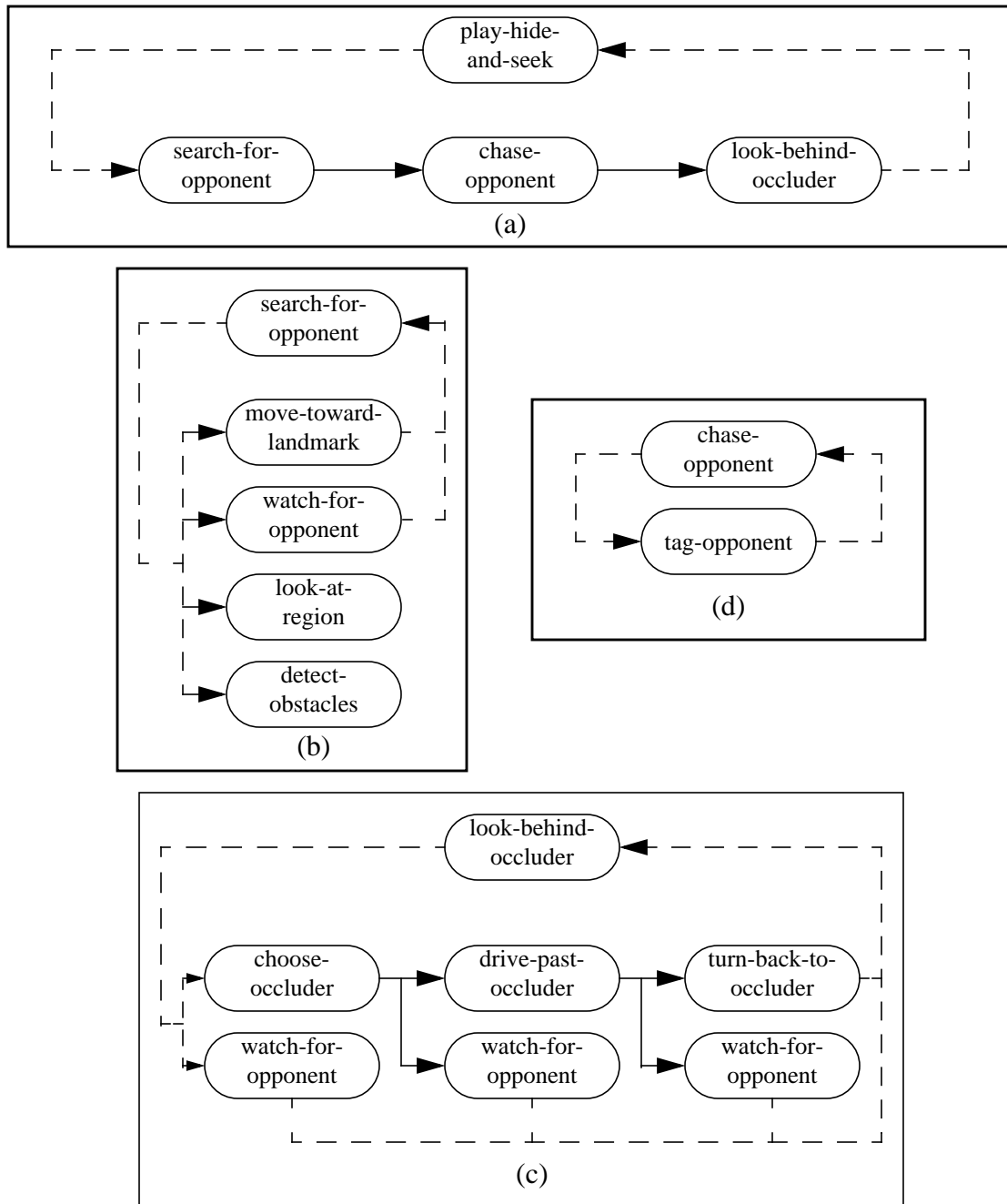
Figure 9. Hide-and-Seek Task Flow Diagrams

consists of a navigation task (move-toward-landmark), an obstacle detection task (detect-obstacles), a camera control task (look-at-region) and a task that monitors for the opponent (watch-for-opponent), all operating in parallel. These are shown as subtasks of the search-for-opponent task in figure 8. In figure 9b, these tasks are shown as stacked ovals, meaning

they operate in parallel. The control flow arrows from the parent task show that they all begin executing at the same time, but only move-toward-landmark and watch-for-opponent can transfer control back to the parent (and thus end the execution of all four tasks). This occurs when Bruce either reaches the landmark he is moving toward, or detects the opponent.

When the opponent is detected, Bruce should begin chasing it. In this situation, control flows from the search-for-opponent task to the chase-opponent task (figure 9a) and from the chase-opponent task to the tag-opponent task (figure 9d). The tag-opponent task simply visual servos Bruce toward the opponent until Bruce's "bump skirt" detects an impact (and the opponent is close enough that Bruce believes he has impacted the opponent and not something else).

If the opponent escapes while being chased, Bruce should look for it behind any object in the vicinity of where Bruce lost visual contact. This process is handled by the look-behind-occluder task and its sequential subtasks, shown in the right of figure 8. The control flow of figure 9c shows that several sets of parallel subtasks are used to check if the opponent is hiding nearby. The watch-for-opponent task runs in parallel with every other task so that Bruce switches to the chase-opponent task if the opponent is detected at any point. When looking behind objects, Bruce must first select an object, near the opponent's last known position, that is possibly occluding his view of the opponent, i.e. the opponent is hiding behind it. This is handled by the choose-occluder task. Then Bruce must drive "behind" it (relative to his starting location). Due to the mechanics of Bruce's drive system, going behind the object involves moving until the rear (drive) wheels are past the object and then turning back to face it. This sequence is performed by the drive-past-occluder and

turn-back-to-occluder tasks. Note that in figure 9c, the watch-for-opponent subtasks only transfer control back to the look-behind-occluder tasks while the other subtasks transfer control to the next subtask in the sequence. This is because the other tasks are sequential steps in "looking behind" an object, while watch-for-occluder indicates that a different branch of the decomposition (chase-opponent) should take over.

4.2. Identify Task Roles

What are the task roles? The search-for-opponent task has three roles, **opponent**, **landmark** and **region**, to represent the aspects of the environment important to searching the game area. Its subtasks must deal with the complexity of navigation. So the move-toward-landmark task has **landmark** and **obstacle** roles since where it directs the agent is influenced by the position of the goal landmark and any obstacles. The detect-obstacles task has an **obstacle** role. Watch-for-opponent has an **opponent** role to associate with the vehicle Bruce must tag, and look-at-region has a **region** role corresponding to the area of the game environment at which Bruce must look.

The chase-opponent and tag-opponent tasks each have the **opponent** role since these tasks are concerned only with driving toward the opponent. The various subtasks that make up look-behind-occluder (choose-occluder, drive-past-occluder and turn-back-to-occluder) all have the **occluder** role. The drive-past-occluder task also has a role called **intermediate-target**, which is not mentioned in the task name. This role is associated with an object that Bruce can use to estimate his position relative to the occluding object (see figure 11b). Figure 10 shows the same task decomposition diagram as figure 8 annotated with the task roles. The roles are shown in bold when they occur in the task name and in callout boxes otherwise.
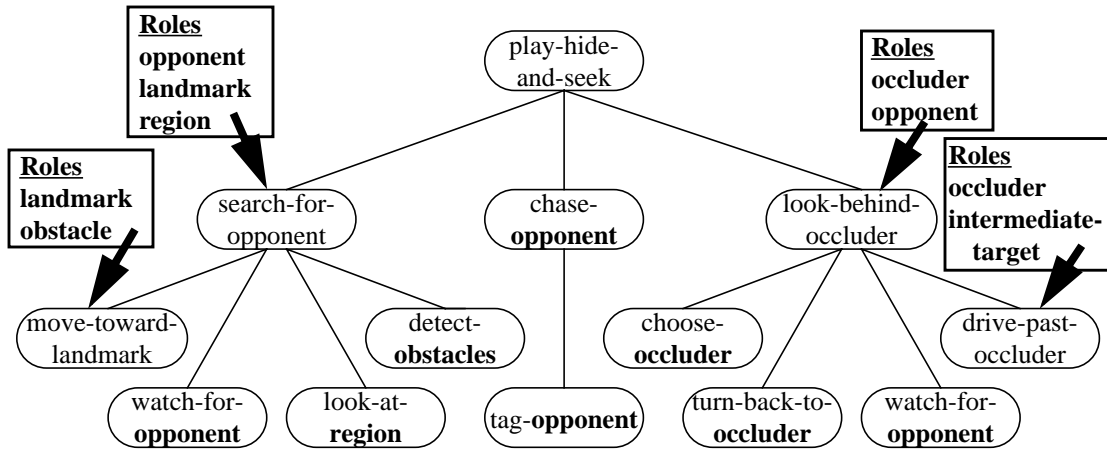
Figure 10. Hide-and-Seek Task Roles

What entities can fulfill those roles? The four roles in the search portion of the hide-and-seek game are **landmark**, **opponent**, **region** and **obstacle**. The **landmark** role will be played, at various times, by the various objects along the agent's search route (see figure 7). The **opponent** role can only be played by Bruce's opponent, which is either the dump truck or the racecar from figure 6b, but not both in the same game. The same is true for the **opponent** role in chase-opponent and its subtask. The **region** role can be filled by any portion of the game area at which Bruce needs to look. In fact, this role can be better described as a neck angle than as specific features of the environment. Finally, the **obstacle** role is filled by any object that is along the direct path between Bruce and his current navigation goal.

In the look-behind-occluder portion of the game, the **occluder** and **intermediate-target** roles can be filled by any object that is in the correct position. That is, the occluder can be any object that has approximately the same azimuth and is closer than the last known position of the opponent while the **intermediate-target** can be associated with any object that is off to the side of the occluder and further away than the occluder plus one Bruce-length. This allows the same motor control system used in the move-toward-landmark task to be

re-used to servo Bruce toward the intermediate-target. However, this time, he must stop when he has passed the occluding object and not when he reaches the intermediate-target.

4.3. Representation of Task Roles

What role bindings are shared between tasks? For Bruce's task, this is simple. The **landmark**, **region** and **opponent** roles are shared between search-for-opponent, and its child tasks move-toward-landmark, look-at-region and watch-for-opponent respectively. The **obstacle** role of the detect-obstacles task is shared with the move-toward-landmark task since that task calculates how to move the agent's wheels based on the **landmark** and **obstacle** roles. The **opponent** role is also shared between the watch-for-opponent, chase-opponent, tag-opponent) and look-behind-occluder tasks. From figure 9 we can see that both the watch-for-opponent tasks in the decomposition hierarchy transition to their parent task which transitions to the chase-opponent task. The **opponent** binding is shared so that Bruce chases the detected vehicle. Finally, the **occluder** role is shared between the subtasks of the look-behind-occluder task (choose-occluder, drive-past-occluder and turn-back-to-occluder).

What information about the entity bound to a task role is needed for the task? In this domain, the agent must navigate through the environment searching for the opponent. The tasks of move-toward-landmark, detect-obstacles, look-at-region, tag-opponent, drive-past-occluder and turn-back-to-occluder all need the positions of the entities associated with their roles to control the agent's effectors. Watch-for-opponent and choose-occluder both determine initial positions for the objects associated with their roles and pass this data on to other tasks. So, the important information about the entities associated with task roles in this domain is position. I chose to store positions in ego-centric, polar coordinates be-

cause the agent's capabilities make position estimates relative to its current location much more accurate than position estimates relative to some external frame. In other words, any position computed from information gathered by sensors mounted on the agent can be turned into an agent-centric position by a simple transform (such as shifting by the current camera pan angle or translating from the sensor's position to the center of the agent's body). Although such positions can be converted to other coordinate frames, maintaining positions in those frames requires either special sensors or knowledge of the positions of multiple other landmarks to triangulate [21]. Since Bruce will not often be able to see many landmarks (and his encoders are error prone), he is much more successful with an ego-centric system.

For what roles would it be useful to develop an explicit representation? The **landmark**, **opponent** and **occluder** roles can all benefit from explicit representation because Bruce may have to take action with respect to them when they are outside his field of view. The move-toward-landmark task may begin with the next landmark outside the current field of view and a representation for that role will allow Bruce to begin moving toward it. The tag-opponent task benefits from role representation because the opponent may drive outside of Bruce's field of view during the chase and Bruce may be able to reacquire the opponent by turning in the direction where it was last seen. The object bound to the **occluder** role will necessarily go outside Bruce's view during the look-behind-occluder task, specifically during drive-behind-occluder. The **occluder** role needs representation so that Bruce can execute the turn-back-to-occluder task because the occluding object will not be visible after completing the drive-past-occluder task (see figure 11b).

The **intermediate-target** role would seem to not need representation because, as shown
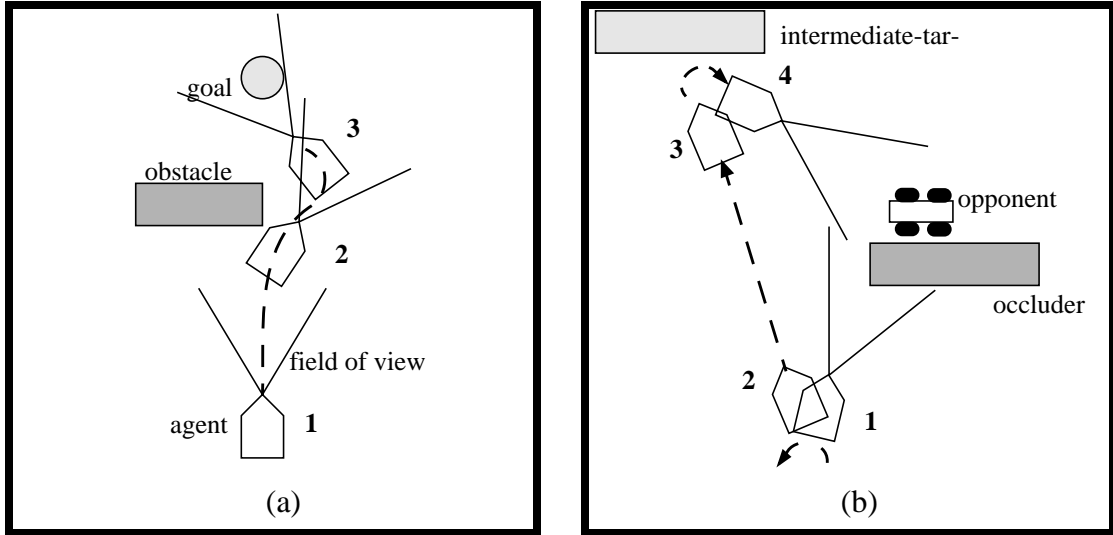
Figure 11. (a) Obstacle Avoidance (b) Look-Behind-Occluder

in figure 11b, it is played by an object that is always within sight of the agent. However, obstacle avoidance en-route could cause the intermediate-target to fall outside Bruce's field-of-view and so a representation for this role could be useful.

The **region** role is interesting because there is no object in the environment that the agent should bind to this role, rather it is described by a head angle to pan the camera in search of the opponent. As such, the head angle could just be a parameter for the "look-at" action and the role needs no representation. However, I have opted to represent this role in order to have a uniform communication structure between parent tasks and their children (see Section 4.7 for a discussion on this communication interface).

The representation of the **obstacle** role is an interesting issue that causes iteration between the representation and perception steps of this methodology. Bruce's vision system often mis-identifies obstacles (false positives). If the **obstacle** role has a representation, it may take longer for the agent to realize that the binding is incorrect than it would take in a stateless system. This is the classic argument against representation put forward by Brooks

[15]. Since a stateless system cannot store information, it must re-evaluate the environment whenever it wishes to act. This means it can forget previous erroneous percepts in the minimum possible time. A system with state or representation may continue to make decisions based on previously determined, incorrect information. This argues that obstacles should not have a representation since they are often misidentified and this may cause Bruce to avoid empty space. However, due to the limited ability of the camera to pan, obstacles go outside Bruce's field of view, as shown in figure 11a, step 2. Without an obstacle representation, Bruce would invariably hit obstacles along his flank when his camera is past them, but his drive wheels are not. This argues for a representation of local-space [14]. The designer can choose to have a representation for the **obstacle** role and try to improve the perception or choose no role representation and try to improve the navigation control algorithm to avoid obstacles outside the field of view. For Bruce, I use a representation compromise. Obstacles that are detected at more than a certain distance away are not bound to the **obstacle** role, although Bruce's navigation system will take their position into account on that cycle of his PA loop. This allows Bruce to be only somewhat fooled by far away, false obstacles. When the obstacles are too close to Bruce, they are bound to the **obstacle** role so that they are represented should they pass outside the camera's field of view. Clearly, there should be multiple **obstacle** roles, since Bruce can be avoiding one obstacle that he can't see when another obstacle blocks his new path. However, Bruce's simple visual avoidance algorithm cannot avoid two obstacles and try to move toward the goal[1].

---

1. Part of the thinking in designing Bruce's obstacle avoidance routine was that a stateless system was almost capable of handling the avoidance. Representation was only needed to remember obstacles that had passed outside the field of view. Since this thesis is about the design of representation systems, significant effort was not spent in designing a routine to avoid multiple obstacles. Once the **obstacle** role is bound, Bruce will not avoid other obstacles until this role is unbound.

How often should the task role information be verified? Since the positions of the entities associated with the roles are used to control Bruce's effectors (wheels and camera platform), they should be verified at the same rate as the effector control loop. The role maintenance scheme here is simple. The agent needs to know about the position of either a landmark or occluder (depending on the current task), plus any obstacle in order to navigate. He also must attempt to detect the opponent at all times. Therefore, for each pass through the active task's effector control loop, the agent must verify the position of the entities associated with the current task's roles and attempt to bind the **opponent** role.

4.4. Perception

What information can be extracted from the environment to recognize the entities that should be bound to the current task's roles? Bruce's perception system consists of a single color camera whose signal is transmitted to an off-board receiver and processed by a combination of a Datacube MV200 image processor and a Sun Sparc 10. This hardware supports two primitive visual operations at speeds high enough to support Bruce's task, edge detection and color histogramming. Bruce's visual identification routines are straightforward given the availability of these operations. First, Bruce finds the ground/non-ground boundary, or groundline [34] as shown in figure 12. Large vertical discontinuities in this line may represent the boundaries of objects (note the Sprite can). Object identification is accomplished by comparing the color histogram of the pixels between pairs of discontinuities with stored histograms for various objects [72]. So, by comparing the truck histogram with the histogram of the pixels within discontinuity pair 1 and within discontinuity pair 2, Bruce can determine that pair 2 delineates the truck, while pair 1 does not.

I designed Bruce with the necessary histograms to recognize the landmarks that appear

discontinuity pair 1   discontinuity pair 2
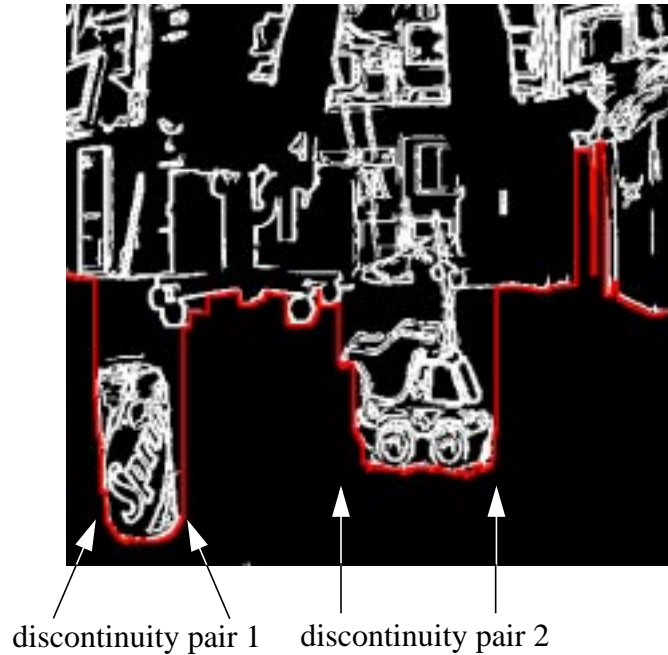
Figure 12. Groundline Image with Opponent

along his search path, as well as the opponent he must chase. Obstacles can be detected as any objects (i.e. pair of discontinuities) that are closer than Bruce's current goal, and on a direct path to the goal (see Appendix A, section A.1.1 for more on how the navigation system works).

Bruce must be able to determine the positions of the entities associated with the task roles. For this, the "ground-plane constraint" [34], which states that the ground is a flat horizontal plane and all objects in the world rest on the ground, was used. This constraint allows Bruce to calculate the distance to an object based on the smallest distance between the bottom of the image and any of the points along the groundline between the object's discontinuity boundaries. That is, using known camera geometry, we can relate distance from the bottom of the image to depth. Bruce computes azimuth from an object's position within the image and the camera's known angular field of view and pan angle. These polar coordinates (azimuth and depth) are ego-centric, meaning with the agent at the coordinate sys-

tem's origin. Since Bruce's tasks consist of moving himself toward objects in the world (either to search or to chase) this works fine for the leaf tasks. However, the main search task must convert from the landmark coordinates stored in Bruce's game area map to ego-centric coordinates for navigation and vice versa. I discuss this in Appendix A.

How does the duration of various role/entity bindings effect the perception system? First, the binding duration of the roles in Bruce's tasks must be determined. In particular, the designer is interested in roles that have "long" binding durations. In other words, roles that stay associated with the same object over long periods of time, i.e. are bound to different objects infrequently relative to their task's execution time. There is no single definition of how much time is "a long time", it is relative to the agent's perceptual capabilities (how often the binding could be changed) and the execution time of the agent's tasks.

In general, there are two ways for a role to have a "long" binding duration. The first is for a single task to have a role remain bound to the same entity for the duration of its execution (which is "a long time"). Bruce does this with the **landmark**, **region** and **intermediate-target** roles of the move-toward-landmark, look-at-region and drive-past-occluder tasks respectively. Each of these roles should remain bound to a single object for the duration of the task that uses them. The second means of having a long binding duration is to have many sequential tasks share a role that they expect to be bound to the same object. Both the **opponent** and **occluder** roles are shared by multiple, sequential tasks and all those tasks expect the role to be bound to the same entity. The only role in this domain with a "short" binding duration, i.e. one that is not long, is the **obstacle** role. On each cycle of Bruce's perception/action loop, the detect-obstacles task will bind this role to any object that is an obstacle to Bruce's current destination, even if that object is the same object that

was an obstacle last time. The catch is that there could be a long binding duration when an obstacle goes outside the field of view of the front mounted camera, but can still be hit by Bruce's body with its rear mounted wheels. If the obstacle is close enough to the edge of the field of view, the role must remain bound to it (so that Bruce remembers its location until he is "past" it).

Now we can use the binding duration information to structure the perception system. Bruce's visual processing can be divided into two classes, location and tracking. Location is the process by which a role is initially bound to an entity in the environment. Tracking is the processes by which the information about the entity bound to a task role is maintained. Most roles remain bound to the same entity for the duration of their task and (once bound) the **opponent** and **occluder** roles remain bound across multiple tasks. This means that Bruce can save a significant amount of visual processing by foveating the regions of space occupied by entities bound to task roles. In this case, foveating means limiting visual processing to a particular region of the image.

In principle the location and tracking processes could use completely different mechanisms for identifying objects in the image(s). Presumably, the location process would be slower but more accurate than the tracking process, while the tracking process could use the context (previous position(s)) for assistance. Bruce, however, uses the exact same visual operations (edge detection and color histogramming) for both location and tracking. Tracking is faster because any discontinuity boundary pairs that are too far from the last known position of any object are not matched to that object's stored histogram. This means fewer objects need to be analyzed and the computation can proceed faster.

One difficulty with this two process scheme is what to do if the associated object moves

beyond the tracking fovea. In other words, what happens if the associated object moves faster than expected (or the agent moves slower) or goes out of the field of view? Bruce needs a means of identifying when the tracking process is failing or has failed, so that he can activate the location process again. Simple thresholding on the "amount of match" between the stored histogram and the histograms of any objects detected in the groundline suffices for this purpose. The histogram matching process is a modified version of Swain and Ballard [72] and Terzopoulos and Rabie [73]. It produces a percentage match value equal to the number of pixels of matching color between the two histograms divided by the total number of histogrammed pixels (see section A.1.2 in Appendix A for more details). When this score becomes too low, Bruce can disable the tracking process and either begin the location process again, or start the look-behind-occluder task.

What level-of-detail (or resolution) is required in the information of the representation?
This question must be answered for each of the primary subtasks of play-hide-and-seek. For the search-for-opponent task, the information being used is the position of various landmarks, obstacles, regions and, at some point, the opponent. Since this is being used for effector control, we want it to be as accurate as possible. This means that Bruce should be estimating (dead-reckoning) such positions as little as possible, implying that he should be pointing his visual resource at these objects as often as possible. During obstacle avoidance, it is often impossible to keep the obstacle, the destination and the agent's body's current heading in view at the same time. Bruce must be looking at the objects effecting his navigation as often as possible, understanding that some trade-offs must be made based on how long it has been since an important object was last seen (see the discussion on confidence in Section 4.5).

The chase-opponent task is similar in that the opponent's position is used for effector control and so must be as accurate as possible. This means Bruce should keep his camera trained on the opponent as much as possible.

The look-behind-occluder task has different needs for the **occluder** and **intermediate-target** roles. The intermediate-target is an object selected for its spatial relationship to the occluding object. Bruce's physical layout (front camera, rear wheels) means he is more successful navigating to a point that is fairly straight in front of him than to a point behind an object he must maneuver around. This is because, at some point when moving around an object, that object can typically not be seen by the camera and so the estimated position of the object becomes inaccurate. So, Bruce selects an intermediate-target such that, if he navigates toward it for a certain distance, he will be in the correct position relative to the occluder to execute the turn-back-to-occluder subtask. Bruce can afford to reduce the accuracy of his estimate of the occluder's position because his present concern is the intermediate-destination, an object that has been specifically chosen for the purpose of getting Bruce to a certain position relative to the occluder. He only needs to accurate enough knowledge of the occluder's position that he can turn around and view the area behind it (remember he's not really interested in finding the occluder, but the opponent).

4.5. Communication

What information is important in inter-task relationships? The play-hide-and-seek task is divided into three groups of communicating subtasks. By examining each group, we can discover the role information communicated between tasks. The search-for-opponent task moves Bruce throughout the game environment attempting to find the opponent. This task has **opponent**, **landmark** and **region** roles about which it communicates with the watch-

for-opponent, move-toward-landmark and look-at-region subtasks respectively. The search-for-opponent task gives the move-toward-landmark task a perceptual description of a landmark to which it can bind its **landmark** role and then maneuver the agent toward the associated object. It also gives the watch-for-opponent subtask a description of the opponent so that the **opponent** role in that task can be bound when it is detected (and trigger the switch to the chase-opponent task). The look-at-region subtask receives a description (the position) of the region to look at in hopes that the watch-for-opponent task will bind the **opponent** role while the agent is looking there.

The search-for-opponent task must communicate this perceptual data to its subtasks because there are various landmarks and regions that Bruce needs to be concerned with throughout the task. In other words, the move-toward-landmark task should not include hard-coded perceptual information because the agent's route plan will dictate what landmarks the agent should move toward. By having the search-for-opponent task pass this role information to its subtasks, the subtasks can be reused to execute each leg of the search route (instead of having a different leaf task for each landmark or region). In fact, since the agent can play the hide-and-seek game against different opponents (see figure 6b), the watch-for-opponent task can be parameterized by opponent perceptual description. The parent task will know what opponent to search for and communicate this to the subtask.

Another important kind of data that is communicated between search-for-opponent and its subtasks is action input and output. Input means parameters that the task needs to perform its action(s) and output means the results of the actions(s). Move-toward-landmark moves the agent to within a certain tolerance of the landmark. This tolerance is determined by search-for-opponent to insure Bruce stops at the correct position (to perform a look-at-

region for example). Move-toward-landmark reports when it has reached the specified landmark and supplies the landmark's current ego-centric position. This allows search-for-opponent to determine the agent's location in its map of the environment and decide on the ego-centric position of the next landmark to be visited. Note that the landmark's current position is already computed as the "information" that the task needs to know about the entity associated with the role (as per section 4.3) and so the action result can be just a boolean value (action completed successfully or action failed) if the position is communicated as well. Both the watch-for-opponent and look-at-region tasks receive no configuration information about how to perform their actions on their task role, and each communicates success or failure to its parent task (watch-for-opponent signals "success" or "failure" by binding or not binding the **opponent** role).

Note that detect-obstacles does not communicate with the parent task. This is because it is a part of the search task that is abstracted away from search-for-opponent task. However, it does communicate with the move-toward-landmark task by binding the **obstacle** role. This role and the **landmark** determine the effector commands sent to the wheels by move-toward-landmark.

Next, I examine the inter-task communication in the chase-opponent task. The chase-opponent task consists of a single subtask, tag-opponent. These tasks communicate the perceptual description of the **opponent** and whether or not the opponent has been "tagged", i.e. the result of the "chase" action. In addition, the tag-opponent task must communicate some information to decide if control should transfer to the look-behind-occluder task. I call this information "confidence" and it represents the agent's belief that the stored position of the opponent is correct. As long as the binding between the **opponent** and the op-

ponent is being adequately maintained, i.e. the tracking function is finding the opponent in the camera images, the confidence stored in the **opponent** representation will remain high. However, if the opponent slips outside the field of view or behind an occluding object, the **opponent** confidence will decrease over time, until the chase-opponent task transfers control to look-behind-occluder.

Finally, the look-behind-occluder task and its subtasks communicate similar perception and action information. The choose-occluder subtask's job is to select an object that is potentially occluding the opponent, based on the opponent's last known position. When the **occluder** role is bound, i.e. the choose-occluder task successfully completes, the perceptual description of the bound object is passed up to the parent look-behind-occluder task so that is can be distributed to that task's other children. The drive-past-occluder and turn-back-to-occluder inform their parent task when they have completed their respective missions (getting within tolerance of the object associated with the **intermediate-target** and facing the occluder). The fact that these tasks communicate this information to their parent task instead of to the next task in the sequence is really a decision that results from implementation control flow (see section 4.6 for an explanation).

4.6. Architecture

How should the agent's tasks be laid out in its architecture? The first issue the designer must tackle is determining which tasks have rapid perception/action cycles, bind their roles to PRs (see section 3.6) and require no inference. Such tasks can be executed by the agent's PA layer. Typically, these processes control some sensor(s) and/or effector(s). Table 4.1 shows the tasks and sensor processing or effector they control.

Most of the leaf tasks in the hide-and-seek decomposition control effectors, as shown by the number of wheel and camera mount controlling tasks in table 4.1. Watch-for-opponent and detect-obstacles control sensors, but also need to be PA layer tasks because their representations are used by other PA layer tasks. **Opponent** and **obstacle** are both PRs and the tasks that share these roles expect them to be maintained at effector control rates (so watch-for-opponent and detect-obstacles need to be in the PA layer).

Table 4.1: Task Controlling Capabilities

Task Name	Sensor/Effector Controlled
move-toward-landmark	wheels
drive-past-occluder	wheels
turn-back-to-occluder	wheels
tag-opponent	wheels
look-at-region	camera mount
choose-occluder	camera mount
watch-for-opponent	opponent detection
detect-obstacles	obstacle detection

The other tasks in the decomposition tree belong in other layers of the architecture. The search-for-opponent task uses a map of the game environment to direct its (now PA layer) subtasks. The map is in a global coordinate system that is different from the egocentric system used to store positions in the PA layer tasks. Therefore, search-for-opponent uses a different representation. This representation does not need to be updated and so I place the search-for-opponent task in another layer of the architecture that I call the task executor layer, or TE. I also place the look-behind-occluder task in this layer so that it can sequence its subtasks to drive the agent behind an occluder. This is not the same control flow shown in figure 9. Control flows to the parent between each subtask. I choose to do this so the TE is in charge of activating and deactivating processes in the PA layer (via markers). This reduces the amount of control flow with which the PA layer need be concerned. Similarly, the chase-opponent task exists mainly to allow the switch between

phases of the hide-and-seek task (searching, chasing, etc.) to take place in the TE. As shown in figure 9, search-for-opponent transitions to chase-opponent, and chase-opponent activates the tag-opponent PA layer task. Watch-for-opponent could directly transfer control to tag-opponent, but I prefer control to flow through the TE.

Bruce does not need a planner to play hide-and-seek in this domain because he is given a search plan to execute[2]. Thus, the play-hide-and-seek task (the root of the decomposition tree) is not a separate task in the implementation. However, a more complex version of the game can be imagined in which play-hide-and-seek has an actual implementation and it is in a third layer of the architecture. Suppose that Bruce is chasing the opponent and he can no longer locate it. At this point, Bruce can continue his current search pattern, but if he had a route planner, he could compute a new search route based on his current location. Play-hide-and-seek could be a route planning task that passed its search plan to search-for-opponent. When a new plan was needed, control would transfer back to play-hide-and-seek and a new plan would be generated. This task, then, should be placed in a new layer of architecture because it requires more inference than the TE layer tasks. In fact, this task requires a full inference (planning) engine, whereas the TE layer tasks merely set the context for the PA layer to execute pre-defined plans. The TE tasks need no inference capabilities and so to preserve their cycle time, the route planning task should be in a different layer of the architecture.

4.7. Bruce Implementation

This section describes the result of the design process carried out in the previous sections. This section details how the answers to the methodology's questions were combined to

2. I generated a search route for Bruce to follow for the game layout shown in figure 7.

form a working autonomous agent. The remainder of this section is organized as follows. First, I describe the agent's PA layer representation and the operation of the PA layer's main loop (including marker maintenance). I then describe the actual PA layer behaviors and how they achieve the agent's goals. Lastly, I discuss how representation is used for inter-layer task communication.

4.7.1 Bruce's Representation

First I will describe the structures that are used to represent task roles in the PA layer. These representations are called *markers* after the work of Agre and Chapman [2]. Based on the information communicated between tasks (section 4.5), each marker has components for all the communicated information outlined in section 3.7.3, except "progress". It also has a "dependency list". Table 4.2 summarizes what is contained in the various marker components and the dependency list. These summaries will be explained in greater detail as I discuss marker maintenance and the general operation of the PA layer's main loop.

Table 4.2: Components of Bruce's Markers

Component Name	Component Contents
Index	The role name used in combination with Action (see below) to select the current action from the PA layer's action table.
Property	Ego-centric, polar (r, θ) coordinates of associated object
Identify	Divided into Locate and Track subcomponents. Each has a color histogram of the object that this role can be bound to an "amount of match" threshold to declare a match successful. Although the histograms are the same, the match thresholds are typically different and the region searched for an appropriate object is different (see below).

Table 4.2: Components of Bruce's Markers

Component Name	Component Contents
Action	Used along with Index to determine an action to take from an action table. Also contains an action result component to store data for parent task about the outcome of a particular action. Note that this component can be null, indicating no action to take on this marker. This happens when a marker represents an object that is part of an action indicated by the Action component of another marker (see Dependency List).
Confidence	Contains a "found" flag that is set whenever the associated object is detected in the current image and a counter for how many images have passed without finding the associated object. Also contains the "instantiated" flag indicating whether or not this marker is currently associated with an object in the world.
Dependency List	A list of other markers in the PA layer that are needed to complete the action specified in the Action component. This is how tasks with multiple roles are executed. A marker's Action component contains the action that completes the task and that task's roles are represented by the marker and the markers in its dependency list. For example, obstacle markers are placed in the Dependency List of the landmark marker for the move-toward-landmark task.

Bruce's PA layer operates in a tight perception/action loop. However, since the PA layer uses representation, it must have representation maintenance as part of that loop. In order to understand the maintenance algorithm and the PA layer's loop, I must first introduce the concept of instantiated vs. uninstantiated markers. Simply, an instantiated marker is one for which the PA layer has selected an entity in the environment to associate with the marker, based on the marker's Identify component. An uninstantiated marker then, is a marker that has not yet been associated with an entity. An uninstantiated marker often reflects a higher layer's expectations about the world and the PA layer can confirm those expectations by instantiating (binding) the marker to some object. For example, when Bruce has completed step 3 of his search plan (see figure 7), object b3 is not within his field of view. Since b3 is the landmark for the next invocation of the move-toward-landmark task (step 4), the TE

layer search-for-opponent task will create an uninstantiated **landmark** marker with the appropriate Identify component for b3. Although the move-toward-landmark task cannot immediately detect object b3, the **landmark** marker contains b3's ego-centric position. Bruce can begin to move toward the expected position of b3 and when b3 comes into view, he can start running the marker's Locate routine to try and detect the object. When the Locate routine finds an appropriate object, it is associated with the marker and the marker is said to be instantiated.

4.7.2 The PA Layer Main Loop

The purpose of the PA layer's perception/action loop is to select appropriate actions for the agent. Other layers influence the PA layer's decision, but ultimately, it is the PA layer that controls the agent's effectors. For the PA layer, actions will be closely coupled to current and recent perceptions. Since Bruce's PA layer has representation, the action selected is a function of both the current perceptions and the information stored in the markers.

The main loop proceeds as follows. First, it executes its maintenance algorithm on the instantiated markers and then it tries to instantiate any markers that remain uninstantiated. Finally, the PA layer examines the Action component of each marker and executes the specified action. I will explain each of the steps in detail.

4.7.2.1 PA Loop Step 1: Update Markers

This step verifies that the positions stored in the Property[3] component of the markers are consistent with the positions of their associated objects in the world. Positions are updated by the following hypothesize-and-test algorithm. First, a new groundline is computed and

---

3. Recall that in section 3.7.3, I decided to call this component Property instead of Position (which would be appropriate here) because different agents may store different data in their representations.

"object regions" are segmented from it as in section 4.4. Next, new positions for all objects associated with markers are hypothesized by transforming the positions stored in the markers based on the wheel encoder values read since the last update. For each instantiated marker, if its position falls within the agent's current field of view, the marker's Track routine is used to analyze the objects detected in the groundline. The results of the Track routine and the distance between each object and the marker's stored position are used to create an ordered list of the detected objects by likelihood that they are associated with the marker. After all instantiated markers have run their Track routines, each object creates an ordered list of markers by likelihood that it is "the" object associated with the marker, i.e. by differences between hypothesized marker position and detected object position. These ordered lists of markers to objects and objects to markers are passed to a stable marriage algorithm [40] that determines marker-object correspondences. If an object is selected as a marker's correspondent, then the position of the object is stored in marker's Property field and its found flag (as mention in the Confidence component of table 4.2) is set. If no object is selected as the marker's correspondent, then the hypothesized position becomes the new stored position.

4.7.2.2 PA Loop Step 2: Instantiate Markers

After the instantiated markers have been updated, the uninstantiated markers can be matched to objects not associated with instantiated markers. Using the transformed positions from step 1, the agent proceeds through a similar matching process, using the results of the markers' Locate functions instead of Track functions. This system is biased toward markers that are already instantiated because an object will only be allowed to correspond to one marker. In other words, because the instantiated markers get matched to the objects

first, an instantiated marker could be matched to an object that is "supposed to" match an uninstantiated marker. This bias is allowed because an object associated with an instantiated marker has presumably been tracked over time, while an object associated with an uninstantiated marker is based on expectations of the higher layers. So, if the foveal image region searched by the Track function contains a matching object, it is more likely that that object corresponds to the instantiated marker than an uninstantiated marker. This is because perceptual context, i.e. previously executions of the PA loop, has lead the Track function to consider that image region, but the uninstantiated marker has yet to be grounded in perception. Bruce values perception over memory or expectations.

4.7.2.3 PA Loop Step 3: Select/Execute Action

After updating the markers, the PA layer looks through its list of markers to see if any have Action components with actions. It searches the list, in order, and executes the first non-null Action component it finds. This Action component specifies some action to be done on (or with respect to) the object associated with the marker. The PA layer has access to an action database indexed by the Action component and the Index component. Each of the "actions" in this database consists of one or more tasks executed in parallel (though the implementation is psuedo-parallel).

4.7.3 Task Implementation

The leaf tasks in the decomposition of figure 8 are implemented as behaviors or skills, called PA processes. These PA processes use the information in the marker containing the action and possibly information in other markers in that marker's dependency list. Several tasks can be started by the Action component of a single marker. Table 4.3 shows the map-

ping between leaf tasks and marker Action/Index components.

Table 4.3: Mapping of Tasks to Activating Marker Components

Task Name	Action and Index Component of Activating Marker
move-toward-landmark	GoTo on **landmark** marker
detect-obstacles	GoTo on marker with any Index
look-at-region	LookAt on **region** marker
watch-for-opponent	(uninstantiated **opponent** marker)
tag-opponent	Chase on **opponent** marker
choose-occluder	(uninstantiated **occluder** marker)
turn-back-to-occluder	AlignWith on **occluder** marker
drive-past-occluder	GoTo on **intermediate-destination** marker

This section discusses how the representation activates PA processes and how they use the representation. The GoTo action on a **landmark** marker begins the move-toward-landmark and detect-obstacles processes. These processes share the **landmark** marker and any **obstacle** marker that is created in order to move Bruce toward a specific landmark. Actually, Bruce's navigation system is more complex than just these two processes and is described fully in Appendix A, but those details are unimportant to the discussion here.

The look-at-region, chase-opponent and turn-back-to-occluder are each implemented by single PA process that are activated by the various marker Actions indicated in table 4.3. The implementation of each of these processes is unimportant as their effects have been amply described in previous sections.

Drive-past-occluder is activated by a GoTo action on an **intermediate-destination** marker. This marker comes from the TE (see next section) and its position and GoTo Action parameter are based on the occluder's position. The parameter to the GoTo action (re-

call that parameters are also stored in a marker's Action component) indicate how close Bruce needs to get to his destination. This parameter is based on the position in the occluder marker and should be close enough to the intermediate-destination to make sure Bruce can turn around and see the area behind the occluder. The GoTo action can be used again because the PA processes that implement GoTo can navigate Bruce to the position stored in any marker. In general, there need not be separate PA processes for each task because, for example, move-toward-landmark and drive-past-occluder can be implemented by the same ones.

The watch-for-opponent and choose-occluder tasks do not need implementing PA processes because they're handled by step 2 of the PA layer's main loop. The **occluder** marker has a special Locate routine that allows it to be bound to any object near the position stored in the **opponent** and the **opponent** Locate routine has a histogram for the truck or car opponent. The parent tasks in the TE monitor for these markers to be instantiated.

4.7.4 Inter-layer Communication

This section describes how the markers are used to communicate with the tasks in the TE layer. When a PA layer action completes, the PA processes associated with that action are deactivated and a reference to the marker whose Action component initiated the processes is passed to the TE. I say "a reference" merely to indicate that the marker is not deleted from the PA layer, and the TE is alerted that it should look at the marker's contents. The TE's response to information it reads in markers is usually to create new markers and pass them to the PA layer. The Action components of these markers will cause the PA layer to begin new actions. Now the TE layer may choose to delete the old marker from the PA layer, based on the completed task. For example, a **landmark** marker whose GoTo action has

been completed will be deleted, but an **opponent** marker that has been instantiated will not.

When a GoTo or LookAt action completes (meaning a landmark has been reached or a region has been looked at), the marker's Action result component is set to COMPLETE and the marker is passed to the search-for-opponent task at the TE layer. Based on the **landmark** or **region's** position relative to the agent and the current step in the search plan, this TE process generates the next **landmark** or **region** marker and passes it to the PA layer.

If the **opponent** marker is instantiated, and the agent is executing the tag-opponent PA process, the opponent marker's confidence is assessed based on the count of the number of frames since the correspondent was found in the image. When this number grows too large, the tag-opponent PA process exits, placing the TARGET-LOST result code in the Action result slot of the marker and passes the marker to the TE chase-opponent task. This task then transfers control to the look-behind-occluder task.

The look-behind-occluder task generates an **occluder** marker that is passed to the PA. When this marker is instantiated, the TE creates an **intermediate-destination** marker with a GoTo action. This begins the drive-behind-occluder task. When that task completes, the TE places an AlignWith action on the **occluder** marker. The **occluder** marker is not deleted from the PA layer after instantiation and so it maintained during drive-behind-occluder. Turn-back-to-occluder uses the occluder position to turn back and face the object. If the opponent marker is instantiated at any point while executing this task, the same transfer of control to the chase-opponent task as before.

Everyone takes the limits of his own
vision for the limits of the world.
- Arthur Schopenhauer

Chapter 5   *Applying the Methodology (Spot)*

Spot [77][79][80] was developed as core technology for several projects at the Texas Robotics and Automation Center (TRACLabs) in Houston, TX and is shown in figure 13. While he represents a building block for a number of applications, I will motivate this work by describing Spot in the context of one of those applications.

Figure 13. Spot

Spot's task is to be a robotic waiter. The imagined scenario is that Spot is at a cocktail party, serving hors d'oeuvres to attendees. As a waiter, the robot should just get within a certain distance of the person to be served and that person will take an hors d'oeuvre from

the robot if they wish. Since there will be multiple people at the party, Spot should keep circulating, moving between guests and offering them food. Every 30 seconds, Spot should check on a new patron.

Spot is a Real World Interface B-14 mobile base with a Directed Perception pan/tilt camera platform mounted on top. Three cameras are mounted on the platform, two grey-scale cameras and a central color camera. The B-14 has an onboard Pentium 120 processor that controls the agent's drive system (wheels and encoders) and the pan/tilt platform, as well as performing the color image processing. An offboard Pentium 200 MMX machine provides stereo imaging using the grey-scale cameras. In addition, this robot has the 3T agent architecture [10] available and was targeted toward that architecture.

5.1. Task Decomposition

What are the agent's primitive skills? In order to determine the agent's primitive skills, we need to examine its capabilities more closely. First, the B-14 base and its supplied control software allow the programmer to set forward and rotation speeds for the wheel base. Encoder values for forward and rotational movement are also accessible. Second, the pan/tilt camera mount can have either of its axes controlled by positional control. Since movements can be interrupted before they complete, velocity control can be achieved by setting an axis' velocity and sending that axis to its position limit in a given direction. The stereo vision system was also available at the beginning of the design process and can track objects in 3 dimensions using virtual fovea known as "proximity spaces" [36][77]. This system can receive requests to start tracking objects at given positions within the current camera view and outputs the current positions of all tracked objects within the camera's view. These positions are relative to the center of the baseline between the two cameras and

not to the body of the robot. The proximity space system can also output if an object is within a space's area and this can be used to determine if a space has lost the target it was tracking. A built-in "search" behavior allows "empty" proximity spaces to try and reacquire their targets by randomly moving in a sphere around their target's last known location. Finally, Spot has a color vision system that can detect skin toned hues in the image.

Spot provides several examples of how capabilities can be used to "build" primitive skills. First, the agent can combine the wheel base's motor and encoder control to navigate to a location. Second, the pan/tilt mount (referred to as the "head") can either smoothly track targets or rapidly redirect the agent's attention (i.e. its cameras). Finally, the agent can monitor the location of entities using the proximity spaces and detect patrons with the skin tone system.

Which tasks can be decomposed into sequential subtasks? Which can be decomposed into parallel subtasks? As part of Spot's task, he must identify potential patrons (humans) and maneuver up to them. He should follow and track them for some time. Then he should move to other patrons and try to serve them. After some time, he should return to patrons he's served before and see if they want anything more. Due to the fact that Spot will not be able to keep all patrons he's ever served in view at one time, he will have to remember the positions of previously served patrons, so he can return to them. A task decomposition diagram is shown in figure 14 and a task flow diagram is shown in figure 15.
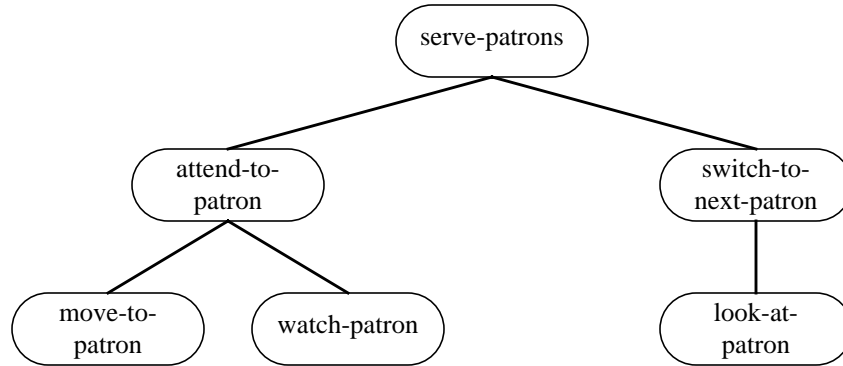
Figure 14. Serve-Patrons Task Decomposition
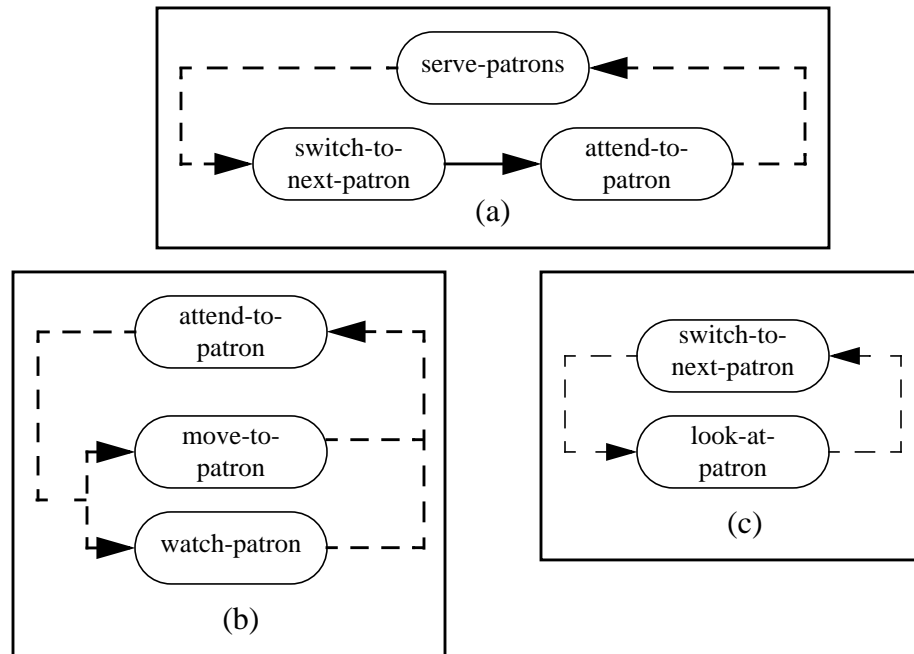
Figure 15. Serve-Patrons Task Flow

Serve-patrons knows about all the people who have been or should be served and directs switch-to-next-patron to change the agent's attention to one of them. This occurs when the look-at-patron moves the camera's pan/tilt mount to view the requested patron and identifies the patron in the image. Then, attend-to-patron moves the agent to within a certain distance of the patron while tracking them. Move-to-patron accomplishes the locomotion, while watch-patron handles the tracking.

5.2. Identify Task Roles

What are the task roles? Each task in Spot's domain has a **patron** role for the person that is being attended to or should be attended. The serve-patrons task has multiple **patron** roles and directs its subtasks to switch the agent's attention between the associated patrons. Figure 16 shows the decomposition diagram of figure 14 with the task roles in bold.
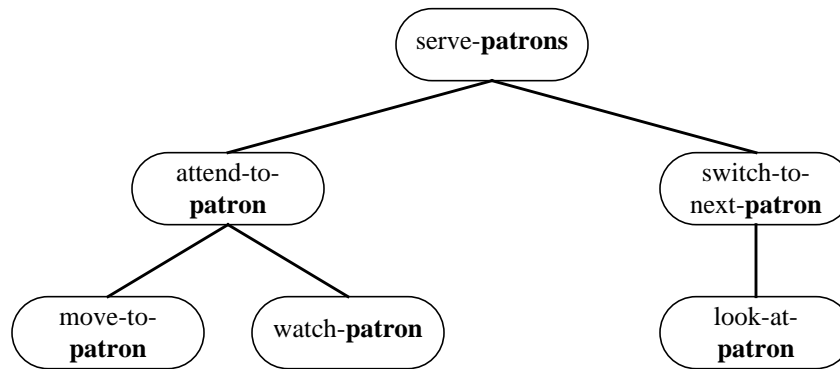
Figure 16. Serve-Patrons Task Roles

What entities can fulfill those roles? The patrons in this domain are all humans. The serve-patrons task has an initial idea of where the people to be served are located and so the **patron** roles in the other tasks can be filled by the humans at those locations.

5.3. Representation of Task Roles

What role bindings are shared between tasks? All tasks have **patron** roles and the tasks are all designed to operate on the same human (once switch-to-next-patron begins using a **patron** role, all other tasks except serve-patrons effect the same patron). Therefore, the **patron** role is shared between all tasks. The serve-patrons task has multiple **patron** roles (each bound to different patrons) and can switch which one is being used by all the other tasks in the decomposition.

What information about the entity bound to a task role is needed for the task? In order to move between patrons, the agent needs to know their positions in the environment. So, like Bruce, position is the piece of information that needs to be computed for the entity associated with each task role. Spot's stores these coordinates as a 3-tuple of distance from the agent (depth), azimuth and elevation angle.

For what roles would it be useful to develop an explicit representation? The **patron** roles of the serve-patrons task need representation because the associated humans cannot all be in view at the same time. Likewise, the **patron** role in the switch-to-next-patron and look-at-patron tasks needs representation because the patron that the agent wants to switch its attention to may be outside the current field-of-view.

The **patron** role of attend-to-patron and its subtasks is an interesting case because the agent's capabilities effect the answer. As mentioned in section 5.1, the agent has a proximity space vision system to track the entities associated with roles. Each proximity space is a fovea covering a certain region of 3-D space. Remembering the fovea's position is essential because perceptual computations are only performed on data within that region. This information seems to require representation for **patron** roles so that there are data structures where foveal positions can be stored. However, as stated in section 5.1, the proximity space system tracks objects and so whatever state it needs to do this is stored internally. This vision system is an agent capability given to the designer and does not require role representation just to store perceptual data.

However, other tasks in the decomposition share the **patron** role with attend-to-patron (and its subtasks) and these other tasks have a representation for that role. Recall that the proximity space system outputs positions of tracked objects within the camera image and

so cannot remember positions outside the field of view. Even though the associated patron might always be in view during the attend-to-patron task, this role should be represented because that representation can be shared with serve-patrons and store the position that allows serve-patrons to have switch-to-next-patron get the patron back into view to restart attend-to-patron.

How often should the task role information be verified? As with Bruce, the roles in attend-to-patron and switch-to-next-patron (and their subtasks) are used for effector control and so must be verified at the rate of the effector control loop. However, the **patron** roles in the serve-patrons task cannot all be verified at effector control rates, since all the associated humans are probably not in view. This is not necessary though since only one of the patrons will be attended to at a time. Spot only needs to verify the positions of patrons often enough that they do not move too far from where he last saw them. In general, how often Spot should check on a previous patron depends on many factors, such as total number of patrons to be served and the time that persons can be expected to stand in a group and not circulate. For Spot's task, a 30 second time window between attending to a patron is part of the task specification.

5.4. Perception

What information can be extracted from the environment to recognize the entities that should be bound to the current task's roles? Spot has two different visual systems at his disposal. First, there is the proximity space system [36][79][80]. This system rapidly tracks the contents of small, spherical volumes of the agent's environment. These volumes contain texture patterns that can be correlated between stereo and temporal image pairs. Recall that the proximity space system is a "black box" that allows Spot to estimate the position

of the objects associated with task roles by "placing" proximity spaces around them and reading the positions of those proximity spaces as they track their objects through time.

The interface to the proximity space system, as described in section 5.1, places some constraints on Spot's design. First, there is the need to initialize proximity spaces. The system can be told to start tracking an object at a certain location, but how does the agent determine that location? Second the agent architecture must activate and deactivate proximity spaces as the field of view moves to contain or not contain the object being tracked. Also, the proximity space system reports locations in head-centric coordinates, so the agent must convert these coordinates to the ego-centric coordinates it uses. Given the known field of view of the cameras, the agent can deactivate any space whose ego-centric position in not within the current field of view. The agent architecture also controls the head and so when tracking an object, the agent must move the head to point the cameras at the location of a given proximity space.

The unanswered question is, how do proximity spaces get initially "placed" on objects? The fact that using one skill (the proximity space system) requires another skill (to initialize the proximity spaces) was not anticipated, but effects the representation. Spot initializes proximity spaces using the color vision system. Recall that Spot's single color camera is used to find humans, the objects associated with the **patron** roles, by examining the image for the red hues associated with skin tones. Areas of the image that respond to the "skin tone" filter are matched to a constraint model of human head and arm positions. A successful match means a human has been detected and a proximity space should be initialized at that location. However, since there is only a single color camera, Spot cannot accurately derive the 3-dimensional position of the human. Therefore, Spot must search for the correct

"depth" to initialize the proximity space. Conceptually, the color vision system produces a vector from the color camera through the center of the head of the detected human (the head is used since facial "texture" works well for the proximity spaces[1]). The human could be at any point along this vector. The proximity space system "slides" a proximity space along this vector until it "lands" on the feature detected by the color vision system. "Slide" means keeping two of the three proximity space coordinates fixed and slowly increasing the depth coordinate (the distance between the agent and the proximity space). "Lands" refers to the proximity space detecting a pattern it can track and beginning the tracking process[2]. The proximity space begins tracking when it finds enough texture that it can match in the left and right images that it believes its volume is occupied by some object. The proximity space system will communicate the fact that a particular space has begun tracking to the agent architecture. A more comprehensive description of the initialization process can be found in [77][79]. Note that humans are PRs because only the agent's given capabilities are needed to maintain information about them (the proximity space system can track them fast enough for effector control and the color vision system can initialize the proximity spaces).

How does the duration of various role/entity bindings effect the perception system? The duration of the **patron** role/entity bindings in this domain is the length of the execution of Spot's application. That is, once a human is bound to a **patron** role, the agent should maintain that binding for the length of the party. As with Bruce, the long binding duration allows

---

1. Strictly speaking, the proximity space system is not a black box since I use knowledge of its inner workings (like initializing the proximity space on the human's head because I know heads are very "trackable" using the system's technique). However, the proximity space system is a powerful capability given to the designer and still provides benefits to the agent's architecture.
2. An additional constraint on the design of the agent head control is that when initializing the position of a space, the agent must not move the head because the proximity space system will be moving a new sphere through space trying to determine an object's depth from the agent. If the head (and thus the cameras move), the proximity space will not end up on the object that triggered the color vision system.

Spot to make use of the proximity space system's tracking capability. Spot can foveate each entity with a proximity space, and so the active proximity spaces form the foci of attention for the agent.

However, the long binding duration does present problems with accurately maintaining the collection of **patron** roles. Since all the patrons will not be in view at the same time, there is a possibility that some patron will move, from the last place the agent saw them, while the agent is attending to someone else. This is a general problem with a limited visual resource. Spot is designed based on the domain knowledge that people at parties mostly stand in groups and don't move much. Provided that the interval between times when the agent attempts to view a patron is sufficiently small, the patron should not move much from their last known location. If they have moved somewhat, the proximity space system's built-in search behavior may be able to find them.

What level-of-detail (or resolution) is required in the information of the representation?
The attend-to-patron task requires the position (or estimated position) of the human patron to operate. Since this task uses the information to control effectors in keeping track of dynamic, moving targets, it requires the most accurate information available, i.e. the data from the proximity space system, not the encoders. The switch-to-next-patron task could also benefit from an accurate patron position estimate to redirect the agent's cameras so the proximity space system can reacquire the patron. However, since the patron is often not visible for some time, the position of the **patron** role in switch-to-next-patron is allowed to be less accurate, i.e. encoder based.

5.5. Communication

What information is important in inter-task relationships? In order to answer this question, let's begin with a review of the task flow depicted in figure 15a. The serve-patrons task has a collection of **patron** representations. It passes one of them to switch-to-next-patron, which redirects the agent's attention to the human associated with that role. Control passes to attend-to-patron, whose subtasks cause the agent to perform the normal tracking and following that Spot uses for attention. At some point, serve-patrons will stop the attend-to-patron task from following the current patron and select a new patron. Serve-patrons passes the new **patron** role to switch-to-next-patron and the cycle repeats.

Clearly, the information moved between all tasks includes the position of patrons. Since the tasks all share **patron** roles, they exchange the position of patrons.

In this domain, all patrons are humans and they are all identified (skin tone detection) and tracked (proximity spaces) in the same manner. This means that no perceptual information needs to be communicated between tasks, since position is the only (perceivable) dynamically changing data that distinguishes entities in this domain. While the proximity space system is given to the designer, and so all information needed to track is held within it and not in the tasks, the tasks could exchange information about how to initialize the proximity space locations. We can imagine a more complex domain in which particular patrons need to be attended to and we can distinguish them by clothing. A perceptual description of their clothing could then be passed from serve-patrons to switch-to-next-patron to allow it to initialize the proximity space location for attend-to-patron. We can also imagine attend-to-patron periodically checking the proximity space for these perceptual cues to verify that the space is still tracking the correct entity.

Another piece of information that must be communicated between tasks is the confidence the agent has in the position stored in its role representations. The confidence can be based on either the time since the agent has had the associated proximity space in view (and hence been tracking it) or whether or not the proximity space system says the given proximity space has lost its target (implying the space should be in view). If the agent's confidence in a role is low because the agent has not seen the associated patron for too long, switch-to-next-patron should be directed to bring the patron into view). If a role's proximity space has lost its associated entity, then watch-patron and move-to-patron stop controlling effectors based on the space's location. The proximity space system itself tries to reacquire its target, so attend-to-patron's subtasks just wait.

When switch-to-next-patron has located a human with the color vision system and is in the process of searching for it with the proximity space, serve-patrons should not ask switch-to-next-patron to try and acquire a new target. Depending on the distance from the human to the agent, the proximity space initialization could take several seconds and so the agent's confidence in the role is high at this time. This fact is communicated to serve-patrons to keep the attentive focus on this human until the initialization is complete.

5.6. Architecture

How should the agent's tasks be laid out in its architecture? The first step is to identify the PA layer tasks. The leaf tasks in figure 14 are PA layer tasks because they control the agent's effectors and bind their **patron** roles to PRs. Attend-to-patron and switch-to-next-patron are also PA layer tasks because they merely activate their subtasks and pass them the role that they receive from serve-patrons. The domain, as described here, specifies no means of selecting among patrons except based on the agent's confidence in their positions.

This means that serve-patrons can also be a PA layer task because it chooses the **patron** role to send to the other tasks without any significant inference, i.e. just based on the time since the patron was last seen. Since all Spot's tasks can be placed in the PA layer, there is no need for a multi-layered architecture. However, since the 3T architecture [10] is the given target architecture for this robot, I make use of its RAP layer (see section 5.7).

5.7. Spot Implementation

This section describes the PA layer representation that results from the decisions made in the previous sections. I also discuss the operation of the agent architecture to show how representation is an integral part. Since Spot uses the 3T architecture [10], I adopt its conventions of referring to the PA layer as the "skill layer" and the tasks that execute there as "skills".

5.7.1 Spot's Representation

The agent's tasks are all in the skill layer and they all use the various **patron** roles. Each patron role is associated with a marker data structure, similar to those used in Bruce. This data structure contains the same Index, Property, Identify, Action and Confidence components. The contents of the marker components are summarized in table 5.1.

Table 5.1: Components of Spot's Markers

Component Name	Component Contents
Index	A task dependent identifier for the associated object's role in the current task. For Spot's relatively simple domain, this is always set to **patron**.
Property	The 3-dimensional, ego-centric position of the associated patron.

Table 5.1: Components of Spot's Markers

Component Name	Component Contents
Identify	1) Locate routine that handles skin tone model matching 2) ID for marker's proximity space in proximity space system (Spot reads data from the proximity space system over a network in the form of <ID, coordinate> pairs. Coordinates of a proximity space are placed in the Property component of the correct marker based on this ID)
Action	Requests for activation of skills: possible values • GoToAndTrack - activates move-to-patron and watch-patron • LookAt - activates switch-to-next patron
Confidence	1) Instantiation state: possible values • instantiated - marker is associated with an object and a proximity space is (presumably) tracking that object • uninstantiated - marker is not associated with an object and the agent is running the marker's Locate routine • partially instantiated - object has been detected by the marker's Locate routine and a proximity space is searching along the generated vector for the object 2) Timer that counts time since proximity space was last in the field of view (when it reaches 30 seconds, switch-to-patron will try to reacquire this marker) 3) Bit holding whether or not the proximity space system is tracking its target successfully (recall this is part of the proximity space system's output)

5.7.2 Skill Layer Main Loop

The skill layer main loop begins by updating the markers. The positions of all markers are transformed, based on encoder values read since the last time through the loop. Any markers with positions inside the field of view have the position of their proximity space stored in their Property component. Proximity spaces whose markers indicate that the associated entity should be in view are activated while those whose entities should have gone out of view are deactivated. The Confidence timers for markers with positions outside the field of view start (or continue) counting down and the timers for markers with positions

inside the field of view are reset to the highest value.

Next, any uninstantiated marker whose associated entity should be in view will have its Locate routine run. If a candidate location is detected, the marker's proximity space will be moved along the vector generated by the Locate routine. The instantiation state of such a marker is set to partially instantiated to keep the NeckControl skill from redirecting the cameras. When the proximity space lands on its target (i.e. no longer reports its occupancy as below the "lost" threshold), the marker is (fully) instantiated. If the proximity space reaches the agent's maximum vergable depth, failure is reported to the agent architecture and the marker remains uninstantiated.

5.7.3 PA Layer Skills

The data stored in the markers drives the skill layer of the agent architecture. The original skill layer of 3T responds to skill activation and deactivation requests from the RAP layer. I have modified the skill layer main loop to handle the role representation used by Spot, and so now markers are maintained before the normal operation of the skill layer begins, i.e. running active skills.

Spot has two active skills called GoTo and NeckControl. The operation of these two skills is driven by the Property and Confidence values of the markers. GoTo controls the agent's wheels to keep Spot within a fixed distance of the human he is attending, i.e. the one whose marker has a GoToAndTrack action on it. On each run of the GoTo skill, it assesses which of the humans associated with **patron** markers Spot should attend, based on the markers' Confidence components. In this way, GoTo performs the function of the serve-patrons task. If the marker whose human should be attended is not the marker currently being tracked by the GoTo skill, the skill the new **patron** marker's Action compo-

nent to LookAt. However, if the current marker is in the partially instantiated state, the GoTo skill will not set another marker's Action component to LookAt until either the marker is (fully) instantiated or the proximity space has been moved to maximum depth and the proximity space system reports that no trackable texture was found there.

NeckControl handles both the look-at-patron and watch-patron tasks. If a marker has a LookAt action on it, NeckControl will redirect the agent's view to that marker's position. Otherwise, NeckControl will track the location of the proximity space associated with the marker that has the GoToAndTrack action on it. Once a marker's position has been brought into view by a LookAt action, NeckControl sets the marker's Action component to GoToAndTrack.

The final question is how skill layer execution begins. The RAP layer [28] of 3T is used to activate the GoTo and NeckControl skills and place **patron** markers into the skill layer. The demonstration application for Spot begins with the RAP system communicating with the skill layer by passing it two **patron** markers with estimates of the positions of two people. One of these markers has a LookAt action on it, allowing the skill layer to execute the agent's task from there.

5.7.4 Summary

Spot's architecture operates by manipulating markers. Two skills accomplish the agent's task by communicating through this shared representation. The most important characteristic of Spot's representation is that, by virtue of its use as a communication medium, it is able to integrate multiple sensory systems into an existing agent architecture.

Chapter 6   *Applying the Methodology (Marcus)*

Marcus [78] is an agent designed to build structures from blocks scattered throughout our lab. He must navigate around, collect blocks, move them to a specific site and assemble them. Marcus is given plans to build structures and in this chapter, he is designed in the context of building an arch.
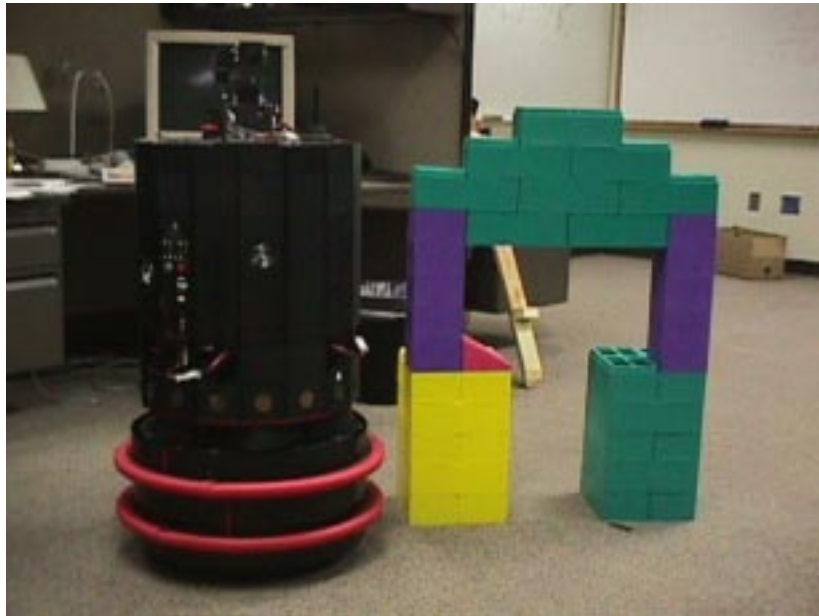
Figure 17. Marcus and the Arch in our Lab

Marcus is an augmented Nomadic Technologies model 150 robot (see figure 17). He possesses a 3-wheel drive base with encoders, an independently (from the base) rotatable turret, and a ring of 16 sonar sensors. He also has a color camera on, a pan/tilt mechanism, mounted on his top deck. All computation is performed onboard by a Pentium Pro 200 system running Linux. Marcus has routines to perform two basic image processing operations,
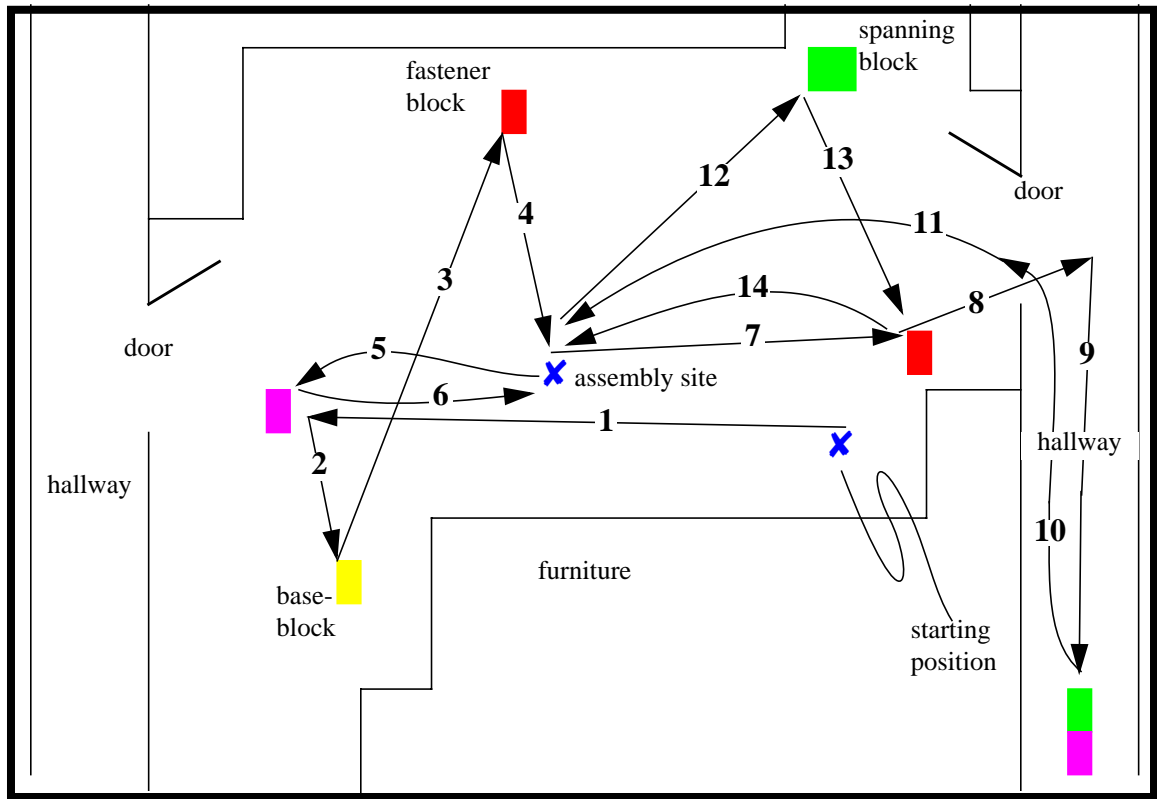
Figure 18. Marcus' Environment - A Portion of Olsson Hall 227

edge detection and blob coloring of image regions with specific RGB values. The results of the image processing are displayed to the user on a laptop (carried on the top deck) with a network connection to the internal computer. Marcus' environment is the computer vision laboratory in Olsson Hall 227 at the University of Virginia and the nearby hallway. Various colored blocks, are placed throughout this environment and a map of their positions is made available to Marcus. An assembly site is also specified on the map. The map is shown in figure 18. The arrows represent approximate paths that Marcus must travel in building the arch and the numbers represent the order in which these paths are followed. Marcus' starting position and the assembly site are represented by ✘s and the various blocks are represented by colored rectangles.

6.1. Task Decomposition

What are the agent's primitive skills? Marcus executes a plan to build an archway using a variety of primitive skills. Marcus can control his camera mount to pan/tilt to any position in approximately the hemisphere in front of his camera. Marcus' independently rotatable wheel base and turret decouple that hemisphere from being in the direction of motion, i.e. the mount itself can be rotated, changing the positions that the pan/tilt axis can move the camera to see. Marcus possesses skills to navigate to a location (implicitly orienting the base and possibly the turret), rotate the base and/or turret to face a location, and realign the camera's mount point with the wheel's "forward" direction. He can also use a speech synthesizer to give directions to his human assistant. Marcus does not have any grasping effector and so, in this discussion, any operations that require such an effector (e.g. picking things up, putting things down) are performed by the author. In addition, an area of his top deck is designated as the "toolbelt". He can both carry and move his camera to view objects placed there. Finally, he can monitor his sonars by reading in current range data.

Although not typically thought of as a skill, the agent also possesses an assembly planner[1]. This planner generates assembly instructions for structures that the agent is to build. The remainder of the design process is involved in the creation of an agent that can execute those plans.

Which tasks can be decomposed into sequential subtasks? Which can be decomposed into parallel subtasks? In order to analyze Marcus' task, we must examine the plan that Marcus executes to build the arch of figure 17. The task decomposition diagrams of figures 19 and 20 show the "top-level" steps in the arch building plan. Each of these steps can be further

---

1. This thesis is not concerned with planning or the creation of planners. The planning capabilities of Marcus are limited in that the "planner" outputs a single plan, to build and arch.

decomposed into collections of subtasks, as shown in figures 21 - 23. The notation under each top-level plan step task indicates the figure that contains its subtask decomposition (more details on this in section 6.2). Notice that a number of plan step tasks use the same subtask decomposition because the tasks are trying to accomplish (very nearly) the same goals. Task flow diagrams for all tasks can be found in Appendix B, figures 29 - 32.

The remainder of this section describes the task decompositions and task flows. Since the arch building plan (figures 19 and 20) is given at the start of the design process, its decomposition and flow are pre-defined. Table 6.1 provides a high-level summary of the tasks Marcus must perform to build the arch. The "At Location" column refers to Marcus being at the end of the indicated travel path number shown in figure 18. Since figure 18 does not show all the tasks Marcus needs to do at a location, there are multiple table rows for each location.

Table 6.1: Marcus Top-Level Decomposition Summary

At Location	Task Performed
start	go to the purple block (path #1)
1	search for base block, which can be either yellow or green
	go to base block (path #2)
2	pick up base block
	search for fastener block
	go to fastener block (path #3)
3	put fastener block in toolbelt
	search for assembly site (path #4 - doesn't typically have to move to get there)
4	put down base block
	go to top block (path #5)

Table 6.1: Marcus Top-Level Decomposition Summary

At Location	Task Performed
5	pick up top block
	go to assembly site (path #6)
6	stack top block on base block
	go to fastener block near door (path #7)
7	turn to view door
	go through door (path #8)
8	search for block stack in hallway
	go to block stack (path #9)
9	pick up stack
	go through door back into lab (path #10)
10	search for stack previously built at assembly site
	go to that stack (path #11)
11	determine the orientation (pose) of the stack to determine agent's relative position
	put down stack from hallway at position relative to stack already at site
	search for spanning block
	go to spanning block (path #12)
12	pick up spanning block
	search for fastener near door
	go to fastener near door (path #13)
13	turn to face stacks at assembly site
	go to assembly site (path #14)
14	determine pose of first stack built at assembly site to determine agent's position and thus the position of the second stack
	span first stack and second stack with the spanning block

Since this plan is a linear series of steps, no flow diagrams are given. Some interesting

decisions have been designed into this plan based on known capabilities of Marcus. For ex-

ample, Marcus gets to the base block (at the end of path #2) by first moving to the purple block at the end of path #1. This is because the base block is not visible from Marcus' starting point and it is easier to navigate around the furniture by using the other block as a waypoint, than to use obstacle avoidance to get to a goal that the agent cannot see. In a similar manner when going to the door that leads into the hallway or when going back to the assembly site after getting the spanning block, Marcus is first instructed to go to the fastener block near the doorway. This is due to the "detectability" of landmarks in both cases. Marcus' perceptual capabilities make it easier to get to his final destination by moving to the easily detectable red fastener block and then looking in the nearby area for either the door or the stack of blocks, than to try and detect the final destination from far away. A final point is that Marcus picks up the fastener block at the end of path #3 because he needs it to build the first stack at the assembly site. Part of the definition of Marcus' domain is that when stacks are assembled, they must be "held together" with a fastener.

Each of the plan steps in table 6.1 is decomposed into subtasks as shown in figures 21 - 23. Since many plan steps represent the same actions in different parts of the environment, I describe the decomposition of the plan steps in general terms and, in later sections, I discuss how these subtasks are made specific to the context in which they are executed.

The plan steps can be broadly classified into four categories, locomotion, manipulation, building and dealing with doors. The tasks in each of these categories are described in tables 6.2 - 6.5.
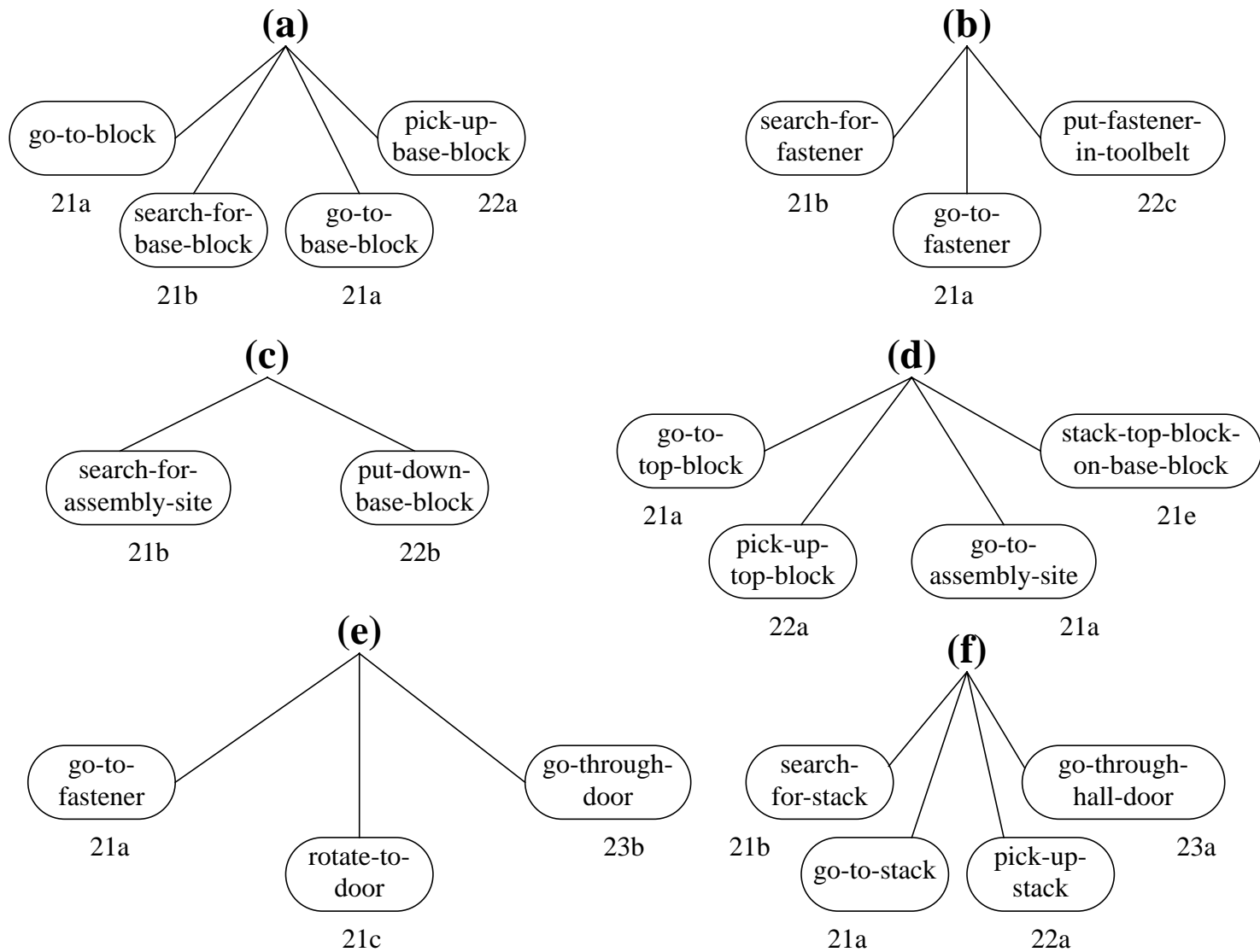
Figure 19. Top Level Task Decomposition for Marcus

(g)

search-for-stack — 21b

go-to-stack — 21a

(h)

determine-stacks-pose — 22e

put-down-stack1-relative-to-stack2 — 22d

(i)

search-for-spanning-block — 21b

go-to-spanning-block — 21a

pick-up-spanning-block — 22a

(j)

search-for-fastener — 21b

go-to-fastener — 21a

rotate-to-stack — 21c

go-to-stack — 21a

(k)

determine-stacks-pose — 22e

span-stack1-and-stack2-with-spanning-block — 21d
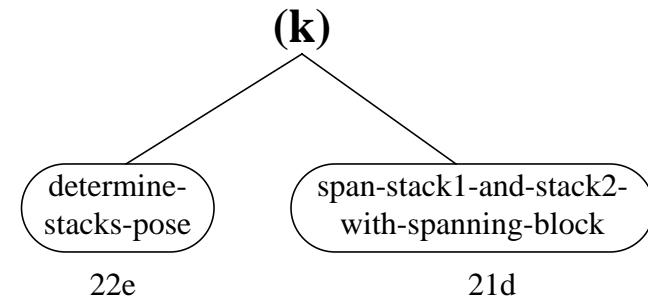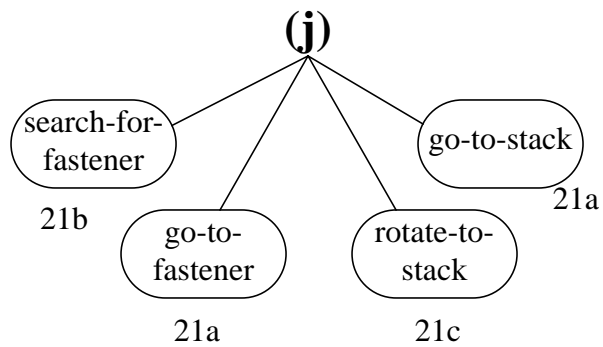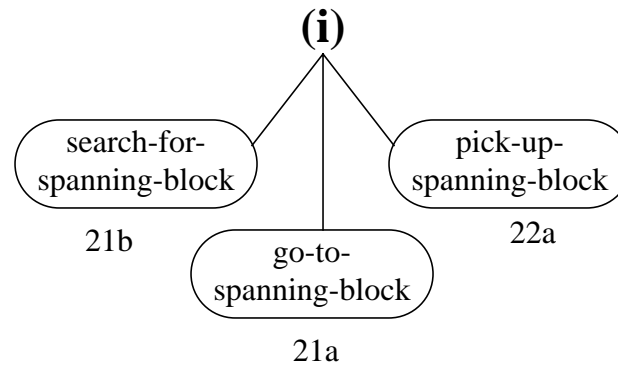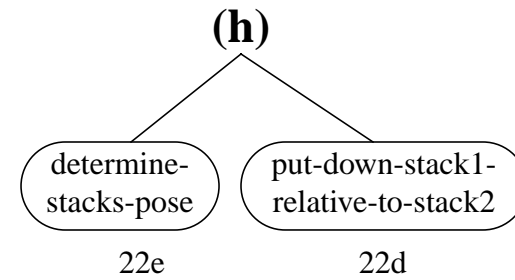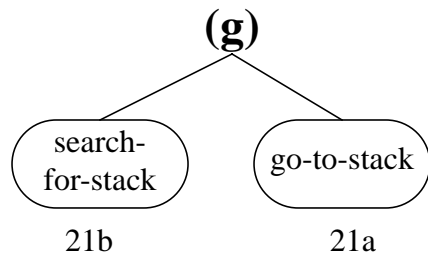
Figure 20. Top Level Task Decomposition for Marcus (continued)
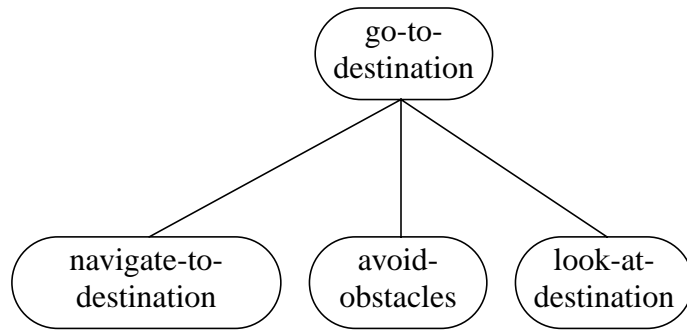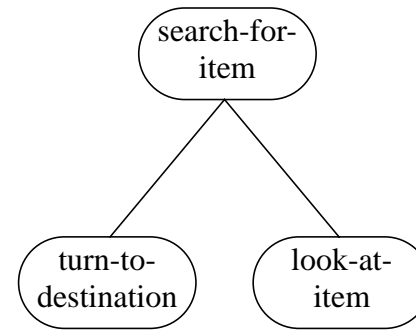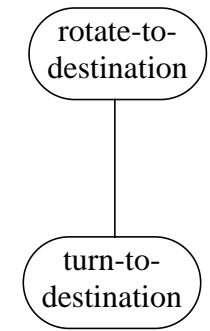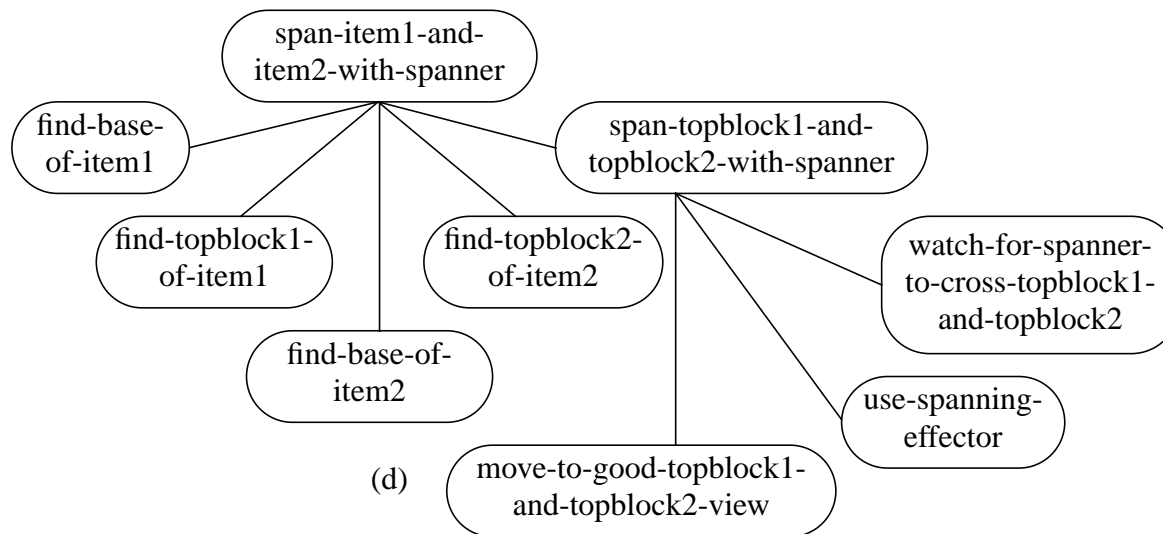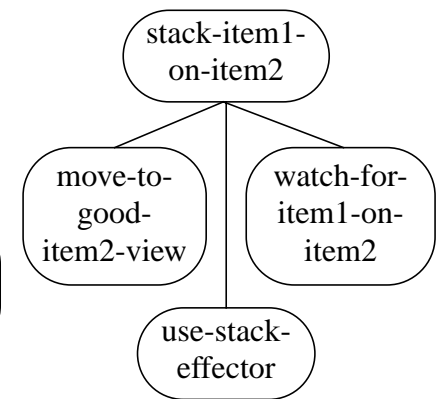
Figure 21. Locomotion and Building Task Decompositions for Marcus
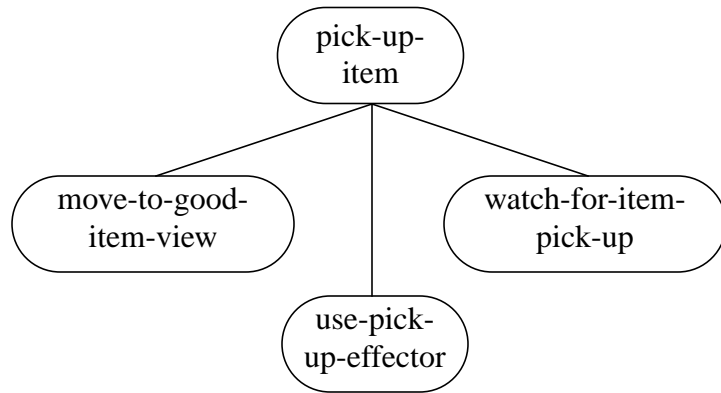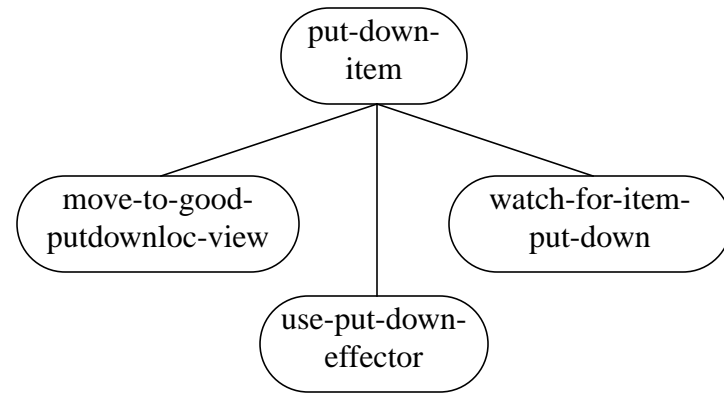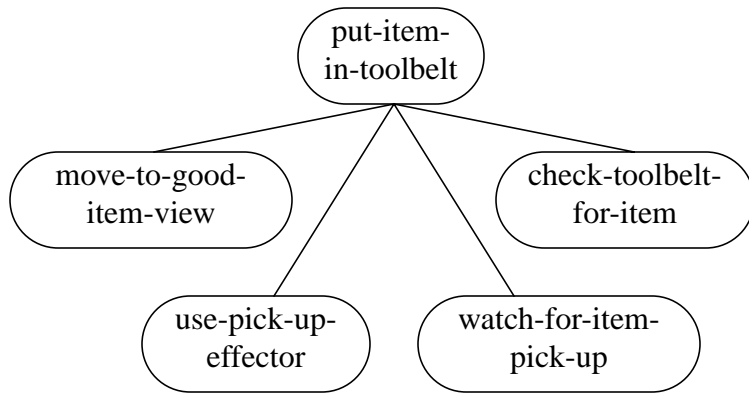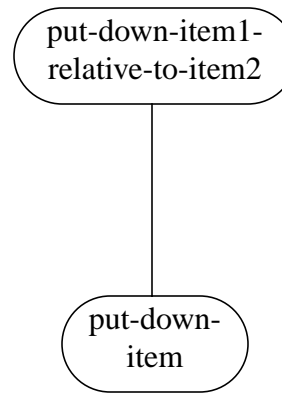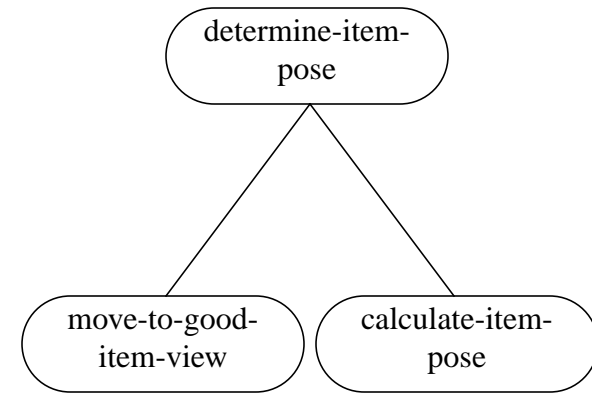
Figure 22. Manipulation Task Decompositions for Marcus
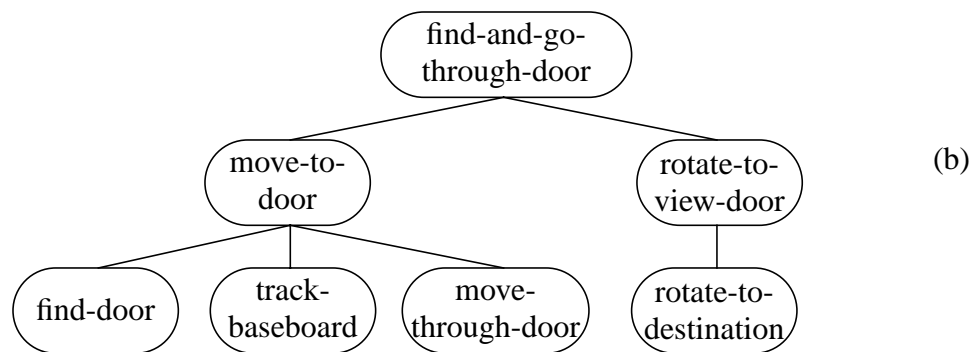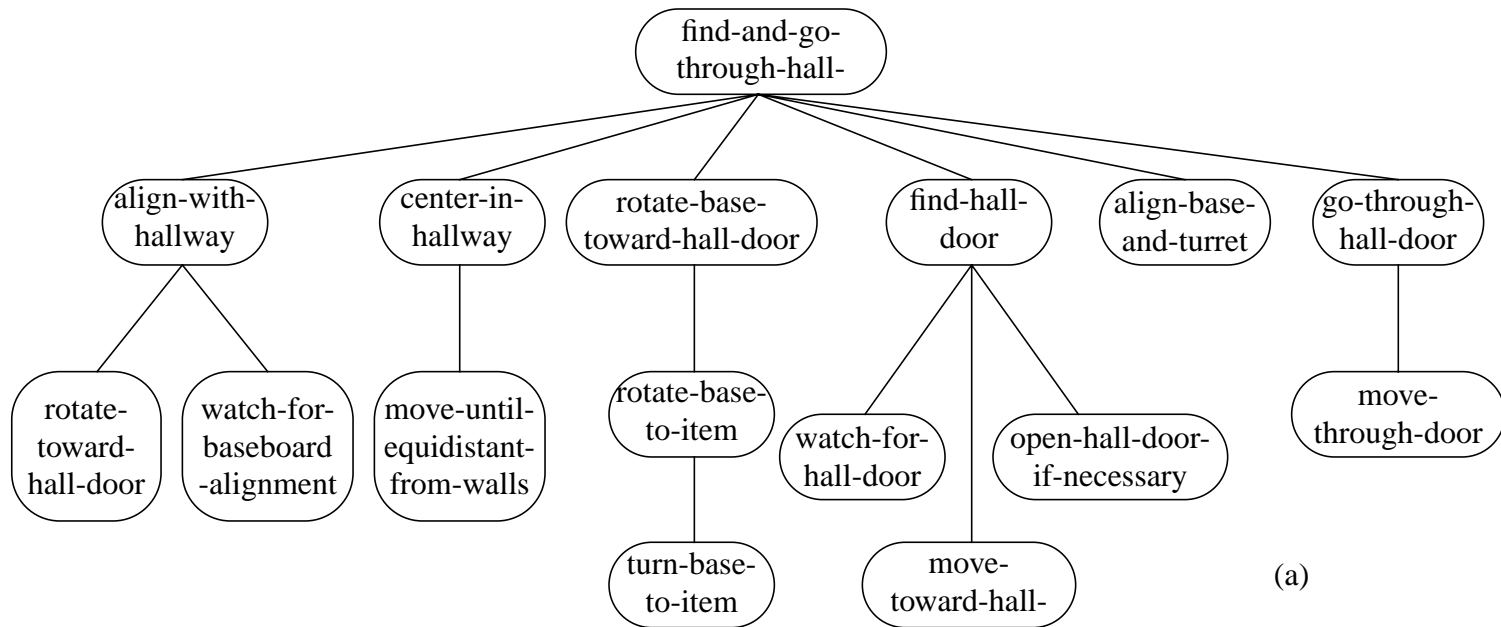
Figure 23. Door Task Decompositions for Marcus

Table 6.2: Locomotion Task Descriptions

Task Name	Description
go-to-destination	Moves the agent toward a specific destination. Consists of three parallel subtasks: • navigate-to-destination ↝ moves agent's wheels toward its destination • look-at-destination ↝ turns camera to face destination • avoid-obstacles ↝ seizes navigation control to steer around objects blocking agent's straight-line path to its destination
rotate-to-destination	Turns agent's wheel base & turret to face a particular location.: Consists of one subtask: • turn-to-destination ↝ rotates agent's body to face the destination
search-for-item	Used to find an item when it is not detected at where Marcus estimated it would be. Consists of two sequential subtasks: • turn-to-destination ↝ rotates agent to face position where item should be • look-at-item ↝ moves camera to several fixed positions relative to agent's turret to see if the item can be detected

Table 6.3: Manipulation Task Descriptions

Task Name	Task Description
pick-up-item	Picks an item up so the agent can carry it somewhere. Consists of one subtask followed by two parallel subtasks: • move-to-good-item-view ↝ moves the agent into a position where the item can be viewed well, often this means backing up. This is needed because even though Marcus does not actually manipulate objects, he needs to visually determine when manipulations have been completed. • use-pick-up-effector ↝ asks human assistant to pick up the item (draws mark on agent's laptop screen to indicate item) • watch-for-item-pick-up ↝ monitors for the item to be removed from view. Since this is a situated task (see section 1.3.2) removal of the item from view is assumed to be due to the item having been picked up.

Table 6.3: Manipulation Task Descriptions

Task Name	Task Description
put-down-item	Puts a held item at a certain location. Consists of one subtask followed by two parallel subtasks: • move-to-good-putdownloc-view → moves the agent into a position where the location for the item to be put down can be viewed • use-put-down-effector → asks human assistant to put down the item (draw a mark on the agent's laptop screen to indicate the desired position to the human) • watch-for-item-put-down → monitors for the item to appear in the view at a specified position
put-item-in-toolbelt	Picks up an item and places it on the agent's top-deck in the location designated as the toolbelt. Consists of one subtask, then two parallel subtasks, and then one more subtask: • move-to-good-item-view → same as in pick-up-item • use-pick-up-effector → asks human assistant to put the item in the "toolbelt position" on the robot's top deck (draws mark on agent's laptop screen to indicate item) • watch-for-item-pick-up → same as in pick-up-item • check-toolbelt-for-item → after watch-for-item-pick-up determines that the item has been removed from view, this task looks at the toolbelt position to see if the item has been placed there
put-down-item1-relative-to-item2	Places an object at a position specified in the coordinate system of another object. Determines the position to put down item1 as a function of the position of item2, then uses put-down-item to accomplish that.
determine-item-pose	Determines the orientation of an object relative to the agent. Consists of two sequential subtasks: • move-to-good-item-view → same as pick-up-item • calculate-item-pose → determines the orientation of the item relative to a canonical orientation

One note about the decomposition of pick-up-item and several other tasks is that move-to-good-item-view (and its variants in other tasks) is necessary because Marcus has a single camera and so must determine distances from some non-stereo cues. In this case, I use the same groundplane constraints that Bruce used. Therefore, it is important to see where an

item meets the floor because it is that point in the image that should be used to determine the distance to a item (and thus whether or not it's where Marcus thinks it should be). This may necessitate moving the wheels, possibly backing up.

Table 6.4: Building Task Descriptions

Task Name	Task Description
stack-item1-on-item2	Used to create towers of blocks. Consists of a subtask followed by two parallel subtasks: • move-to-good-item2-view → same as in pick-up-item • use-stack-effector → asks the human assistant to put item1 on top of item2 (draws mark on agent's laptop screen to indicate what is item2) • watch-for-item1-on-item2 → detects when item1 has been placed on top of item2
span-item1-and-item2-with-spanner	Places an object across the tops of two other objects. Consists of five sequential subtasks: • find-base-of-item1 → determines the position of the base of item1 • find-topblock1-of-item1 → determines the position of the block on top of construction item1 • find-base-of-item2 → determines the position of the base of item2 • find-topblock2-of-item2 → determines the position of the block on top of construction item2 • span-topblock1-and-topblock2-with-spanner → see below
span-topblock1-and-topblock2-with-spanner	(Note: this is a subtask of span-item1-and-item2-with-spanner) Consists of a subtask followed by two parallel subtasks • move-to-good-topblock1-and-topblock2-view → moves to a position where both topblock1 and topblock2 can seen in the same camera image • use-spanning-effector → ask the human to place the spanning block across topblock1 and topblock2 • watch-for-spanner-to-cross-topblock1-and-topblock2 → detects when the spanning block is both adjacent to, and above, both the topblocks

Span-item1-and-item2-with-spanner is one of the more complicated tasks that Marcus must perform. The first stage involves finding the appropriate parts of item1 and item2 to be spanned. In Marcus' domain, item1 and item2 are constructions of one or more blocks and so the parts he is interested in finding are the "top" blocks of each item. This task defines "spanning" as crossing the two top blocks with a third construction called the spanner. In order to find the top blocks, Marcus executes the task of find-base-of-item1 before find-topblock1-of-item1 and similarly find-base-of-item2 before find-topblock2-of-item2. The reader should not be confused by the use of names like find-base-of-item and find-top-block-of-item. What constitutes a "base" and a "top" block is specific to the kind of structure under consideration and them these names do not imply that these blocks are necessarily the lowest and highest blocks of the structure. Marcus spans structures that are towers and so establishing the position of the tower base allows accurate positioning of the camera to find the top (due to the groundplane constraint). However, if the structure was a collection of blocks placed side by side, the "top" of this structure would be whichever block was to be connected to the spanner and the "base" would be any block that aided in determining which block was the "top" one.

Table 6.5: Door Task Descriptions

Task Name	Task Description
find-and-go-through-door	Moves the agent from the lab into the hallway. Consists of two sequential subtask: • move-to-door ↠ determines if there is a door in view and moves the agent through it if so. If not, control transfers to rotate-to-view-door. • rotate-to-view-door ↠ changes the agent view by rotating the body to a set of positions around the door's estimated position. Transfers control back to move-to-door after rotating. Once all possible orientations have been tried, this task transfers control back to find-and-go-through-door reporting failure.
move-to-door	(Note: this is a subtask of find-and-go-through-door) Consists of a subtask followed by two parallel subtasks: • find-door ↠ searches the image for a door. If one is found, control transfers to the following subtasks, otherwise it goes to rotate-to-view-door • track-baseboard ↠ monitors the door's position (via the hallway baseboard - see section 6.4) • move-through-door ↠ drive the agent through the door
rotate-to-view-door	(Note: this is a subtask of find-and-go-through-door) Consists of a single subtask: • rotate-to-destination ↠ discussed in table 6.2
find-and-go-through-hall-door	Moves the agent from the hallway to the lab. Consists of 5 sequential subtasks: • align-with-hallway ↠ rotates the agent until its camera and wheels are facing the wall that the door to go through is on • center-in-hallway ↠ moves forward/backward until the agent is in the middle of the hallway (still facing the wall) • rotate-base-toward-hall-door ↠ turns the wheel base (but not the turret on which the camera is mounted) to be perpendicular to the wall the agent is facing and in the direction of the door • find-hall-door ↠ moves agent down the hall, toward the door, scanning the wall for the door frame • align-base-and-turret ↠ moves the wheel base to point through the door frame • go-through-hall-door ↠ moves the agent through the door

Table 6.5: Door Task Descriptions

Task Name	Task Description
align-with-hallway	(Note: this is a subtask of find-and-go-through-hall-door) Consists of two parallel subtasks • rotate-toward-hall-door → turns the agent in the direction of the wall that the door is on • watch-for-baseboard-alignment → detects when the hallway baseboard is horizontal in the image (meaning the agent is roughly facing straight toward the wall)
center-in-hallway	(Note: this is a subtask of find-and-go-through-hall-door) Consists of a single subtask • move-until-equidistant-from-walls → uses the agent's sonar to determine the position of the wall in front of and behind the agent and moves until their positions are roughly equidistant
rotate-base-toward-hall-door	(Note: this is a subtask of find-and-go-through-hall-door) Consists of a single subtask • rotate-base-to-item → turns the agent's wheel base to align with item without turning the turret (this task also has a subtask called turn-base-to-item, which actually handles the moving)
find-hall-door	(Note: this is a subtask of find-and-go-through-hall-door) Consists of two parallel subtasks followed by a single subtask • watch-for-hall-door → monitors the wall for the door frame • move-toward-hall-door → moves the agent along the hall toward the door's estimated position. Also attempts to keep the agent a certain distance from the wall (to help him see the door) • open-hall-door-if-necessary → checks if the door is closed and if so, asks the human assistant to open it. Also monitors for when the door has been opened.
go-through-hall-door	(Note: this is a subtask of find-and-go-through-hall-door) Consists of a single subtask • move-through-door → same as move-to-door in find-and-go-through-door

The final class of tasks deals with moving in and out of doors in Marcus' domain. Two aspects of the decompositions of these tasks were not initially thought to be necessary. First, the fact that separate tasks are needed to move the agent through the same door from

different sides was not anticipated. This issue is one of situated routines, specifically situated perception. The door appears different from different sides because of factors such as the agent having to detect the door at closer range in the hallway than in the lab and the unpredictable nature of what will be visible through an open door from the hallway as opposed to the lab. More details on door detection are provided in section 6.4. The important point is that intuitively similar tasks can be executed in different situations and therefore need to take account/advantage of those situations with different task decompositions.

The second unanticipated aspect of the door task decompositions is in the find-and-go-through-door task. The agent alternates between trying to detect (and then move through) the door and rotating to a new position to change the agent's view of the door. This strategy was not thought necessary until the door detection routines were created (see section 6.4).

6.2. Identify Task Roles

What are the task roles? Marcus' arch building task is more complex than the tasks of the previous agents and so the task roles are also more complex. Still, the roles are those entities that effect the agent's actions during a task. While the usual task role diagrams appear in Appendix B, figures 34 - 38, in this section I discuss the reasoning behind them. The remainder of this section describes the roles in the tasks of figures 21 - 23 and then the roles in the plan step tasks.

In general, the reason that any particular task has those roles is based on the definition of roles (see section 3.4.2) and the belief that actions take place on objects (see section 1.2.3). In other words, a task has its roles because the objects associated with those roles are the entities that effect the outcome of that task. For example, the go-to-destination task of fig-

ure 21a maneuvers the agent toward a specific location in the environment. **Destination** is a role because the location of the destination determines the goal point toward which the agent servos. Likewise, **destination** is a role in the navigate-to-destination and look-at-destination subtasks because these tasks handle the wheel and camera control to move toward, and look, at the destination, respectively. The avoid-obstacles subtask has **obstacle** and **destination** roles because obstacles effect the agent's movement toward its destination. This task is designated to handle steering around an obstacle, which it does based on the position of the destination.

Similar logic, i.e. that **X** is a role because X effects the actions taken by a task, applies to all the roles in all Marcus' tasks and so I will not reiterate the argument for each. Instead, I will discuss those tasks that have roles that are not obvious from the task name or that have interesting relationships between their roles or between their roles and the roles of their parent task.

Some roles are not obvious from the name of their task. For example, the find-topblock1-of-item1 and find-topblock2-of-item2 tasks have not only **topblock1** (or **topblock2**) and **item1** (or **item2**) roles, but a **base** role as well. The base is used to determine where to look for the topblock, based on the kind of structure with which **item** is associated. Another example is the many use-X-effector tasks, where X can be pick-up, put-down, spanning, stack, etc. These tasks ask for human assistance to perform various jobs and they need to know the various portions of the environment to refer to in their request. So, the use-pick-up-effector task has an **item** role (the same **item** in the pick-up-item task) so it can know enough about the item to ask the human to, for example "Pick up the purple block" and draw a mark on the appropriate point in the image on the laptop screen.

The put-down tasks (put-down-item and put-down-item1-relative-to-item2) have **put-downloc** roles that denote the position where the item should be placed. As we'll see shortly, these tasks need to have this role to receive instructions from the plan steps and share it with their subtasks. In the same vein, the align-with-hallway task has a **hall-door** role because its rotate-toward-hall-door subtask needs to be told the direction of the hall-door and this information ultimately comes from a plan step task.

The tasks that move the agent through doors (go-through-hall-door and move-through-door in figure 23a and find-door and move-through-door in figure 23b) have **destination** roles not mentioned in the task name. This is because each task is concerned with moving the agent through the door, to a specific destination. In the case of the find-door task, it must bind the **destination** role to an object so that move-through-door can operate.

Another important aspect of roles in Marcus' tasks is that certain roles are dependent on other roles. It is important to notice these instances because any actions that effect the entity associated with one role effect the entity associated with the other dependent roles. This means that if the agent receives information about one entity, it may need to change its information about other entities to reflect this. For example, the find-base-of-item1 and find-base-of-item2 subtasks of span-item1-and-item2-with-spanner have **base** and either **item1** or **item2** roles. The interesting aspect of this is that the base is a sub-component of item1 or item2. In other words, if **item1** were associated with a stack of blocks, then **base** would be associated with the block on the bottom. If **item1** were associated with a single block, **base** would just be that block. Similarly, the **topblock1** and **topblock2** roles of find-topblock1-of-item1 and find-topblock2-of-item2 are associated with different sub-components of item1 and item2, namely the topmost block. There is an epistemological link be-

tween these roles because anything that happens to the bottom or top block of a structure effects the structure as a whole (given that the blocks are connected).

Other role dependencies exist between subtasks of the find-and-go-through-hall-door task of figure 23a. A dependency exists between the **hallway** role of the center-in-hallway task and the **front-wall** and **back-wall** roles of its move-until-equidistant-from-walls sub-task. The front-wall and back-wall are sub-components of the hallway, i.e. they are the two walls that define the hallway. A similar dependency exists between the **baseboard** role of watch-for-baseboard-alignment and the **hallway** role of its parent task, align-with-hallway. The angle of the baseboard tells the agent something about the hallway (its alignment relative to the agent). These dependencies represent the same sort of link between these roles as exists between **item1** and **base** above. That is, information about the front-wall, back-wall or baseboard effects information about the hallway. Also, there is a dependency between the **hall-door** role and the other roles in the find-hall-door subtask (see figure 38a in Appendix B). This is due to Marcus' perceptual capabilities and is discussed in section 6.4.

For put-down-item1-relative-to-item2, the **putdownloc** role must be computed based on configuration information for a plan step (where to put item1 in relation to item2) and the **item2** role. This means **putdownloc** is dependent on **item2**. However, nothing effects the putdownloc (its just an empty spot at the assembly site) and so no information about it can effect item2. This creates a situation in which the roles are not mutually dependent. **Putdownloc** is dependent on **item2**, but **item2** is not dependent on **putdownloc**.

The final interesting issue about roles in the tasks in figure 21 - 23 is the roles that some tasks do *not* have. This is important because sometimes domain knowledge can be used in place of roles. For example, the put-item-in-toolbelt task does not have a **toolbelt** role. The

toolbelt is special because it exists on the agent's person and therefore its location is always known. Also, the put-item-in-toolbelt task cannot put the item in any other location, just the one pre-defined as the toolbelt. Since there is a single toolbelt and no need to know information about an object "out" in the world, there are no dynamic toolbelt properties to maintain (except perhaps its contents). Therefore, this task does not have a **toolbelt** role even though the toolbelt effects the task. The toolbelt merely represents a piece of domain knowledge that can be encoded into the task. Similarly, the align-base-and-turret task (of the find-and-go-through-hall-door task) has no roles. This task realigns the wheel base and the turret so that the wheels are pointing in the same direction as the camera. Since roles are aspects of the *environment* that influence the current task and this task only requires knowledge of the agent's own body (which can always be sensed) no roles are required.

At this point, the roles in all tasks except the plan step tasks (figures 19 and 20) have been explained. The roles in the plan steps are straight-forward and are shown in figures 34 and 35 in Appendix B. The reason that these tasks have those roles is the same reason that most roles of tasks of figures 21 - 23 have their roles, because those roles are the entities in the environment that the tasks are designed to effect.

What entities can fulfill those roles? The answer to this question strongly impacts the plan steps of figures 19 and 20. The roles in these tasks must be bound to specific objects[2] in the environment in order to correctly execute the plan. For example, the block role in the go-to-block task of figure 19a (the first step in the plan) should be filled by the purple block

---

2. Even though correct execution of the plan requires that the role be bound to one specific object, any perceptual routine written to recognize that object will actually recognize a set of objects. The size of this set depends on the perceptual routine and the environment. It is important to remember that it is possible for Marcus to bind a role to an "incorrect" object even when only one object in the environment can correctly fill the role. Recovery from such errors can be arbitrarily difficult depending on how far along the agent gets before the error begins to effect the plan's execution. This document does not address error recovery, but rather the design of representation systems.

at the end of path #1 in figure 18.

The remainder of this section is divided into two parts. The first part provides a table that describes the objects that are meant to fulfill roles in the plan steps. This table refers to the path numbers in the environment diagram of figure 18. The second part discusses objects that can fill the roles in the "plan implementation" tasks of figures 21 - 23. Section 6.2.1 looks at the roles in these tasks in light of the agent's capabilities and provides insight into the effect that roles in the plan step tasks may have on the design of the implementing tasks.

Table 6.6: Entities in the Environment that Fulfill Roles in Plan Step Tasks

Plan Step Task Name (role in bold)	Entity that Fulfills Role
go-to-**block**	purple block at end of path #1
search-for-**base-block**	block at end of path #2 (but see below)
go-to-**base-block**	same as previous
pick-up-**base-block**	same as previous
search-for-**fastener**	red block at end of path #3
go-to-**fastener**	same as previous
put-**fastener**-in-toolbelt	same as previous
search-for-**assembly-site**	position of ✗ at end of path #4
put-down-**base-block**	block picked up from end of path #2
assembly-site role	position of ✗ at end of path #4
go-to-**top-block**	purple block at end of path #5
pick-up-**top-block**	same as previous
go-to-**assembly-site**	base-block now at assembly site
stack-**top-block**-on-**base-block** (the stack thus created is now referred to as the stack at the assembly site or the first stack at the assembly site)	**top-block** = purple block from end of path #5
	base-block = yellow block at assembly site
go-to-**fastener**	red block at end of path #7
rotate-to-**door**	door that path #8 goes through

Table 6.6: Entities in the Environment that Fulfill Roles in Plan Step Tasks

Plan Step Task Name (role in bold)	Entity that Fulfills Role
go-through-**door**	same as previous
search-for-**stack**	purple and green block stack at end of path #9
go-to-**stack**	same as previous
pick-up-**stack**	same as previous
go-through-**hall-door**	door that path #10 goes through
search-for-**stack**	stack now at assembly site
go-to-**stack**	same as previous
determine-**stack**s-pose	same as previous
put-down-**stack1**-relative-to-**stack2**	**stack1** = stack picked up in hallway
	stack2 = same as previous
search-for-**spanning-block**	green block at end of path #12
go-to-**spanning-block**	same as previous
pick-up-**spanning-block**	same as previous
search-for-**fastener**	red block at end of path #13
go-to-**fastener**	same as previous
rotate-to-**stack**	first stack at assembly site
go-to-**stack**	same as previous
determine-**stack**s-pose	same as previous
span-**stack1**-and-**stack2**-with-**spanning-block**	**stack1** = same as previous
	stack2 = second stack at assembly site
	spanning-block = green block from end of path #12

The roles in the plan step tasks should be filled by specific objects in the environment. The exception is the **base-block** role that is used starting with the second step in the plan. This role can be bound to either a yellow or a green block and even though the map of figure 18 indicates that a yellow block is at the end of path #2, Marcus does not have this infor-

mation. He only knows that a base-block is there and that he should get it. Section 6.4 discusses the perception of base-blocks in more detail.

Now that we have looked at the roles in Marcus' plan steps, we can examine the entities that can play roles in the tasks that implement those steps. The roles in the parent tasks of figures 21 - 23, i.e. the root node of each decomposition, can typically be filled by any object in the environment (section 6.3 discusses how specific objects are selected). For example, the **destination** in go-to-destination can be any entity the agent needs to move toward. The reason that the roles in these tasks can be filled by so many objects is that these tasks are meant to be executed many times, by various plan steps, in different situations and must work properly in each. The exceptions to this are the **door** and **hall-door** in the door tasks of figure 23, which can only be filled by doors.

There are also entities that can fulfill roles in tasks "below" the level of abstraction of the "root" tasks. For example, the **obstacle** role in the avoid-obstacles subtask of the go-to-destination task can be filled by any object that is blocking the agent's path to its destination.

The span-item1-and-item2-with-spanner task of figure 21d has **base**, **topblock1** and **topblock2** roles that are meant to be filled by the lowest block in a stack, the highest block in stack1 and the highest block in stack2 respectively. The find-and-go-through-hall-door task of figure 23a has a number of additional roles that need to be bound. Two subtasks have **hallway** roles that can be played by any hallway in Olsson hall. The **baseboard** role in the watch-for-baseboard-alignment subtask can be bound to any baseboard strip in the Olsson Hall hallways, but it should be the portion in the hallway nearest the agent and on the same side of the hall as the hall-door. The **front-wall** and **back-wall** roles in the move-until-equidistant-from-walls subtask can be played by any entities that reflect the agent's

front and rear sonar (presumably the walls of the hallway). Finally, the **destination** role of the go-through-hall-door and move-through-door subtasks can be played by any point beyond the door frame (in this case, 1 m beyond the door frame is used).

Move-through-door (figure 23b) has a **destination** role that can be played by the baseboard strip beyond the door in the hallway. Similarly, the **baseboard** role in track-baseboard can be played by the portion of the baseboard visible through the door. The **destination** in find-door can be played by any position on the other side of the door and is passed to move-through-door, which maintains that position using the baseboard.

6.2.1 Observations about Roles

For the leaf tasks of Marcus' decomposition in figures 21 - 23, the entities that can fulfill the roles depend on the agent's capabilities. Marcus can manipulate blocks[3], so for example, the **item** role in the use-pick-up-effector subtask of pick-up-item (figure 22a) can *only* be filled by a single block. This is true for all the **item** roles in the leaf subtasks of the manipulation tasks of figure 22 (pick-up-item, put-down-item, put-item-in-toolbelt, put-down-item1-relative-to-item2 and determine-item-pose). Many of the roles in the building tasks of figures 21d and 21e can also be filled only by single blocks. The **base**, **topblock1** and **topblock2** roles of span-item1-and-item2-with-spanner's subtasks, obviously, are meant to be bound to particular blocks on the extent of structures bound to the **item1** and **item2** roles. The **spanner** in use-spanning-effector must also be played by a single block because this task uses Marcus' manipulation capabilities. Similarly, since the **item1** role in the use-stack-effector subtask of stack-item1-on-item2 will be manipulated by the agent's

---

3. Since Marcus' manipulator is, in fact, a human, this decision is somewhat arbitrary. However, it allows illustration of the use of hierarchical representation to separate information into data that is pertinent to, and only known by, different components of the agent's architecture.

effector, it must also be played by a block.

However, the parent tasks of these leaf tasks have no such restrictions on their roles. For example, Marcus needs to be able to pick up stacks of blocks, so the **item** role in pick-up-item should be capable of being bound to a stack of blocks. The next section discusses how the **item** role can be bound to different entities in parent and child tasks.

Marcus' domain shows that the entities that can fulfill roles can influence the design of tasks. Since the plan steps tasks are meant to be implemented by the tasks of figures 21 - 23, which are reused throughout the plan, the designer will want to make these implementing tasks as general as possible. This means that they should be able to operate with their roles bound to many different entities, so that they can be used in various situations. However, the designer may also use domain constraints to know the possible entities to which a task might be asked to bind its role(s). The more of this knowledge the designer can apply, the more "situated" the skills will be. The trade-off is between the efficiency of situated skills and their inflexibility to novel situations. Marcus makes these sorts of trade-offs in the implementation of the tasks of figures 21 - 23. For example, the go-to-destination task is flexible because it can operate with most any entity bound to its destination role. The find-and-go-through-hall-door task is less flexible because can only operate properly if its hall-door role is bound to a door in Olsson hall.

6.3. Representation

What role bindings are shared between tasks? We examine the roles shared between tasks because role sharing is a primary motivation for developing a representation for a given role (as discussed in section 3.5.2). One of the most common forms of role sharing in

Marcus' task is sharing of roles between the plan step tasks and the locomotion, building, manipulation and door tasks of figures 21 - 23. This sharing is how a specific plan is created from the general tasks that the agent can execute. This type of sharing for the arch building plan is summarized in table B1 of Appendix B. For example, the go-to-block and go-to-base-block tasks of figure 19a have **block** and **base-block** roles, respectively. These tasks are implemented by the go-to-destination task of figure 21a by having its **destination** role be associated with the same object as the **block** (or **base-block**) role.

A key to understanding this form of role sharing in Marcus' task is to realize that unlike previous agents, Marcus' actual plan is shown in the task decomposition (figures 19 and 20). Bruce, for example, can follow any route plan by changing the bindings of the **destination** role in his move-to-destination task and thus his task decomposition applies to all plans he can execute. The complexity of the structures that Marcus' skills allow him to assemble make it impractical to create a task decomposition that encompasses all possible plans. This means that there needs to be some standard way of taking the specific manipulations denoted by a plan (in this case, the arch building plan) and implementing them with Marcus' general purpose assembly skills. This is done by mapping roles in the plan step tasks to the roles in the tasks of figures 21 - 23. For example, the go-to-block task is implemented by the go-to-destination task by having its **block** role and the **destination** role in the go-to-destination task refer to the same object.

Various other forms of role sharing exist in the tasks decompositions of figures 21 - 23. Consider the go-to-destination task decomposition of figure 21a. Observe that all tasks have **destination** roles. This means that destination is a shared role since all tasks need it to be bound to the same entity. There are two important classes of role sharing exhibited

here. The first is sharing between parallel subtasks. The subtasks benefit from role sharing by not having to maintain independent copies of the role information. The second is parent/child role sharing (go-to-destination shares **destination** with its children). The benefit here is that information stored in the parent task can be abstracted to the role used by the child task. In other words, even though the roles in the parent and child tasks are bound to the same object, different information about the entity bound to the role may be computed for each task. The parent task may wish to provide the child task with a portion of the information it knows about the entity in order to simplify the computation done by the child. This will become more clear as more examples are discussed.

Consider the role sharing of two other tasks from figure 21. In the search-for-item task (figure 21b), notice how both search-for-item and look-at-item have **item** roles. This role is shared between these task, but also with the turn-to-destination task. Recall that the names of roles need not be the same for two tasks to share a role. The **destination** role in turn-to-destination is also played by the entity playing the **item** role, because turn-to-destination must rotate the agent toward the entity if the camera cannot be panned far enough to observe it. The reader may question why this task is not called turn-to-item and the answer is because the task was designed as a subtask for rotate-to-destination (figure 21c). Here, we can see that both these tasks have **destination** roles that they share. The fact that turn-to-destination can be used in other tasks makes its name seem inconsistent in those cases.

The three tasks mentioned so far exhibit the parent/child form of role sharing mentioned above. This form of role sharing occurs in all the other tasks of figures 21 - 23. This only makes sense because the parent task is accomplished by its subtasks acting on, or with respect to, the entity playing the role in the parent task. From now on, except where noted, it

will be understood that the roles of any parent task are shared with some subset of their children. Interested readers can easily see this sharing in the task role diagrams of Appendix B (figures 36 - 38) where similar names in parent and child tasks indicate shared roles. The reader may have noted that this parent/child role sharing is similar to the plan step/implementor task role sharing. The **block** in go-to-block should be bound to the same object as the **destination** in go-to-destination, which is bound to the same object as the **destination** in navigate-to-destination. The important issue is that each of these tasks may retain different information about the entity associated with the shared role binding.

Another important form of role sharing can be observed in the center-in-hallway subtask of find-and-go-through-hall-door task (figure 23a). This task has a **hallway** role and its subtask has **front-wall** and **back-wall** roles. The **hallway** role is actually a "combination" of the roles in the move-until-equidistant-from-walls subtask. The position of the entities playing the **front-wall** and **back-wall** roles combine to form the agent's notion of the position of the hallway (relative to the agent). The **hallway** role can store the hallway's position as an abstraction of the information present in the subtask's roles. This form of role sharing (where multiple roles, possibly in multiple tasks, are "shared" by a single role in yet another task) often occurs when roles are dependent on other roles as discussed in the previous question. One role "combines" data from other roles into a more abstract view of the entities associated with those roles. The degenerate case, when the number of roles that combine to make up another role is one, is equivalent to the parent/child form of role sharing. Different information, i.e. a different abstraction, is contained in each role. For example, the **item** in the pick-up-item task may be bound to a construction of blocks, while the **item** in its subtasks must be bound to a single block (such as the one that the agent should

maneuver with respect to in order to have a good view of the impending pick-up). Since the agent's capabilities force the item role in the subtasks to be associated with a single block, the designer should make use information in pick-up-item's **item** role to determine the block in the structure to bind to the other **item** role. Roles that are shared in this way and need representation, must have their representations "linked" so that information about bound entities can be captured in both representations (see section 6.5).

The remainder of the answer to this question concentrates on shared roles that are not present in the parent task. The span-item1-and-item2-with-spanner task (figure 21d) has subtasks containing **base**, **topblock1** and **topblock2** roles. The **base** role occurs in both the find-base-of-item1 and find-base-of-item2 tasks, but it is actually bound to different entities in each case. **Base** only remains bound long enough to establish the position of **topblock1** or **topblock2**. So, **base** is shared by find-base-of-item1 and find-topblock1-of-item1, but not by find-base-of-item2. The **topblock1** and **topblock2** roles should be bound to the same entities in the find-topblock tasks and the span-topblock1-and-topblock2-with-spanner task (so that the same blocks are spanned) and thus these roles are shared.

The put-down-item1-relative-to-item2 task (figure 22d) is interesting because it has a role (**item2**) that is not present in any of its subtasks. This is because the entity playing the **item2** role is used only to establish the **putdownloc** role. It is this role that is needed by the put-down-item subtask, not **item2**.

The **hall-door** role in the find-and-go-through-hall-door task (figure 23a) appears in many of its subtasks. All these tasks either attempt to bind the role or move the agent with respect to the role's associated entity. Therefore, the tasks all need **hall-door** to be associated with the same entity and thus the role is shared. There are also two subtasks (align-

with-hallway and center-in-hallway) that are concerned with the agent's position relative to the hallway and thus have **hallway** roles that they share. What is interesting is that the **hallway** role does not appear to be shared with those task's subtasks. However, the **hallway** role in align-with-hallway is defined in terms of the **baseboard** role in watch-for-baseboard-alignment. That is, while the **hallway** role is meant to be played by a hallway in the building, the visual feature that can be recognized and used for this task is the baseboard. Marcus aligns himself with the hallway by turning until the baseboard appears horizontal in the camera image. Information about the current angle of the baseboard can be used to determine information about the current alignment of the hallway. So, for the purposes of this task, the **hallway** and the **baseboard** form a shared role. The **hallway** role in center-in-hallway is actual a "combination" of the **front-wall** and **back-wall** roles of its subtask as discussed above.

One final example of role sharing is in the find-and-go-through-door task (figure 23b). One of its subtasks, move-through-door, has a **destination** role representing the location at which the agent can consider itself "through" the door. Find-door determines where that location is by detecting the door and determining a position beyond it. Therefore, these tasks share the **destination** role. The **baseboard** role in the track-baseboard subtask is associated with a recognizable feature whose position can be determined. Find-door binds the **destination** to the baseboard because it can be seen through the door and can be tracked. This means that **baseboard** and **destination** are bound to the entity and so shared by all find-and-go-through-door subtasks. Likewise, the rotate-to-destination subtask of rotate-to-view-door shares binds its **destination** role to the same entity as the **door** role and so they form a shared role. Since the door is the object that the agent wants to be facing, it is natu-

rally the destination of the agent's rotation.

In summary, we have seen two types of role sharing. One is when sequential or parallel tasks each have roles that should be bound to the same entities. The second occurs when information about entities bound to one or more roles, in one or more tasks, effect an entity bound to a role in yet another task. This most often occurs between the roles of a parent task and its subtasks with the parent holding some more "abstract" view of the entity bound to the its role (such as the **hallway** being defined by sonar readings of the front and back walls).

What information about the entity bound to a task role is needed for that task? This question examines the information about the entities bound to task roles that is needed by the tasks. This information, along with the role sharing, will determine if a role should be explicitly represented by the agent architecture.

For the vast majority of tasks, the position of the associated entity is needed. As discussed in the chapters on previous agents, position is needed to determine any other information about an entity because it is needed to direct the perception system. One exception to this is the align-with-hallway subtask of find-and-go-through-hall-door. For this task, all that is needed is the orientation of the hallway relative to the agent. The position of the hallway (relative to the agent) is not important and, in fact, the subsequent center-in-hallway task will deal with the agent's position. Two other interesting tasks are determine-item-pose and put-down-item1-relative-to-item2. They require not only the position, but the pose of the entities associated with their roles. Determine-item-pose uses the position of the item to determine its pose and put-down-item1-relative-to-item2 uses this pose to calculate its **put-downloc**.

For tasks that require the positions of entities bound to roles, positions may be in different forms. As discussed in chapters on previous agents, e.g. section 4.3, it is often more accurate to maintain positions of objects in ego-centric coordinates. The tasks of figures 21 - 23 can use ego-centric positions because they control effectors. The agent's effectors are fairly centered on its "body cylinder", and since they are at the origin of the ego-centric system, such coordinates are a natural way to control them. The agent's raw sensor data is also easier to convert to ego-centric coordinates than to non-ego-centric coordinates because less information about the environment is needed to estimate ego-centric coordinates (non-ego-centric coordinates require external reference points to "triangulate"). By contrast, the roles in the plan steps should specify positions for the entities with which they should be associated, in non-ego-centric coordinates. Most of the entities involved in the plan are not perceivable from the agent's starting position, so the agent needs position data from some other source to plan routes. These positions are encoded in a map and therefore are relative to some map coordinate frame. The map could contain ego-centric positions for all objects (based on the agent's known starting position and orientation), but maintaining those positions accurately throughout the arch building task would be extremely error prone since Marcus will not be able to see most of the objects, most of the time (and hence their positions will be dead-reckoned). It is more effective to have the map in some coordinate system with a fixed origin and transform from those coordinates to ego-centric coordinates as new objects become pertinent to the agent's task. Marcus' mechanism for doing this is discussed in section A.2.1 of Appendix A.

For what roles would it be useful to develop an explicit representation? The answers to the previous two questions help determine what roles should be (explicitly) represented.

First, let's look at the roles in the tasks of figures 21 - 23. Many roles in these tasks need representation because the entity associated with the role will not always be within the field of view. These roles are the **destination** in go-to-destination (figure 21a), the **item** role in search-for-item (21b), the **destination** in rotate-to-destination (21c), the **hall-door** in find-and-go-through-hall-door (23a) and the **door** in find-and-go-through-door (23b). Recall that the **destination** in the turn-to-destination subtask will be played by the same entity that plays the **item** in search-for-item and so it will have a representation by virtue of the fact that **item** will be represented.

Another important reason to represent roles is due to role sharing. Shared roles can use representation in two ways. First, multiple tasks can take advantage of the binding and maintenance of a role by sharing a physical data structure. This means only one task needs to do the binding/maintenance work and all others can just read from the shared data structure. Second, representation for shared roles can assist in inter-task communication. This is discussed in more detail in section 6.5, but having an explicit representation allows two tasks to communicate information about an entity in the environment.

The roles of the span-item1-and-item2-with-spanner task (21d) should have representation because they are shared among the subtasks (and also with the parent task). For example, the **topblock1** and **topblock2** roles are bound by two different tasks and then passed to the span-topblock1-and-topblock2-with-spanner task (which shares them with its subtasks). The **item1** and **item2** roles are similarly shared between the parent and the tasks that bind the **topblock** roles. The **spanner** should be represented as well because it is shared by four different tasks at different levels of the decomposition. The **base** in find-base-of-item1 is shared with find-topblock1-of-item1 and between find-base-of-item2 and find-

topblock2-of-item2. This is somewhat an artifact of the decomposition and one could imagine a find-topblock1-of-item1 task that "internally" computed the base position, and used it to determine the topblock position with the base computation being invisible outside the task. Here the task decomposition has created a role that might not have needed representation except that the dependency between **base** and **item1** (or **item2**) means that a representation of the base makes a useful place to pass information about a portion of item1 (or item2) to the find-topblock tasks.

The **item1** and **item2** roles in stack-item1-on-item2 (21e) should be represented due to the fact that they are shared between the subtasks and with the parent task. The same argument applies to the **item** role in pick-up-item (22a), put-item-in-toolbelt (22c), and determine-item-pose (22e) as well as the **item** and **putdownloc** roles in put-down-item (22b). The **putdownloc** and **item1** roles in put-down-item1-relative-to-item2 (22d) both should be represented to facilitate communication with the child subtask (remember that **item1** maps to **item** in the child). The **item2** role does not necessarily need representation because it is only needed to compute the position of the putdownloc. However, it may be useful to represent **item2** so that it can be passed in from some other task (like a plan step). Marcus represents **item2**.

Although the **hall-door** role in find-and-go-through-hall-door (23a) is already represented because it is typically outside the field of view until the end of this task, the fact that it is extensively shared with its subtasks further motivates its representation (remember that it also maps to the **item** role in rotate-base-to-item and the **door** role in move-through-door).

The find-and-go-through-door task (23b) has a **door** role that is shared amongst most of

its subtasks and so this role needs representation. The **destination** role of the move-through-door task should have representation because it forms a shared role with the **destination** in find-door (where its position is initialized) and the **baseboard** in track-baseboard (where its position is maintained).

Many other roles in Marcus tasks appear to not need representation. Some of them are represented for various reasons. The **destination** role in the move-through-door subtask of find-and-go-through-hall-door may not need representation, as it is played by some entity that should always be visible through the door as the agent moves into the room. However, **destination** should have a representation because it is shared with the go-through-hall-door task that determines **destination**'s position based on the hall-door. Also, the **destination** is just a point in space (which goes against the types of entities that should be associated with roles in section 1.2.3) that the agent needs to dead-reckon to and so some state is needed to hold how far the agent has moved. This is done because the agent can't be sure what will be visible through the door and so its easier to just dead-reckon into the room. The **door** role does not need representation in the move-through-door task because during the task, the door is always within sensor range. The agent can always sense the necessary information about the door, i.e. the edges of the door frame, with its sonar. This information can be used to maneuver the agent through the door, but it is difficult to use it to maintain the door's position since the azimuth information provided by the sonar is fairly coarse. Since the agent is only really concerned with getting to the **destination** at this point, it need not try and maintain the **door** representation. So, while the **door** is still a role in this task because its associated environmental feature effects the task execution, its not represented by this task. This is interesting because **door** (which will not be represented) maps to **hall-**

door (which is represented). This seems contradictory, but remember that the agent does not need any information about the hall-door (only the destination), and so even though **hall-door** conceptually maps to **door**, the designer need not communicate any information about the hall-door to the task if it is not needed.

The case for representation of the remaining roles in find-and-go-through-hall-door's subtasks (**hallway**, **baseboard**, **front-wall** and **back-wall**) is not clear-cut. The **hallway** role is common to two different subtasks, but they do not really exchange any information about the hallway. One subtask is concerned with the orientation of the agent and the other with the agent's position. The aspects of the hallway that must be sensed to complete these tasks are different and so one subtask's sensory data cannot contribute to the other. Therefore, the **hallway** role probably does not require representation. Note that although this is contradictory with the previous discussion about the hallway role as a shared role, it is often the case that there are arguments both for and against representation of a particular role. In the case of the **hallway**, no representation is used because the data associated with the roles in the subtasks is not needed outside those subtasks and so placing some form of it in the **hallway** role is unnecessary. The **baseboard** in the watch-for-baseboard-alignment subtask of align-with-hallway probably does not need representation either. While aligning with the hallway, the agent turns toward the hall-door until the baseboard appears horizontal in the image. Since the position of the hall-door controls Marcus' rotation, the base-board is only needed to tell when to stop. This highlights the possible iteration between the perception and representation steps of the methodology. Marcus does not actually run a perceptual routine to detect the baseboard and determine its angle. He uses a situated routine that just determines if the baseboard is horizontal in the image. In other words, if the baseboard is

not horizontal, Marcus will not even detect it. Since this role is not shared, the required role information is always available to the agent's sensors (given that not detecting the baseboard is ok) and previous orientations of the baseboard are not helpful (and can't be determined), the **baseboard** should not be represented. However, the baseboard perception could be changed to calculate the rate of change of the baseboard angle and estimate what the current angle should be in each image. This would mean having to detect the baseboard at a variety of angles. but would provide insurance against sensor errors that might cause Marcus to "miss" the horizontal state. This would allow him to turn back, but it would also require some representation to store previous role information. However, given the environmental context of this task (a brown strip against a mostly featureless white wall), perceptual errors are rare and the overhead of representing the baseboard is not necessary. Similarly, the **front-wall** and **back-wal**l roles of the move-until-equidistant-from-walls task do not need representation because they are not shared and the required information about the entities bound to those roles can always be gathered from the agent's sensors. The agent can always determine the current positions of the walls via sonar.

Representation for the **obstacle** role in the avoid-obstacles subtask is also not straightforward. The argument against obstacle representation in Bruce was that representation would prolong the time to recover from a false obstacle identification, of which there were many. The argument for representation was that inevitably, obstacles pass outside of the sensory field before the agent is actually clear of them. Representation would allow the agent to remember that the obstacle was there and fully move beyond it. Marcus has an advantage over Bruce in that he has a 360° sonar ring, thereby increasing his sensory field. However, it is more difficult to distinguish the destination from an obstacle with sonar than it is with

vision and so having a memory of previous obstacle (or destination) positions might help distinguish the two. In the end, Marcus' capabilities provide for a limited representation for **obstacle**. Marcus does not need to maintain information about any particular obstacle because he can continually rebind the role to any entity that is currently an obstacle. This may be the same object or it may be a new one, but it doesn't matter as long as it is an obstacle. When the role is bound and Marcus subsequently tries and fails to detect any obstacles, he should not immediately forget about the entity bound to **obstacle** and start going back toward the destination. Instead he should continue on his present (obstacle avoiding) course for a short period of time, in order to get "past" any obstacle that might be outside the sonar sweep[4], but still in his path to the goal. After this time interval, Marcus should forget the current association between **obstacle** and its object ("unbind" the role). Marcus can use the position of the obstacle when the role is bound to determine the motor commands to get around it. He need not maintain any information about the obstacle because once he chooses an avoidance course he follows it for a while, independent of any data about the specific obstacle. If Marcus has discontinued the association between **obstacle** and its previously associated object, but that object is still an obstacle, Marcus can bind the **obstacle** role to the same object and start the whole process again.

Finally, there are the roles in the plan step tasks (figures 19 and 20). I argue that the roles

---

4. It may appear that nothing could be outside the agent's angular sensor range when the agent has a 360° sensory view. While this may be true, Marcus does not use all his sonars. He uses only the front 4 sonars because he has a simple obstacle avoidance algorithm (since the purpose of this thesis is not to develop sophisticated motor control strategies). A possible improvement would be to use just the sonars in front of the agent to monitor where it is heading and some sonars facing the direction of the destination to tell when the agent has "cleared" an obstacle. The problem with this is that Marcus' encoders have a high rotational error, compared with their straight-line dead-reckoning performance. When turning to avoid an obstacle, especially a close one that requires rapid turning, the goal's angular position can quickly become incorrect. Since angular position determines which sonar(s) are appropriate to use, this is a serious problem. In fact, this causes a problem even when trying to reacquire the destination with Marcus' camera. I discuss that problem in section 6.5.

in all the plan steps should be represented. As shown previously, each role in the plan step tasks maps to some role in the tasks that implement that step. Since many of these roles are represented, it is reasonable to represent the roles in the plan steps in order to facilitate communication between the plan step tasks and the tasks of figures 21 - 23. At this point, no further argument is made, but I will return to this issue in section 6.5.

How often should that information be verified? For the roles of the locomotion, building, manipulation and door tasks, role information is used for effector control and, as such, needs to be verified at the rate of the effector control loop. However, as we will discuss in the next section, there are situations in which the entity bound to a task role will be "too complex" for the perception system to maintain the role information and effector control rates. This can also be the case for the entities bound to the plan step roles. In section 6.5, I will show how the requirements of the agent's task lead to a representation design that addresses this problem.

6.4. Perception

What information can be extracted from the environment to recognize the entities that should be bound to the current task's roles? Marcus' perceptual needs can be divided into detection of blocks and detection of doors. I begin by discussing perception of doors, which is complex and caused several iterations back to the task decomposition question. The door detector is a situated skill [70] that uses' Marcus' edge detection capability to detects patterns of edges that correspond to doors in Olsson Hall. The images in figure 24 were generated during the find-and-go-through-door task that moves Marcus from the lab into the hallway. They show how Marcus' edge detection and colored blob finding capabilities can be used to find doors. Figure 24a shows an edge image, with overlaid graphics, during the

initial door location process. Figure 24b shows the tracking involved in moving through the

(a)   (b)
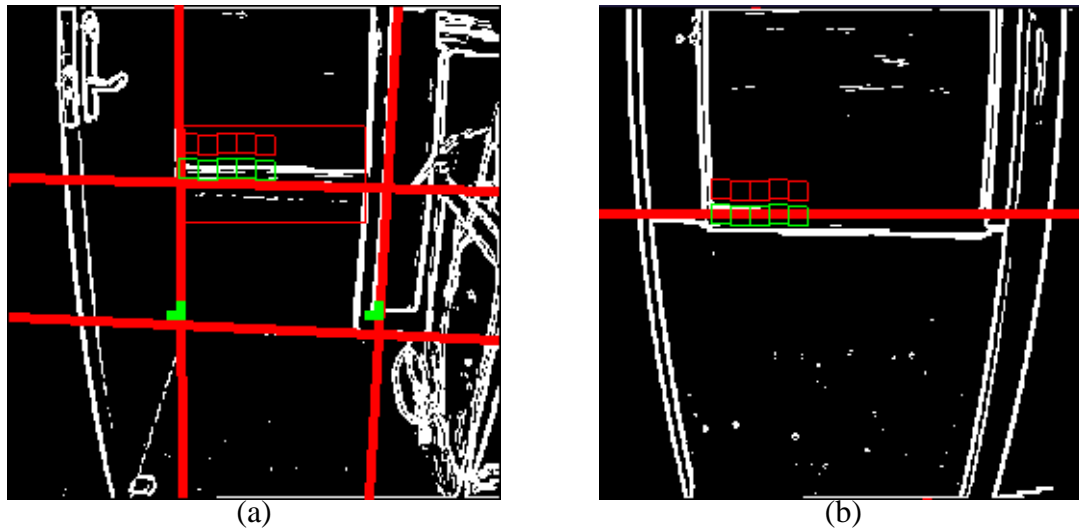
Figure 24. Door Location and Tracking

door frame. Detecting a door from the inside of a room (as opposed to the hallway) can be

performed as follows. First a Hough transform [24] is performed to search for (nearly) hor-

izontal lines in the lower half of the image. The line is meant to be the edge between the

carpet and the door frame and it is visible whether the door is open or closed (if the door is

closed, the line is actually formed by the bottom of the door). This line is shown by the low-

er of the two red, horizontal lines in figure 24a. Note that the lines in this figure are drawn

across the entire image for display purposes and do not represent the length of the actual

detected edge. Also, some lines (like this one) are obscure most of the white edge pixels

because they are drawn over them. Once the lower line is found, a second Hough transform

searches the image area above the first line for a second (nearly) horizontal line. This line

should be the baseboard on the opposite side of the hallway from the door and will be vis-

ible only if the door is open. This serves to detect if the door is open or closed. While this

may seem a simple means of determining if the door is open, it is fairly effective because

when the door is closed, it forms a fairly featureless surface on which edges are rarely de-

tected. When the door is open, the contrast of the brown baseboard against the white hall-
way walls forms a strong edge that is rarely missed. This is an example of a situated routine
(see section 1.3.2).

Once the two horizontal lines have been established, two more Hough transforms search
for vertical edges in the left and right half of the image. The detected edges in figure 24a
are shown by the vertical red lines. These edges represent the edges of the door frame. The
intersections of the vertical lines with the lower horizontal line (denoted by the green marks
on or near those points) are used to determine the distance and direction to the edges of the
door frame.

Marcus determines if the distance between the two intersections is large enough for him
to fit through. If it is not, then this cannot be a door. To further verify that this edge pattern
is indeed a door, Marcus performs a search between the vertical lines and around the higher
horizontal line for the brown of the baseboard with white wall above it. He uses the same
technique used to detect adjacency between color blocks (discussed below). Note that pre-
viously, Marcus had just been looking for edges, but now is concerned with color. The area
searched is shown by the thin red box around a portion of the higher horizontal line in figure
24a. The small green and red rectangles within this region represent regions actually ana-
lyzed for the colors brown and white. All of this is performed by the find-door subtask of
find-and-go-through-hall-door. Since this task has already detected the baseboard, it can
bind the **destination** role that it shares with move-through-door. This role is bound to a
point that is as far away as the baseboard and along an azimuth that is midway between the
azimuths of the two intersection points.

Once the door has been detected, perception of the roles in this task can change because

the information the agent needs changes. The move-through-door subtask drives the agent toward the position stored in the **destination**. This subtask can use the sonar sensors slightly left and right of the agent's forward direction to steer the agent away from the door frame edges while he moves forward. Assuming he is roughly in front of the door when the find-and-go-through-door task begins, which he must be in order to detect the door (see below), this technique will allow him to "wiggle" through. An interesting point here is that the find-door subtask detects the door visually, while the move-through-door subtask detects the door using sonar. This is a shared role meaning some form of sensor fusion must be going on between the role information computed by the two tasks. In fact, shared roles are supposed to be represented so that role information doesn't have to be computed by each task that uses the role. However, it turns out that if the door is detected by find-door (visually), it will always be in perceptible to the move-through-door task (using sonar) for the duration of time that is important (before the agent passes through the doorway). Since the agent can always detect the door with sonar, it is not necessary to use the **door** representation in the move-through-door task. The importance of this is that initial task decompositions had the move-through-door subtask determine its destination based on the shared **door** role. However, since move-through-door does not update the door role's position, it seemed better for the find-door role to bind the **destination** and share it with move-through-door[5].

Once the door (and hence the baseboard) has been detected, the track-baseboard subtask can maintain the position of the baseboard so that move-through-door can servo on its position. Remember that move-through-door's **destination** is shared with track-baseboard's

---

5. It is a somewhat arbitrary choice to have the find-door skill bind the **destination** role, a role that it does not use. Move-through-door could just as easily compute the destination's position from the **door** role and then not access the **door** role again.

baseboard. Tracking the baseboard is a simplified version of detecting it and is shown in figure 24b. One Hough transform is used in the area around the last known location of the baseboard[6] (making use of the **baseboard** representation's ability to store this information) to detect candidate edges. These edges then have the brown/white adjacency check performed along them and if they "pass", then this position is used to update **destination**. Marcus can determine the distance to the baseboard as the baseboard runs along the floor, but cannot determine an azimuth because the baseboard runs across the entire width of the door. The baseboard is effectively at all azimuths. So, to update the azimuth of the **destination** Marcus must dead-reckon from the original value set in the find-door subtask. This is reasonable because there is a limited range to the orientations that Marcus can have after successfully navigating through the door and so any dead-reckoning errors can be compensated for by using the search-for-item task to detect the agent's next destination in the hallway.

Limits placed on the allowable angles for the "horizontal" and "vertical" lines used in door detection limit the number of potential doors considered. However, this means that depending on the agent's orientation to the door frame when the find-and-go-through-door task begins executing, the door may be too skewed in the image to be recognized. This unanticipated fact caused an iteration back to the task decomposition step of the methodology, which resulted in the addition of the rotate-to-view-door subtask. This task rotates the agent to several different orientations under the assumption that if the find-and-go-through-door task is being executed, a door should be nearby and if find-door fails to detect it, it is probably due to the agent's orientation.

---

6. The reader will note that this constitutes a form of foveation.

Detecting the door from the hallway requires a different set of situated perceptual routines. The main difference is that from inside a room, the baseboard can be seen in the hallway through an open door. When looking into a room from the hallway, there is no discernible feature that is always visible. Another issue is that the distance between the agent and the door when the find-and-go-through-door (room side) task is executed is larger than is possible in the (hallway side) find-and-go-through-hall-door task due to the width of the hallway. In the hallway, the door takes up a larger portion of the image and so the distortion of the camera's wide angle lens becomes a factor. Its "fish eye" distortion causes the straight door frame edges to appear curved when they are far enough from the center of the image. In fact, the width of the door means it is not possible to view both left and right door frame edges as vertical lines in the same image from the distances that Marcus will be at in the hallway.

A conceptually simple solution to this problem would be to search the image for appropriate curves. However, this is computationally more complex than simple line finding and the agent is moving down the hall (due to the move-toward-hall-door subtask) while performing this computation. This means that the perceptual routine would have to be fast enough to not allow the image with both curves to pass without being processed. Another solution (that was adopted) is to introduce additional roles into the task. **Left-edge** and **right-edge** roles hold the positions of straight lines detected near the center of the camera image, where they actually appear vertical. When a potential edge, say the left-edge, is detected the next line found is bound to **right-edge**. If the distance between left-edge and right-edge is within tolerance for being a door, the **hall-door** role is bound to a position in-between them. If not, the candidate right-edge becomes the new left-edge and the search

continues. Again Marcus' capabilities determine task roles and representation. At least **left-edge** needs representation because it will not be detectable (since it will be curved) when the right-edge is detected. Recall that the **door** in move-through-door did not need representation and that the task was only concerned with the **destination** role. By the task decomposition, we can see that at this point in the task, only the **destination** role matter, Marcus does not need to track the hall-door and so representation of the left and right door edges need not be maintained.

When the **hall-door** role is bound, the agent needs to be able to determine if the door is closed. Again, the baseboard cannot be used for this purpose since it is not visible inside the rooms, so the fact that some form of "visual clutter" is usually visible is used. When the door is open, the image area between the door frame edges is usually full of edges from the various objects visible inside the room. When the door is closed, its "flat" surface shows few edges (the surface's wood grain is not a problem). If the number of "edge pixels" between left and right door edge locations is above a certain threshold, the door is considered to be open.

Now I can address the perception of the other important class of objects in Marcus' domain, blocks. Many of the entities that are bound to the roles in the arch building task are blocks or constructions of blocks. Blocks in this domain come in 4 different colors. Marcus has capabilities to band filter the input camera image for the range of particular RGB values that make up a color and blob color [9] the resulting binary image. In order to recognize constructions of blocks (stacks), marcus must detect blobs of appropriate color and verify that they appear appropriately adjacent in the image. He does this by testing the boundaries of appropriately colored blobs to see if there are enough adjacent pixels.

One perceptually interesting role is the **base-block** role, which can be filled by either a green or a yellow block. Marcus needs to run perceptual routines for both blocks when trying to initially locate the base-block. Once an entity has been detected (and the role is bound), only the color of the selected entity need be tracked.

Other tasks require more complex perception that often has effects on answers to other questions in the methodology. For example, Marcus' capabilities only allow determine-item-pose to determine the pose of stacks of blocks. Each stack must have 2 blocks, one on top of the other, as well as a red fastener block. Based on the visible amounts of these blocks, the agent can determine the orientation of the stack compared to a canonical expected orientation (see Appendix A). Initially, it was thought that the **item** role could be bound to any block construction, but the pose perception computation only works for stacks.

Another perceptual important class of roles are those that can be bound to *constructions of blocks*. The perceptual routines to recognize arbitrary constructions of blocks can be arbitrarily complex and the agent may well not be able to recognize a given construction from all viewpoints (due to object range or partial component occlusion). This means it is difficult or impossible to recognize arbitrary constructions of blocks at effector control rates. If a construction is bound to a role in a task that controls effectors (a leaf task), the agent cannot perform that task effectively. However, Marcus needs to operate on collections of blocks in the arch building task and so he needs to bind these constructions to roles in some tasks. Furthermore, he needs to be able to recognize when the spatial relationships between blocks that define particular constructions, exist and when they do not. For example, he needs to be able to tell when blocks are stacked and when stacks of block have become "un-stacked". In other words, if Marcus needs to "pick up a block stack", he has to maintain the

position of that block stack for his effector control loop, but he also wants to verify that the stack remains a stack. So, the question is how to balance these needs? How can Marcus' perception system be structured so that it can provide timely role information for action (such as the bound entity's current position) yet maintain all required role information (such as the entity's structure)?

The answer lies in the fact that all the role information is not needed in the effector control loop. Performing an action on any construction of blocks, will generally involve effectors contacting only a portion of the construct's surface. Exactly how the effectors are used to perform the action will depend on the action and type of construction. Marcus deals mainly with stacks of blocks and blocks in this domain snap together vertically. This means that, for example, if Marcus wants to pick up a stack of blocks he should lift the bottom block in the stack because the rest of the blocks will move with it. If Marcus wanted to move a row of blocks, he would have to move each one individually (since they do not interconnect horizontally).

The perception system can be designed using knowledge of the actions that the agent must take. Consider the go-to-destination task. The leaf subtasks of navigate-to-destination and look-at-destination share the **destination** role with the parent task, but they need information about the **destination** at effector control rates. The **destination** could come from a plan step like go-to-stack (which occurs, for example, in figure 19f) and so the **destination** would be bound to a construction. This points us to a division in perceptual responsibility. Complex perception must often be divided into routines that run at effector control rates and routines that cannot. The agent must have routines that can generate the information needed for effector control at effector control rates. In order to meet this requirement the

agent can often not have a single routine that determines all needed information about the entity bound to a role. However, this is not a problem because information that is not needed for effector control can typically be computed much less often. In fact, as stated in section 3.8.2, information not related to effector control should not be computed in the effector control loop because it slows down the PA layer (later, this stance is softened a bit for cases where the agent has sufficient computational power to handle additional perceptual work in the PA layer). So, when executing a leaf task that uses information about individual components of a composite object, the agent can use perceptual routines that determine just the position of the components needed to control that task's effector(s). The agent can then periodically verify that the spatial relationship between components of the construction still exists. The information gathered by both of these perceptual routines should be stored in the task role. For the go-to-stack example, the agent needs to move toward the position of the bottom block because it rests on the floor and Marcus uses the same groundplane constraint that Bruce uses to relate height in the image to distance from the agent. Since the other blocks in the stack are off the ground, their positions will be incorrectly estimated. Marcus can update the position of the bottom block in the effector control loop and verify the "stacked" relationship between blocks before and/or after the go-to action.

All of the other locomotion and manipulation tasks function similarly. Their leaf tasks can have their roles bound to individual blocks, while the roles in the parent tasks can be bound to constructions. This means that Marcus can use his blob coloring routines to make the individual blocks PRs for the leaf tasks and his adjacency detection routines (that use multiple blob colorings) to detect block constructions.

The building tasks (stack-item1-on-item2 and span-item1-and-item2-with-spanner) are

interesting because they are used to create constructions of blocks. Both tasks could bind their **item1** and **item2** roles to any block structures and so their subtasks must bind their roles to the blocks that determine when the stacking or spanning action is complete. For example, the span-item1-and-item2-with-spanner task has subtasks to determine topblock1 and topblock2, i.e. the individual blocks that the spanning block will cross[7]. In principle, the spanner could also be a composite object and the **spanner** role in the leaf tasks would have to be bound to the portion of the spanner that was going to contact the topblocks.

Even after splitting the perception process into those portions that can and must be inside the effector control loop and those that do not have to be, there is still the important issue of view-dependent perception [49]. Simply, this means that different entities look different from different perspectives. This is a general problem in vision and this dissertation does not develop novel vision techniques to solve it. However, it must be addressed in the design of agents. I assume that the agent's capabilities allow it to handle this problem for the domain's PRs. The job of the agent designer is to assist the agent in recognizing composite objects. In general, some parts of a construct may be partially or fully occluded. Also, certain spatial relationships may only be detectable from a certain distance. Marcus needs multiple perceptual routines to recognize different constructions in different situations and a strategy for when to apply them. Knowing which perceptual routine to apply can itself be a complicated question and the agent will need some guess as to the position and orientation

---

7. For other tasks, such as go-to-destination, the parent task determined if a role was bound to a composite object and, if so, what portion of that object should be bound to the leaf task roles for that action. Why then does span-item1-and-item2-with-spanner have separate subtasks to do this? The decision is somewhat arbitrary in that I could have made go-to-destination's decomposition contain separate subtasks to do this. However, because this task must determine the topblock of *two* different constructions, some camera movement is often required (unlike tasks that deal with only one construction). Since this movement itself requires an effector action, it needs to be done by a leaf task (since only they handle the effectors).

of an object to select a routine. The problem in Marcus' domain is that he cannot detect adjacencies in constructs from a distance. This means that if he wants a stack of blocks, he may have to guess that a far off block is actually part of a stack and be prepared to search elsewhere if it is not.

The strategy Marcus uses to select perceptual routines is based on his capabilities. Marcus has separate perceptual routines for individual blocks as well as stacks and spans of blocks (in various color combinations). Which routine he uses depends on the task role. Marcus needs to do three basic recognition tasks, recognize individual blocks, recognize constructions of blocks, and recognize component blocks when they are part of constructions. The first and last task may be different as discussed in section 3.6.2. If a task role is bound to a PR, the first type of routine can be used. If a task role is bound to a construction, Marcus will use a routine of the second type. Should that role be shared with a task that needs to maintain information about some individual component of that structure, a perceptual routine of the third type is required. Marcus may have to use alternate routines of the second and third types depending on his (estimated) perspective on the object. In order to understand how perspective is used to select a routine, we must understand some details of how Marcus detects constructs.

Marcus determines spatial relationships using a technique based on operators called "springy dumbbells". This technique provides a means of detecting color adjacency in the image without the need for a precise description of the boundary between the colored objects. Each dumbbell is two rectangular regions separated by a variable distance. For Marcus' arch building task, it is sufficient for these regions to be separated only in the y direction (in image coordinates). Detecting adjacency involves making some coarse esti-

mate of the boundary's location (usually just a horizontal line) and placing dumbbells along it such that the two regions fall on opposite sides. Next, the two regions are analyzed. If the top region contains enough pixels of the "top" block's color and the bottom region contains enough pixels of the "bottom" block's color then this dumbbell votes for these blocks being on top of each other. If the regions do not contain enough pixels of the appropriate color, different vertical distance between the regions are tried. If enough dumbbells vote for a relationship, Marcus decides it exists. Appendix A provides more details on this technique.

Marcus can use simple blob coloring to identify single blocks and the springy dumbbell technique to identify constructions of blocks. When he needs to recognize a block that is part of a structure, he needs to use the dumbbells to determine if a block is in the structure and then he can monitor the block using blob coloring (note that blob coloring doesn't verify that the block is still in the construction and so periodic verification of that fact is still necessary). Details of how the various perceptual routines are swapped in and out in the appropriate situation will be saved until section 6.7.

Having an understanding of Marcus' perceptual capabilities, we can now address the perspective issue. Marcus can perceive individual blocks at all angles and distances (up to some maximum) since they are primitive-recognizables. For constructions of blocks, the problem is that spatial relationships may not be perceptible from a certain distance. Marcus can handle this issue for stacks of blocks. If he estimates that a stack is too far away for normal (dumbbell) recognition, he should use a different routine to recognize just one block. This routine will yield the position of "candidate stacks", i.e. the objects that might be part of the desired stack. This routine should be different because, in addition to monitoring the position of the detected block, the routine should look for another block above

the first. Marcus must move closer to the detected block and see if another block becomes visible on top of the first (so the stack is detected). Once this happens, the perceptual routines can be changed to the normal stack/block routines. If Marcus gets close to the block without another block being found on top, the agent is in a difficult situation. Possibly, this block is part of the desired stack and it has just come unstacked or maybe this is just a singleton block. There is no general purpose solution for this problem. Marcus could search the local area for either other candidate stacks or maybe for the block he thought should have been on top of this one (indicating the stack had come undone).

In summary, we have now seen that Marcus has many perceptual routines for the various task roles. In section 6.3, many of the roles in Marcus' task decomposition were shown to be shared roles. In fact, many roles were given representation for this reason. In this section, we have seen how the perceptual routines can be structured to determine information about the entity bound to a shared role where the tasks using that role need different information about the entity. In the next section, we will see how the information determined by these different perceptual routines can be propagated between tasks when it effects how a task views the entity associated with its role.

How does the duration of the various role/entity bindings effect the perception system?
Most entities remain bound to roles for the length of the associated task. So, Marcus can have separate locating and tracking functions for most entities. He possesses XLocate and XTrack routines, where X is one of the colors he can recognize (e.g. PurpleTrack). These routines are essentially the same except that XTrack is more restrictive in what objects it will accept as matches because it assumes a reasonably accurate estimate of the position has been determined by XLocate.

For the location and tracking of composite objects, a trade-off exists. The designer must decide on the amount of computation used in tracking and the amount used in locating (or verifying that a composite object still retains the correct structure). Since tracking of individual parts is at effector control rates, the more parts that are tracked, the more computational resources are consumed by the main perception/action loop. However, the periodic verification of the relationships present in the construct is made easier by already having positions for more of the parts. This means the agent needs to expend less effort reacquiring, i.e. running the locate routines of, parts. Given the structures in Marcus' domain and the computational power available, all the parts can be tracked in the action loop. However, one can imagine a more complex structure like those that exist in BridgeWorld [81]. In this domain, structures have complex shapes and large numbers of parts. The agent will continually track only those parts with which it interacts. For example, to pick up an object, only the parts where the agent places its "hands" are tracked. These parts then serve as "seed" points for the reacquisition of the other pieces of the construction.

An issue now arises because despite earlier claims that information not needed for effector control (position of all components or the fact that the components still have the same relationship) should not be maintained at effector control rates, Marcus can afford to do it. Marcus has compound object perception routines of the form XonYLocate and XonYTrack, where X and Y are both colors (e.g. GreenOnYellowTrack). The XonYLocate routines use the springy dumbbells technique described earlier. The XonYTrack functions just use the bounding boxes of blobs of the correct colors to determine if the blobs are "adjacent". This method is less accurate, but faster.

The question is, if all the pieces of a structure can be tracked in the effector control loop,

why not just have a single task role for the entire construct? In other words, why have a role for the whole object and separate roles for its parts? Why not just have the entire object fill roles in all tasks? The answer is that the capabilities of this agent allow it to act on (or with respect to) single blocks, not constructions of blocks. This means that some portion of Marcus' architecture must translate each plan step that effects a structure of blocks into effector commands that manipulate only single blocks. This is why Marcus picks up a stack of blocks by lifting the bottom block. The pick-up-stack task has been changed into a "pick-up-block" task that will fulfill pick-up-stack's goal. In addition, the separation of constructions into components allow the agent to deal with different perspectives when relationships are not discernible (as discussed above). If the structure could only be recognized by a single monolithic routine in the effector control loop, there would have to be different routines for each possible perspective. By having perception routines for individual components, Marcus can derive some information about the environment even when the desired relationships between components cannot be verified.

What level-of-detail (or resolution) is required in the information of the representation? Most roles are used for effector control and so Marcus needs the highest accuracy available (in this case, that comes from vision). However, as discussed above, entities bound to some role can be perceived accurately enough with (coarser) sonar (such as the door in the move-through-door skill, or the obstacle in avoid-obstacles).

6.5. Communication

What information is important in inter-task relationships? We have already seen several important classes of information that are important to the relationship between tasks and communicated between tasks. First, there is the information about the entity bound to a role

that the tasks use (as decided upon in section 6.3). For the represented roles in Marcus' task, that information is the associated entity's position. Tasks exchange this information for a variety of reasons. Sometimes a parent task passes this information to one or more of its child tasks as a way of representing expectations about where the entity should be. The child tasks will then, presumably try to acquire the entity with the agent's sensors. At other times, a parent task will pass positional information about only a portion of the entity bound to its role to a child task. This occurs then the child task does not need all the information stored in the parent task's role (such as data on the entirety of a structure when the child will only operate on one piece). Two sequential tasks may exchange positional information about an entity that one task has bound to a role when the next task shares that role. In emergent behavior schemes [16] parallel tasks often exchange positional information since they often each control different sensors/effectors that all take some action with respect to the same entity.

A second kind of communicated information that has already been mentioned is perceptual information. Typically, one task will pass perceptual information about some entity to another task in order to have that other task bind or maintain a shared role. For example, search-for-base-block tells search-for-item that base-blocks can be yellow or green. Search-for-item then binds the role to a block of one color or the other and reports the results back so that later tasks sharing that role need only track one color. Sometimes, a task with a role filled by a structure will pass perceptual information about only a portion of it to another task.

Another kind of exchanged information that has previously been hinted at is action input and output. As with Bruce, action input refers to the parameters needed by the action. For

example, when executing a go-to-destination task, the agent needs to know how close to get to that object and that will depend on what action comes next. If the agent wants to pick up the object, then it must get within effector range, but if the object was merely a convenient target to servo on, the agent may not need to get very close at all. Although none of Marcus' actions require it, there could be parameters for multiple, parallel actions to be taken on different parts of a compound object (recall the show shovelling example of section 3.7.2). Action output refers to the results of actions. Child tasks report their outcome to the parent task which decides on the "meaning" of this result in the agent's current context. For example, in the find-and-go-through-door task, when find-door does not find a door, it reports this fact to move-through-door. However, move-through-door does not necessarily interpret this as meaning there is no door. Instead, it may have rotate-to-view-door change the agent's perspective because this is sometimes the cause of door finding failure. After enough perspective changes, if the door can still not be found, move-through-door will report to find-and-go-through-door that there is no door.

Another type of information that is communicated between tasks is progress information. Progress is different than the results of actions because it represents how an ongoing action is advancing. Progress can be used by tasks wishing to monitor how another task is proceeding. Tasks can interpret progress information using other information that only they know. For example, a parent task may notice that a child task is not making adequate progress and stop its execution in favor of another task. Alternately, tasks can change their operation based on how another task is doing. Marcus uses progress information during the go-to-destination task to denote when he is in the process of avoiding an obstacle (as opposed to heading straight for the destination). The problem is that when turning to avoid

obstacles, the destination often goes outside the field of view. This means its position has to be dead-reckoned. Marcus' encoders have a high rotational error, i.e. dead-reckoned positions become highly inaccurate when Marcus is rotating, and so the estimate of the destination's position drifts. When the obstacle has been avoided and Marcus faces the position where he believes his destination is, he may not be able to match it to the task role because either the object is not viewable or the actual and estimated positions differ too much. This usually occurs after Marcus has had to turn sharply to avoid a close obstacle. Informing the task that maintains the **destination** that an obstacle is being avoided allows that task to accept larger discrepancies between detected and estimated position when doing the correspondence for the **destination** role.

Marcus also exchanges a form of confidence information between tasks. In this domain, confidence is fairly simple except with respect to composite objects. Confidence that the entity bound to a task role is the (or a) "correct one" is binary. If an object is acceptable to an Identify routine, then the agent binds the role. If a role is bound, then the agent has enough confidence to operate on that entity. If a role is not bound, then the agent does not have enough confidence in any detected entity. However, for compound objects, there is a complication because of the need to separate those objects into their components for the perception/action loop. Marcus may be tracking the individual components of a stack, but those could drift apart over time. The cue to periodically verify a component relationship is based on confidence information. Whenever a relationship between components has been verified, the confidence in that relationship is high. Over time it should decay until it reaches a threshold at which Marcus verifies the relationship again. This should cause one task to communicate the low confidence to a leaf task that can handle the verification by

running a locate routine. Since Marcus has perception routines that verify the stack relationship at effector control rates, confidence in the roles with this relationship should always remain high.

As we have seen, all the information exchanged between tasks is about the entities bound to the task roles. This is exactly what we were expecting since that is why we have roles in the first place. The tasks that share roles are mutually concerned with the same entity in the environment and so it is logically that they communicate information about the entity.

Now we can discuss how roles can be connected when they have an epistemological link. Since certain information in Marcus' domain can not be maintained at effector control rates, there may be multiple data structures holding information about the same entity. For example, the pose information returned by determine-item-pose is not maintained like position information. It is merely stored for use by other tasks until the agent moves from the location where the pose was determined. Sometimes multiple roles in subtasks are linked to a single role in the parent task. For example, the **base** and **topblock1** roles are linked to **item1** in the span-item1-and-item2-with-spanner task. The **base** and **topblock** roles need to store the positions of those blocks while **item1** stores the relationship between them. Removing unnecessary detail, such as this, from the consideration of the perception/action components of the architecture was a major motivation behind the concept of the task role (see sections 3.4.2 and 3.5.2). Although the agent needs to operate on multi-part structures, some tasks in the decomposition do not need to know about the entirety of the structure to be effective [35]. Links between representations are created by communication between the parent task and its children. The child task needs to report its role information to the parent and the parent needs some function to synthesize the data from (possibly multiple) roles

into its representation of the entity. In the case of the representation that stores the pose of an entity, this function is fairly trivial, just store the action result of the subtask's representation. For a representation such as the arch the **arch** that Marcus is building, the position stored in the representation is determined by the positions of its components. There must be a function to abstract the positions of the various pieces into a single arch position. For Marcus, the arch position is defined as half-way between the positions of the two stacks that are spanned. Each of the stacks is also a multi-part entity. If the agent needs to detect some component of the arch, a function of the arch's position and the agent's knowledge of arch structures can be used to make an initial estimate of the component's position that can subsequently be perceptually grounded. The key feature of these functions is that they are defined by the kind of structure that the role represents. There will be functions for arches, different functions for stacks, etc.

6.6. Architecture

How should the agent's tasks be laid out in its architecture? The first step in defining the architecture is to divide the agent's tasks into PA layer and non-PA layer tasks. The leaf tasks of figures 21 - 23 are all PA layer tasks because they bind their roles to PRs and they perform their actions by controlling the agent's effectors. These tasks have a short cycle time because once an effector starts moving, a rapid perception cycle is needed to control it properly. In other words, once the agent starts moving, it is a matter of critical safety that it be responsive enough to stop in time. This implies that the representation of these tasks must be well maintained and because of the PA layer's cycle time, it can do this.

The top most task of figures 21a-c, 21e and 22a-e should not be PA layer tasks partly because their roles can be bound to composite objects that may be too expensive to maintain

at PA layer rates. However, for stacks, Marcus can maintain the position/relationships of all components and so these tasks could be in the PA layer. I do not place them in the PA layer because they do not effect single blocks (which is all that is allowed by the PA layer's capabilities). The span-item1-and-item2-with-spanner task (21d) is not a PA layer task for the same reason (since **item1** and **item2** could be composite objects), but its span-topblock1-and-topblock2-with-spanner subtask should be in the PA layer. While it is not a leaf task, its roles are bound to PRs by the preceding tasks. This task does not control effectors (since it is not a leaf), it does share its roles with its children and they complete the action.

The find-and-go-through-hall-door task, as well as some of its immediate children, are not PA layer tasks. The align-with-hallway and center-in-hallway tasks are not PA layer tasks because the hallway is not a PR and so their representation could not be appropriately maintained. Rotate-base-to-hall-door is interesting. It uses the rotate-base-to-item task, which would normally be a non-PA layer task because **item** could be bound to a compound object. However, rotate-base-to-item is not used at any other time during the arch building task. It would be possible to place a restriction on the entities that could be bound to **item** and place it in the PA layer also. However, I have chosen to make rotate-base-to-item fit the pattern of other similar tasks and be outside the PA layer. Since rotate-base-to-item is a non-PA layer task, its parent task cannot be and so rotate-base-to-hall-door is a non-PA layer task. Find-hall-door is a PA layer task since hall-door is a PR[8] and find-hall-door uses

_____

8. The reader may recall that the hall-door is actually identified by the left and right edges of the door frame, two separate objects. It may seem that the hall-door cannot be a PR, but the left and right edge representations are not used outside of the find-door task. The hall-door representation is bound to a point between the left and right edges and is not updated perceptually based on them (it is just dead-reckoned after it is bound). So, the hall-door is functionally a PR because its representation has no parts to be identified and it can be maintained at effector control rates.

its subtasks to manipulate effectors. Align-base-and-turret is also a PA layer task since it has no roles, but it controls the wheel and turret effectors. Finally, go-through-hall-door could be a PA layer task since hall-door is a PR and it is passed to move-through-door.

In the find-and-go-through-door task (figure 23b) the move-to-door and rotate-to-view-door subtasks could be PA layer tasks since the door is a PR and their immediate children handle the effectors. However, rotate-to-view-door uses rotate-to-destination which is not a PA layer task and so it cannot be either.

In Marcus' implementation, the first level of subtasks of both door tasks (figures 23a and 23b) are non-PA layer tasks even though some of them could be. This is to keep with the ideal of limiting what goes on in the PA layer to those tasks that need high-bandwidth connections between sensors and effectors. The first level subtasks of the door tasks are sequentially executed tasks that use other tasks to accomplish their goals. In some cases, they make decisions based on context that is best not stored in PA layer representation. For example, the find-and-go-through-door task remembers the different viewing angles that have already been tried to recognize the door.

The plan step tasks are not PA layer tasks. They often bind their roles to non-PRs and they have access to information that the PA layer tasks should not, such as maps of the environment. Since the PA layer tasks use local-space coordinates to control effectors, the coordinates of the agent's maps must be converted to be of use.

Now the question is, what if any division can be made among the non-PA layer tasks? Here, the issue is tied to the knowledge available to a particular task. The more knowledge a task brings to bear on a problem, the more processing time it requires. The plan step tasks, for example, have the ability to use the results they receive from their children to determine

the agent's next step. They possess the context to interpret the result codes from the agent's actions and might be able to initiate replanning because they have access to a planning engine[9]. Other non-PA layer tasks, such as, find-and-go-through-hall-door are simple sequencers for their child tasks. They follow a predefined pattern of tasks, but they relieve the PA layer of having to decide what an action results means in the agent's current state and what action comes next.

The non-PA layer tasks should be divided into two different layers, one containing the plan steps and the other containing the rest of the non-PA layer tasks. This division allows one layer to hold data that does not need the PA layer's continual maintenance and the other to hold a planning engine (and possibly even more information). I make this division assuming that the agent's next PA layer task can usually be selected without any inference. In other words, a sequencer that maps action result codes to "next tasks" can be used. The purpose of keeping the planner this far from the action selection process is to lessen the temptation to use its "heavy weight" processing unless it is necessary, e.g. when the sequencer cannot decide on the next action.

6.7. Implementation

The implementation of Marcus' representation system and architecture must follow the design decisions made in the previous sections. There need to be multiple representations for each represented task role that is shared between tasks at different layers. One needs to be usable by the PA layer and another by a non-PA layer because there will be some infor-

---

9. As previously stated, Marcus does not have a planner. Rather, he has a pre-generated plan. Since this thesis is not about the design or construction of planners, this pseudo-planner is acceptable because it has a reasonable planner interface and the agent is designed with the consideration that using the planning component is possibly a long, slow process.

mation about the associated entity that should not be stored (and thus maintained in the PA layer).

The structure of the representation used by tasks at each layer can be very similar because the information that the representations must carry fits into the categories specified in section 6.5. The difference is in the portion of the environment that the layers consider when determining values for their representation components, e.g. some layers consider whole objects and others consider parts. In the remainder of this section I will provide details of the two representations used in Marcus' architecture and how they perform the tasks of building the arch. Then I will discuss how these representations are kept up to date with the state of the world.

Marcus uses two forms of representation, one for his PA layer tasks and another for his non-PA layer tasks. I call the PA layer representations *markers*, as with previous agents. I divide the non-PA layer portion of the architecture into the Task Executor (TE) and the planner. The TE's representations are called *protomarkers* while the planner's representation is not the subject of this thesis.

Table 6.7 summarizes the components of the marker data structure. Marcus' markers are similar to the markers used by Bruce and Spot because similar information is communicated between tasks.

Table 6.7: Components of Marcus' Markers

Component Name	Component Contents
Index	The role name used in combination with the Action component to determine how the agent should act with respect to the object associated with the marker.
Property	The ego-centric, polar (r, θ) coordinates of the associated object.

Table 6.7: Components of Marcus' Markers

Component Name	Component Contents
Identify	Two routines, Locate and Track, that are used to initially bind the marker to an object and to maintain the Property component thereafter. These routines are manipulated by the TE based on the situation. Also contains a "processed" flag to decide if one of this marker's Identify routines needs to be run (see below).
Action	What the agent should do with (or with respect to) the object associated with the marker. Action is a pointer to a function that the TE parameterizes. This component also stores the result code when the action is completed.
Confidence	Consists of an "instantiated" flag indicating that this marker has been associated with an object in the environment.
Progress	Contains a code indicating how the action being taken on the associated object is progressing. Often, this is simply "OK" because the action is going according to plan.
Dependency List	A list of other markers that are needed to either complete the action specified in the Action component or update this marker. For example, Marcus is capable of perceptually maintaining information about whether or not the "stacked" relationship between entities in the PA layer, but his capabilities still make it sensible for the entities to have separate representations. Since determining if two blocks are stacked requires determining the positions of both blocks, a single perceptual routine is used to update the Property component of multiple markers. Markers for blocks in a stack each have perceptual routines that update the positions of all the stack's markers. The marker whose routine runs first will handle the updates to all markers and the Identify routines of other markers will be skipped (on this cycle of the PA loop).

6.7.1 The PA Layer Main Loop

Since Marcus' PA loop performs the same basic functions as the PA loops of the other agents, i.e. maintaining representation and selecting actions, it has a similar structure. First, the PA layer attempts to update the Property components of any instantiated markers by running their Identify routines (the Track routine in this case). Then the PA layer tries to

instantiate any uninstantiated markers using their Locate routines. Finally, an action is selected and performed. The remainder of this section discusses the difference between the steps executed by Marcus and those executed by Bruce.

6.7.1.1 PA Loop Step 1: Update Markers

This portion of the loop proceeds in a similar manner to marker maintenance in Bruce except that some markers have Track routines that can detect correspondents for multiple markers. This happens when the PA layer verifies the relationship between some entities and so has also determined their positions (and might as well update them). When one marker has such an Identify routine, it has the other markers to update in its dependency list. These markers set their "processed" flags and so their Identify routines will not be run on this execution of the loop.

There is another difference with Bruce the creates a problem when trying to update or instantiate markers. Bruce had a single means of image segmentation (the groundline) and these segments, not the whole image, were analyzed by his Identify routines. Since all routines considered the same image regions, those regions could be ranked by how well they matched a given marker. This made it reasonable to use the stable marriage algorithm, which insured that two markers were never matched to the same image region (and thus presumably the same object). Each of Marcus' perceptual routines handles its own segmentation, it is difficult to guarantee that multiple Identify routines are not detecting appropriate objects in overlapping image regions. Solving this problem would involve keeping track of image regions already associated with markers. However, the means of describing those regions so as to include only the area that is really part of the object is computationally expensive with Marcus' blob coloring routines (the main segmentation method) because only

the bounding box is retained for each blob. It was empirically determined that Marcus need not address this problem since important entities rarely look like each other (because their colors are easily distinguishable) and overlap in the image (so previous position can resolve which marker should be matched to a region).

6.7.1.2 PA Loop Step 2: Instantiate Markers

Marker instantiation is also similar to Bruce. If a correspondent for a marker is found, its position is stored in the Property field and the "instantiated" flag is set. Multiple markers can also be instantiated by finding a correspondent. As with the Track routine, a marker's Locate routine can instantiate other markers in the marker's dependency list. Locate routines that instantiate other markers in the dependency list are trying to detect a group of components, based on their spatial relationship, and associate each component with a marker. Therefore, one correspondent for each dependent marker must be found or the relationship cannot be verified. However, as discussed in section 6.4, it may not be possible to detect the relationships necessary to find multiple correspondents from all perspectives and so insisting that a Locate routine instantiate all or none of its dependent markers may be too restrictive. Marcus deals with this problem by not placing an "all or nothing" Locate routine in markers if he believes he is not in a good position for the required relationship to be visible. For example, when he is too distant from a stack that he wishes to go to, he will put a "single block" Locate routine on the bottom block's marker and a XonInstYLocate on the top block's marker (X and Y are block colors, e.g. green or yellow, and XonInstYLocate detects a block of color X on top of a block of color Y that is associated with an instantiated marker). The bottom block's marker can be instantiated from any perspective (since its a PR) and when the top block's marker instantiates it signals the TE (see the discussion on

monitor functions in the TE below). The TE will then change the Locate and Track functions of both markers to be XonYLocate and XonYTrack. These routines will update/instantiate both components or neither, but the agent should now be at a perspective where this is possible.

6.7.1.3 PA Loop Step 3: Select/Execute Action

Marcus' PA loop now selects and action for the agent to perform in a manner similar to the one used by Bruce. Again, when multiple markers have values in their Action components, the order in which the markers were added to the PA layer determines the order in which the actions are performed.

6.7.2 PA Layer Behaviors

The actions that can be specified in the Action components of the markers are made up of a collection of parallel tasks called PA processes (actual execution is pseudo-parallel). These processes map readily to the leaf tasks of the decomposition. The implementation of these processes (except those described in Appendix A) is not very important as their effects have been amply described in previous sections. So, the remainder of this section just lists the cases where the PA processes for a task do not correspond to the exact layout of leaf tasks in figures 21 - 23.

The go-to-destination task (figure 21a) combines the navigate-to-destination and avoid-obstacles tasks into one PA process so that only one process needs to control the agent's wheels. This process looks for obstacles and if any are detected (or the **obstacle** role is bound) it avoids them, otherwise it moves toward the destination. A common pattern is to combine move-to-good-X-view, use-Y-effector and watch-for-Z tasks into a single PA

process. X, Y and Z depend on the task. For example, for stack-item1-on-item2 (figure 21e), X is item2, Y is stack and Z is item1-on-item2. I do this because the move-to-good-X-view and watch-for-Z tasks are generally concerned with the same object(s), i.e. X is often part of Z as above. One tasks is looking for the conditions indicating that the human has performed some action and the other is trying to position the agent so that it can observe when these conditions occur. Unlike the flow indicated by the flow diagrams in Appendix B, these operations go on continually so that the agent can always adjust its position to a good view if something in the environment changes to ruin the old view. For example, when Marcus is looking to see when item1 gets put on top of item2, if item2 moves, Marcus needs to adjust his view so that he can still observe item2 properly. This combination of subtasks occurs in the stack-item1-on-item2 (21e), span-topblock1-and-topblock2-with-spanner (21d), pick-up-item (22a), put-down-item (22b), and put-item-in-toolbelt (22c) tasks. A similar combination occurs with the move-to-good-item-view and calculate-item-pose subtasks of determine-item-pose (22e).

Two other leaf task combinations occur in the find-and-go-through-hall-door task (23a). The subtasks of align-with-hallway are combined into a single PA process that senses if the baseboard is horizontal and if not sends a rotation command to the base. If it is, the base is stopped and the PA process completes. The watch-for-hall-door and open-hall-door-if-necessary subtasks of find-hall-door are also combined into one PA process. This process checks if the **hall-door** role is bound and if it is, looks to see if the door is open (asking for human assistance if it is not). If the role is not bound, the process continues monitoring the wall trying to bind the role.

A final question that the reader may have is why there is a track-baseboard behavior in

move-to-door? **Baseboard** is a role and roles are automatically maintained as part of the main PA layer loop (section 6.7.1), i.e. they do not need separate maintenance behaviors. This was done because tracking the baseboard did not fit into the usual interface for the Track routines. Track routines normally produce (r, θ) coordinates for candidate correspondents and the marker is matched to one of these. As mentioned in section 6.4, it is difficult to estimate an azimuth for the baseboard, but this does not matter to task execution. So track-baseboard just updates the **baseboard**'s r and its θ is dead-reckoned as part of normal representation maintenance.

6.7.3 The Task Executor

Tasks in the TE layer are different than PA layer tasks because they do not have the same restrictions on cycle time. They can use different representation structures with component names similar to PA representations. The TE uses structures called protomarkers that are summarized in table 6.8.

Table 6.8: Components of Marcus' Protomarkers

Component Name	Component Contents
Index	The Index component consists of a unique identifier for each protomarker. Plan steps (defined below) can refer to a particular objects by the Index of their associated protomarkers. When a plan step causes a protomarker to be created (see below) the protomarker is given a unique Index that further plan steps can use to refer to the associated object.
Property	The Property component is divided into two fields called Where and Prop. The Where component contains the position of the associated object. This is not in ego-centric coordinates, but relative to some other object in the environment. Marcus uses assembly site relative coordinates, i.e. coordinates relative to the position of the first object placed at the arch assembly site. The Prop component stores information that have been determined by the PA layer, but is not maintained in the PA layer's main loop. Marcus stores the pose of objects here.

Table 6.8: Components of Marcus' Protomarkers

Component Name	Component Contents
Identify	The Identify component of a protomarker is in charge of handling the Identify routines of its dependent markers (see Dependency List). It contains information about the relationship(s) between components of the protomarker's associated object (if there are more than one) and how to manipulate dependent marker Identify routines for different perspectives or situations.
Confidence	A protomarker's Confidence component contains a bit that is used to decide when to reverify spatial relationships between its object's components. After the protomarker's associated object goes outside the field of view, the bit is set so that the next time the agent wishes to act on the object, the relationship(s) specified in the Identify component must first be verified. This is how the agent "periodically" verifies the associated object's structure. Newly created protomarkers have zero confidence since their markers have not be instantiated.
Dependency List	This component holds a list of the markers that are associated with this protomarker. Each marker represents one of the parts of the object represented by the protomarker. There can be a single marker in a protomarker's Dependency List. As we'll see below, all plan steps refer to protomarkers and so tasks that effect simple objects will create protomarkers with one marker.

The reader will note that there is no Action component as there is in the representations of other agents. Marcus does not need a protomarker component that is processed by the TE's main loop to decide what action to take on the associated object. That information is held in the current plan step. It would be simple enough to have the planning layer of the architecture place this information in protomarkers as they are passed to the TE, but that is not how Marcus is implemented.

The layout of tasks in the TE is as discussed in section 6.6 and does not need further discussion, except for one task. In the span-item1-and-item2-with-spanner task, the find-base and find-topblock tasks are PA processes, as are the leaf tasks of span-topblock1-and-topblock2-with-spanner. Span-item1-and-item2-with-spanner and span-topblock1-and-

topblock2-with-spanner are in the TE. What is interesting here is that the span-item1-and-item2 task sequences a number of other tasks, some of which are PA processes and one of which is a TE layer task in and of itself. In this architecture, TE tasks can be nested, i.e. TE tasks can execute other TE tasks, but PA processes are not permitted to activate other PA processes. A PA process can perform some action (like binding a shared role) that causes another PA process to also take action, but only the TE can start a PA process executing (and thus monitoring for the cues produced by the other PA process). This protects the responsiveness of the PA layer, by not allowing any process to execute in that layer without the explicit request of the TE.

6.7.4 The TE Main Loop

The task executors main loop consists of the following steps; check monitors, handle completed actions and determine next action. I will explain each of these steps separately.

6.7.4.1 Check Monitors

Monitors are small pieces of code that are examine the TE's protomarkers to determine if specific environmental conditions have occurred. Monitors are used to implement several of the behaviors discussed in the design of Marcus. Each monitor specifies a "firing condition" and what to do when that condition occurs. The firing condition is based on some state of a protomarker or its associated markers. On each pass through its main loop, the TE will check to see if any of these conditions exist. If so, the monitor function is executed and then removed from the current monitors list.

In principle, monitors could be used to do any monitoring task, such as checking the Progress component of a protomarker's markers to see how an action is proceeding or wait-

ing for component objects to be in certain positions. Marcus uses monitor functions to do three jobs, each of which has a monitor condition based on the instantiation of markers. First, monitors provide the mechanism for handling the switching of marker Identify routines when the agent's perspective changes. Recall that when the agent is too far away from a stack that it wants to associate with a protomarker, the TE creates a marker for the bottom block with single block Identify routines. Then a second marker for the top block is created with the XonInstYLocate routine as its Locate routine. A monitor is set up to watch for the instantiation of this second marker. This will happen when the bottom block's marker is instantiated and the top block's marker's Locate routine has found a suitable block above the bottom one. The monitor function switches the Identify routines of both markers in the monitor function's protomarker to the appropriate Locate and Track functions for this new perspective. For the stacks in Marcus' domain these functions are XonYLocate and XonYTrack. These functions depend on finding correspondents for both markers, as opposed to the previous function that allowed one marker (for the bottom block) to be updated independent of the other.

The second use of monitor functions is in the span-item1-and-item2-with-spanner task. At the start of this task, the agent must find the topblock of each construct since these are the ones where the spanning action can be observed. The agent looks at the bottom block and sets up a monitor function that keeps count of the number of times the agent has run the **topblock** marker's Locate function (XonInstYLocate) and failed to find it. The monitor function executes if either the marker is instantiated or the frame count exceeds a certain threshold (10). If the marker was instantiated, then the monitor function merely exits and is removed from the monitor list. However, if the frame count was exceeded, the TE deter-

mines that the stack is no longer in place. The monitor function places a Stack action on the bottom block's marker and an uninstantiated **topblock** marker in its dependency list. This should cause the stack to be re-created. The TE can then continue the spanning action.

The last use of monitor functions is in the instantiation of **base-block** markers. In general, a role can be played by many objects and while it is possible to have a single Locate function that can detect if any of them are present, there are times when the designer already has a collection of separate functions that will do the job. This is the case with Marcus. A **base-block** marker can be associated with either a yellow or a green block. YellowLocate and GreenLocate functions had to be developed anyway, so the easiest way to instantiate a marker on either a yellow or a green block at a particular location is to put two different uninstantiated markers in the PA layer. A **base-block** protomarker is created in the TE and when it is uninstantiated, it has two markers in its dependency list. One has a GreenLocate function and the other has a YellowLocate function. Both have the same coordinates in their Property components. When the agent tries to instantiate the base-block protomarker, both markers are placed in the PA layer and a monitor that will fire when one of them is instantiated is placed in the monitor list. The monitor function will remove the other (uninstantiated) marker from the PA layer and subsequent references to the protomarker will effect the single remaining marker. The protomarker can also store the color of the detected base in the Prop field of its Property component, thought its marker already has the correct Identify routines and so this information would only be for use by the TE or planner.

6.7.4.2 Handle Completed Actions

Once the monitor functions have been checked and run, the TE checks to see if the actions on any of the markers associated with its protomarkers have completed. If so, the PA

layer will have passed a reference to a marker to the TE and the TE must search its proto-markers to find the one that contains that marker. Next Marcus checks if the action that just completed is the last of the plan step[10] and if the plan step has a completion function. A completion function is similar to a monitor function except that it fires when its associated plan step is finished. When this function runs, it alters the state of the protomarker(s) referenced by the completed plan step. Marcus uses completion functions to set the position portion of the protomarker's Property component, usually after a put-down-item task. In one case, the origin of the assembly-site coordinate system is set (the first stack built at the site is set to be at (0, 0)). As other blocks are added, their assembly-site relative positions are calculated based on the position of the original stack and the ego-centric coordinates of the marker(s) for the new blocks.

After running any completion functions, the TE makes further modification to its representations based on the action that was completed and the return code placed in the marker's Action slot. The TE must also check if it's in the middle of a sequence of actions that are part of a single plan step. This is because the completion of, for example, a rotate-to-item task has different meanings if it was executed as part of the rotate-to-stack task or find-and-go-through-door task. Table A2 in appendix A provides a list of the TE operations when the various possible PA layer actions complete.

Next, Marcus deletes markers, if necessary. A marker with a completed Delete action has been removed from the PA layer, but the actual memory cannot be freed until this point

---

10. The code that actually ran on Marcus had a bug in that it forgot to check if the action that just completed in the PA layer was actually the last action of a particular step in the plan. Plan steps that required multiple PA layer actions and had completion functions would have those functions run when the first action completed. However, this bug did not effect Marcus because none of the plan steps that would have triggered the bug had completion functions. The description above is what Marcus *should* do.

when Marcus removes all references to this marker by other markers and protomarkers. The last part of the this step of the TE main loop is to delete references to any markers in the dependency list of the marker with the completed action. This is done because those markers were there to support the completed action and are no longer needed (if they are needed for another action, they will be added again). The problem is that sometimes markers are in the dependency list because there is a perceptual dependency, i.e. a relationship between the markers is being verified in the PA layer. These references should not be removed from the dependency list. Marcus knows which marker Index values indicate perceptually dependent markers (e.g. **topblock** or **fastener**) and so all others are removed.

6.7.4.3 Determine/Execute Action

Once any completed actions have been dealt with, the TE can have the PA layer execute new actions. Each step in the plan (see below) references the Index of a protomarker, or the indices of a set of protomarkers, whose associated entities are involved in the step. If the TE does not have a protomarker with a given Index, it will create a new one for the needed object. Then markers (with appropriate Action components) are created for any new protomarkers or for any needed protomarker that doesn't have the necessary markers in its dependency list. For example, a plan step to execute a pick-up-stack task on a stack at a certain global coordinate (x, y) may create a new **stack** protomarker. Then two markers would be created for this protomarker. Their Property components would be initialized using the global-to-local conversion of appendix A. The marker for the bottom block would be an **item** marker and it would have the pick-up-item action on it. The other marker would be a **topblock** marker and both markers would have XonYLocate and XonYTrack Identify routines (assuming that the agent is at the right perspective to observe the stack which it

should be if its about to pick it up).

It is possible for a protomarker to have no markers in its dependency list after it is created in the following way. Markers can be removed from the PA layer when their actions complete by the delete-marker plan step. This is a good idea because any marker (even with a null Action component) consumes resources when it exists in the PA layer since the PA layer will be maintaining the marker's Property component. Efficient plans (such as the arch building plan) remove markers from the PA layer when their associated objects will not be acted on by the agent in the near future. The protomarkers that contain those markers usually are not deleted from the TE, especially if the agent will need to deal with the associated entity again. Protomarkers consume much less computational power because their data is maintained only when the TE is about to act on their associated object. Allowing the protomarker to remain when the markers have been deleted allows the agent to remember certain environmental data at a low cost. The markers can be recreated again when the object associated with the protomarker is acted on again. The important issue here is that when recreating a marker, it may be difficult (or at least highly error prone) to estimate the associated object's ego-centric position as correctly as if that object's marker had remained in the PA layer and been maintained. So, a trade-off exists between the difficulty of re-establishing an object's position and the computational resources that the designer/planner is willing to allocate to maintaining that position.

Once the markers have been created and appropriate values have been placed in their Action components, they can be passed to the PA layer. If the new markers are instantiated by the PA layer, the TE knows that any spatial relationship specified by the protomarker exists because the Locate routines check this. Later, if the protomarker's markers are already in-

stantiated, but the protomarker's Confidence bit is set, the TE will verify the protomarker's

relationship by uninstantiating the markers and seeing if they instantiate again. Once the

specified relationship has been verified successfully, the TE can initiate a new action by

changing the Action component of the pre-existing markers. Table A3 in appendix A sum-

marizes how the TE begins new plan steps (recall that subsequent PA layer actions that are

part of a plan step are initiated by the TE as described in table A2 of Appendix A). If the

relationship was not found between the markers of the protomarker, the TE must execute

an action to rebuild the structure, such as stack-item1-on-item2 or span-item1-and-item2-

with-spanner.

6.7.5 Plan Step Structure

The structure of the plan steps in Marcus' plan are shown in table 6.9.

Table 6.9: Plan Step Components

Component Name	Component Contents
Action	The task the TE should execute. Note that this is not an action in the PA layer, but a TE task, i.e. one of the root nodes of the decomposition diagrams of figures 21 - 23.
Protomarker Tags	Specifies the Index fields of the protomarkers involved in this task. Up to three can be specified in this implementation (the maximum needed by any of Marcus' tasks). A protomarker with the specified Index is created if one does not already exist.
ID	The Identify functions for the entities to be associated with the step's protomarkers. For example, XonY_ID indicates a stack with a block of color X on a block of color Y and Red_ID means this protomarker will be associated with a single red block. This field is only used when new protomarkers are created.
Position	The position of the entity to associate with the protomarker specified in map coordinates.

Table 6.9: Plan Step Components

Component Name	Component Contents
Parameters	Parameters for the task. Used for data such as the stop radius on go-to-destination actions, i.e. how close the agent should be to consider the action complete.
Completion Function	The function to run when the plan step completes.

6.7.6 Summary

Marcus' architecture is divided into three layers, the planner, the task executor and the perception/action layer. The planner and PA layer each interface with the TE. The planner communicates its plans as a linear series of plan step structures to the TE and the TE causes the PA layer to execute the plan by sending it various markers. Markers (actually references to markers) return from the PA layer when actions are complete. The TE interprets the results and sends more markers to the PA layer or manipulates components of existing markers. If the TE discovers that the planner's plan has gone awry, it should send protomarkers to the planner to give it information about the environment from which to replan[11].

The plan steps all refer to protomarkers and it is up to the TE to translate from its representation to the representation of the PA layer to have the plan steps carried out. The protomarkers group sets of markers together and the plan step can simply refer to the whole collection of components at once. The TE is responsible for "distributing" the planner's desired action(s) among the components of an object. This allows the planner to take a more abstract view of the environment and reduces the planning search space.

At the same time, the TE reduces the PA layer's burden by keeping the active markers to

---

11. Marcus' planner does not handle replanning, but the BridgeWorld agent [81] which has a similar planner/TE interface does.

a minimum. In other words, the PA layer expends computational resources to maintain each marker in its memory even if it is not doing any action on that marker's object. The assumption is that if a marker is present in the PA layer, it is important to the current task (or will soon be important) and is being maintained because the accuracy of its Property field is crucial. However, this allows the designer to overload the PA layer with too many markers, causing none of them to be maintained effectively. The TE handles the notion of the "current task" by passing the PA layer just the markers needed to express the task to be done. The PA layer can create additional markers on it own (**obstacle** markers for example), but these will be automatically deleted by the PA layer when the action that caused them to be created is completed. Markers created by the TE will be passed back (again, just a reference is passed), but the TE is responsible for deleting them. This allows the TE to make tradeoffs between the computational load of the PA layer and the accuracy of the information stored in the markers (and hence available to the agent). The protomarkers are used to store information extracted from the environment by or the agent's sensor/effector actions, but that should not be maintained in the PA layer. Spatial relationships can be difficult to maintain at effector control rates and so they are kept in the TE and periodically verified by changing the instantiation status and/or Identify routines of associated markers.

Chapter 7   *Evaluation*

In order to evaluate the usefulness of a representation system, we must first define what makes a representation system effective. The methodology in this thesis is based on the idea of task-oriented design, where control loops that closely couple sensors and effectors are necessary to achieve a task's goals. The computational demands of these loops lead to designs based on task roles because task roles point the designer to the entities in the environment that are important to the task (and thus the agent should be built to spend its time and computation on them). An agent, then, can use representation to store information about the entities associated with task roles, if doing so makes it more efficient and/or effective at its task than it would be without representation. I specify the criterion by which the influence of a system of representation on an agent's task performance can be judged with the following three requirements (some of which are similar to the "desired properties" used in [44]).

7.1. Requirements

R1. The architectural components using the representation must be able to find necessary information using a search of manageable combinatorics (where what is manageable is dependent on the component's "cycle time", as defined in section 3.8.2).

R2. The representation should aid the designer in creating tasks that manage the agent's limited computational resources. Representation systems can be designed for different trade-offs between the accuracy of the information they contain and the cost of maintaining that information.

R3. The representation should aid the designer in creating tasks that manage the agent's limited perceptual resources. Representation should assist tasks in handling such perceptual problems as a limited "field of view"[1] and occlusion.

I argue that any system of representation should meet all of these requirements and not meeting any is grounds to reject that system. R3 is important because this thesis addresses the design of representation systems for *physical* robots in dynamic domains. Such robots typically receive sensor input from a first-person perspective in domains with occlusion. A system of representation should augment the perceivable world by helping the agent remember important, but not currently perceivable information. (recall the "limited field of view" argument for representation from section 3.5.2).

Of course R3 can only apply to the representation systems of layers that are directly connected to sensors, i.e. PA layers. Other layers have no notion of occlusion or a limited field of view. R3 must still be observed, however, because the PA layer grounds the representation used by higher layers.

R1 and R2 are similar in that they place constraints on the computation done by the PA layer. This is of critical importance for agents with multi-tiered architectures because the PA layer handles the agent's interaction with the world. The agent's ability to react, i.e. its ability to adjust its actions to changes in the environment, depends on the PA layer's ability to devote all its computational power to perception and action. Therefore, the PA layer should be efficient in its use of resources (R2) and not be asked to perform computation not useful to the current task (R1).

R1 means there should be no searches of unpredictable length through the representation.

---

1. I use the term "field of view" even for non-visual sensors, though perhaps "perceptual field" would be more accurate.

No extended inferencing should be allowed, i.e. the data stored in the representation must be in a form that is directly usable by the tasks that access it. This is why specific information needed about entities associated with task roles is considered in the methodology (section 3.5). Of course, reducing the search space for one task may mean increasing the search space of another, and in such cases the representation should be more direct for the task requiring the faster perception/action cycle.

R2 says the representation should be designed with an understanding of the trade-offs between the precision of the information stored in a representation, i.e. how closely it corresponds to the current state of the world, and the amount of computation required to maintain that information. These trade-offs will be based on a combination of the importance of the information to the ongoing task and the time required to compute that information. Representations used by the PA layer are generally used in sensor/effector control and so require frequent verification. The representations used by other layers can usually be verified less often because their data is less crucial for moment-to-moment action (that's why the data is not stored in the PA layer) and maintenance at the same rate as the PA layer's representation would represent a significant computational burden to the agent.

A system of representation must meet R1, R2 and R3. A system that met R1 and R2, but not R3 would be myopic in that it would loose track of important entities that went outside the field of view or became occluded. A system meeting R1 and R3, but not R2 would either have to represent only primitive recognizables (see section 3.6) or would bog down under the computational load of maintaining a complex representation. Finally, a system meeting R2 and R3, but not R1 might be designed to operate properly, but allows the designer to imbed time-consuming search processes in the PA layer, hampering its efficiency.

I present two additional "requirements" that are based on my belief that representation forms an appropriate medium for communication both within [10][65] and between layers [71] of the agent's architecture (see section 3.7.2). These are not strictly requirements of the representation system (since any communication protocol can be used), but do provide additional benefits if they are met. If the designer develops a representation system that can be used for communication in an agent architecture, the designer need not create a separate communication system.

R4. The representation system should facilitate communication *within* layers of the architecture. In other words, since all tasks must know how to interpret the representation, no communication protocol is necessary other than exchanging representation.

R5. The representation system should facilitate communication *between* layers of the architecture. The methodology's task/subtask decomposition provides a natural linkage between roles in parent and child tasks. These tasks are sometimes placed in different layers of the architecture and so the designer should exploit this link for inter-layer communication.

R4 says that the communication between all tasks at a layer of the agent's architecture should be facilitated by representation. Note that this requirement restricts the processes from communicating their "internal" state to each other. Internal state is any data that a process does not place in the representation (because it does not fit in any of the representation's components). A process has the context to interpret its own internal state, but other processes will need to be told how to interpret such information. By restricting what information is communicated, the designer is forced to carefully consider what that information is used for and whether it really needs to be shared.

R5 says that representation should assist in the communication between tasks at different layers. For example, one layer may hold information that is not needed for the agent's cur-

rent effector control. This information can be stored in the representation at one layer and passed to the lower layer when it becomes crucial. Alternatively, the changes in representation of an entity at one layer can be interpreted by another layer to effect its representation of the same entity (e.g. the way Marcus determines the positions of composite objects, based on individual block positions, see section 6.5). In addition, higher layers can communicate their beliefs about the environment to lower layers that can ground those beliefs in perception. Communication by placing and removing representation of some entity in another layer's memory is a simple way to do this.

The systems of representation developed by the methodology in chapter 3 meet requirements R1 through R3 and the representation of the agents developed in chapters 4 - 6 also meet R4 and R5. R1 is met by representing only the data needed by the tasks (see section 3.5). Other information about the environment is ignored. By representing only this data, the search space is reduced. The crucial thing about this representation is that it contains information of the form "my car is parked at (x, y)" as opposed to a map representation in which the agent must search for the car to determine the (x, y) position of its parking place.

The search is further reduced by the designer limiting the number of representation elements that exist at any one time (see sections 3.4.2 and 3.5.2 for discussions on having few roles and on deleting representations when their tasks are over). This is particularly important in the PA layer for the maintenance process to be suitably fast. By deciding what roles in the tasks need representation, the designer seeks to limit the number of elements active at any one time (for example, Marcus never uses more than 6 markers at a time). The Dependency component of markers and proto-markers further serves to lessen the search for information by pointing to elements that contain information needed by the task indicated

by the marker's Action component.

The methodology helps designers meet R2 by using the task role hierarchy that results from the task decomposition (see section 3.7). The agent will need to compute some number of environmental properties during the execution of its tasks. For some tasks, the computational load required to continuously verify these properties will reduce the agent's ability to react below acceptable levels. The natural epistemological link between roles in different tasks provides a means of dividing up the storage of properties that the agent needs to know about an entity. Those that are needed for effector control can be stored in the representation used by the leaf tasks, while other properties can be stored in the representations of "higher" tasks. By only updating such properties periodically, computation of all needed information becomes feasible. For example, Marcus updates the positions of individual blocks in the PA layer since they are needed for effector control, but only verifies the proto-marker's relationships between blocks when he is going to effect the associated object.

Sometimes, the link between roles in different tasks represents a constraint on the information stored in each role. For example, the position of a stack determines the position of its component blocks and the position of these blocks determines the position of the stack. In such cases, properties stored in non-PA layer representations are simple functions of properties stored in their component representations [57][81]. This may further simplify processing since additional perceptual computations are not necessary to determine such properties.

In addition, the perception portion of the methodology (section 3.6) addresses R2 by directly addressing the amount of perceptual computation that the agent needs to do to maintain its representation. By considering the binding duration and level-of-detail, the designer

looks at the ability of the agent to use "context" in maintaining the representation. By context, I mean using previous properties of the entities bound to the representation to help compute the current properties.

The designer can use domain knowledge and task constraints to simplify the update process, i.e. the tasks should exhibit "situated activity" (section 1.3.2). Spot's marker's Confidence "timer values" (section 5.7.1) and the maintenance of representation in Marcus' door detection routines (section 6.4) are examples.

The methodology also asks the designer to consider the level-of-detail required in a representation to assist the designer in creating an appropriate maintenance scheme. By considering when less than the maximum amount of precision is feasible, the designer can develop an efficient allocation of computational and perceptual resources. The designer can use precise (and presumable expensive) sensors only where needed and develop control strategies to direct the agent's sensors toward the portion of the environment where the current, most important, information can be determined.

The methodology encourages the "effective field of view" paradigm [13][14] to help designers meet R3 (see section 3.5.2). By designing agents that remember important entities that are outside the perceptual field, the designer allows the agent to not only act using that information, but to store context to assist in reacquiring the entity without the expense of initial identification.

The PA layers of the agents described in chapters 4 - 6 meet R4 by exchanging information between PA processes via the markers (see sections 4.7, 5.7 and 6.7). This means of communication is similar to a blackboard system [47] in that the PA processes are knowledge sources who compute information (about entities) based on information posted by

other knowledge sources. The PA processes post markers and thereby make their important results, i.e. the ones concerning the task roles, available to other PA processes. The agents meet R5 by having higher layers command lower layers simply by placing appropriate expectations in their representation, i.e. uninstantiated markers (see sections 4.7, 5.7 and 6.7). For example, Marcus places markers in the PA layer and the Action components of those markers triggers sensor/effector actions.

7.2. Evaluation of Other Design Methodologies

In this section, I analyze how current design methodologies can fail to produce representation systems meeting the requirements specified above. In doing so, I will consider how these methodologies would be used to design an agent to execute the task that Marcus executes. Since the methodologies discussed here are not strictly representation design methodologies, there is a certain amount of interpretation in deciding how they effect representation. This can be good and bad in that the designer is free to use different design techniques, but at the same time is more free to shoot himself in the foot.

7.2.1 The Subsumption Methodology

The subsumption methodology [17] needs to only be discussed briefly because it does not provide for any form of representation. It cannot design agents to perform Marcus' task without serious contriving of the environment because all the blocks needed are not perceivable at once. This means a subsumption system could not even turn to acquire the base-block in step 2 of Marcus' plan because once the first (purple) block is reached, there is no perceptual cue to tell Marcus to turn toward the next block. This is a violation of R3.

7.2.2 The 3T Architecture

The next methodology I examine is part of the 3T architecture [10]. 3T's design methodology concentrates on decomposing a task and then deciding which of the resulting subtasks belong in which layers of its architecture. This architecture has three layers, a planner on top, a PA layer (called the "skill layer") at the bottom and a sequencing layer (based on the RAP system [28]) in between. The design methodology does not address the issue of representation and this can cause problems.

The problems stem from the representation used in the sub-systems of 3T's architecture. The RAP system uses LISP predicates as its fundamental representation element, while the skill layer uses arbitrary data structures (both within the skills and as the inputs and outputs of communication channels between skills). Note that we are only interested in the bottom two tiers of 3T, as planning is beyond the scope of this thesis.

In order to see where problems can occur, we first need to create a "3T decomposition" for a portion of Marcus' arch building task. Consider the goto-destination task that is executed once Marcus has put down the initial base block at the assembly site and is heading back to a previously seen (purple) block that will be used to create the first stack. The goto-destination task (with **destination** bound to the purple block), is decomposed into navigate-to-destination, look-at-destination and avoid-obstacles. These tasks should all be skills because they need to function in tight perception/action loops to control their effectors as the agent moves through the world. One more skill, detect-destination, must also be added to be the process that detects the destination in the sensory stream. The goto-destination task itself should be a RAP because the skills that achieve it can be activated by the RAP systems and parameterized with the destination location.

While the reader may question the use of the same task decomposition for a different architecture, the tasks used are based on Marcus' capabilities and not 3T's. Avoid-obstacles or detect-destination could be wrapped into navigate-to-destination, but this will not affect 3T's ability to meet the requirements stated above.

Representation comes into play because we know that the skills cannot be written in a purely reactive fashion (see section 7.2.1). Since 3T's methodology does not address representation, the designer could create representations that are the same as those outlined in section 3.7, or that contained more or less data. If the representation elements developed by the 3T designer turn out to have the same structure as the representation developed in section 3.7, then the methodology of this thesis is superior because it aids the designer in creating such structures instead of hoping that they occur by chance. If the representation elements have less data than those developed in this thesis, then the reader is referred to section 3.7 for arguments as to why the representation will be ineffective.

Suppose the skills used a representation that contained more data than the markers used by Marcus' PA layer (section 6.7). The skills might have some map of the environment so that they could figure out how to get from the assembly site to the purple block (and thus the RAP system would only need to send them the name of the current destination). The skills might also represent more properties of the world than the markers, such as the spatial configurations of various blocks, e.g. on top. The former violates R1 and the later R1 and R2 as discussed below.

R1 is violated because if the skill layer uses maps then it does not reduce the search space considered by the skill layer's action selection machinery. That is, R1 is violated because the skills must look through the map (whatever form the map takes) in order to determine

the position of the purple block. If the navigation skills just had information about the purple block, the task of determining that information would fall to the RAP system (or the planner). This is in line with R1 because the skills have tighter PA cycles than the RAPs.

R1 would be violated if the representation stored properties of objects (such as configurations of blocks) not needed by the skill layer's tasks. R1 is violated because, as stated above, the skills will have to search through unnecessary data about objects and properties. In fact, any data stored in the skill's representation that need not be (meaning the skills should compute that information from current sensor values or get it from the RAP systems) violates R1.

R2 could also be violated because representation of information must be updated for it to continue to be useful. Since 3T's methodology provides no guidance with respect to representation maintenance, there would be a temptation to update everything at the effector control rate, i.e. the maximum rate required by any skill. This may cause the precision/computation trade-off to be poorly made. That is, the representation could violate R2 if the designer makes a poor trade-off by permitting the go-to-destination skills to get bogged down verifying information that is not germane to the task.

So, if the skill's representation contains more data than the representation developed for this thesis, that representation violates R1 and R2. This means that skill should use the representations designed by this thesis and this methodology could be used along with the 3T methodology to design an agent's skill (PA) layer.

3T uses multiple representation systems because the RAP layer's representation is different than the skill layer's. Actually, the RAP layer's representation is more strongly specified than the skill layer's. The fundamental element is the LISP clause, e.g. (on block-5

block-6) or (purple block-5). This type of representation violates R1 because its lack of structure means it is not deictic [1]. Unification is used to determine if facts are stored in RAP memory. Since there is no one place where the information about a particular entity is stored, this look-up takes an unknown amount of time.

In addition, R1 can be violated because of the difficulty with removing facts from the RAP memory. RAP memory contains time-stamped predicates that increase the RAP system's search time until they are removed by some form of garbage collection (which itself might be computationally expensive and require unknown time). By using my methodology, the designer finds roles in the agent's tasks. The data contained in the representation of these roles is known to be important to the role's task and so when that task is over, it is easy to find and remove data that is no longer necessary.

The methodology of this thesis avoids these problems by advocating the creation of a deictic representation of important, task dependent, objects for all layers of an agent architecture. Since the 3T methodology places either no restrictions (skill layer) or the wrong restrictions (RAP layer) on representation, designs using my methodology can produce more efficient and effective systems of representation.

7.2.3 The PURE Architecture

The PURE architecture [45][65] is a subsumption-like architecture, that unlike traditional subsumption architectures [17] has a system of representation based on structures which the authors also call "markers". The main contribution of the work to the design of representation is the architecture's use of representation to coordinate/communicate between behaviors. However, PURE's design methodology does not address the important issues of how to decide what to represent, how to structure that representation and how to devise a

scheme for effective representation maintenance.

Suppose Marcus was designed using ideas from the PURE architecture. Markers would then be able to represent multiple points in the environment and specify a relationship between them (such as the trajectory markers used in PURE's unblock task [65]). This would make them similar to protomarkers except that all information would be kept in the PA layer. There might, then, be an arch marker that contained the positions of the blocks in the two stacks and the spanning block. If the agent tried to maintain the information in the arch marker at effector control rates, that would violate R2 because no mechanism was given to make trade-offs between accuracy and computation. However, the PURE system does provide a means of controlling marker updates via the gaze behavior. Whenever a marker's position is close to the fovea of the PURE robot's stereo system, the agent tries to update that marker. By controlling the gaze appropriately, the designer can implement a maintenance scheme that prioritizes some information over others.

Putting this mechanism in the gaze behavior hides one of the crucial aspects of the data contained in the markers, i.e. the usefulness [14] of each piece of information. Different tasks require different precision in the information they use. The hierarchical representation created by my methodology makes clear the priority that is placed on the precision of a piece of information. This determines the rate at which a piece of information needs to be maintained. By placing different information in different layers of the representation hierarchy, the designer can tailor a maintenance scheme to the needs of the layer using the representation. The gaze behavior, on the other hand, completely hides the importance of information because it encodes it in terms of where the agent should be looking. This makes it harder to change some information's priority level or change how a priority level is main-

tained. By providing a hierarchical representation system that can represent different data at different levels and by providing a Confidence component, my representation makes it clear what data is important and the priority with which it should be updated.

Another problem with the PURE architecture arises from the fact that the architecture has a single layer. It has a default behavior that is active when no other behavior is and control is transferred from behavior to behavior based on environmental conditions. This is similar to Pengi [2] and, as discussed in chapter 2, suffers from the same scaling problem. The more complex a task an agent must complete, the more aspects of the environment must be registered to determine when to switch behaviors. This bogs down the PA layer and eventually violates R1 because the agent may have to search through representation that is not used by the current task, but rather by some future task. My representation provides an easy means of saving off information that the agent wants to store, but that it cannot afford to maintain, i.e. verify, as part of its current task. The protomarkers, kept at another layer of the architecture, allow Marcus' PA layer to remove elements of representation that are not needed for the current task, yet retain information that can be used to create them again when they are next required. Although such designs can cause an agent to miss serendipitous updates (e.g. when Marcus sees a structure that he has a protomarker for, but since its markers are not in the PA layer at the time, no update is performed), the methodology makes it easier for designers to create representation systems for agents with complex tasks (and/or in complex environments) because the designer can decide how much "looking for data that may be useful in the future" to do.

7.2.4 The Soar Architecture

The Soar architecture [46][62] has been extended to interact with the world. In particular,

Robo-Soar [46] is used to control autonomous robots. The design methodology advocated by the Robo-Soar is to cast the agent's task and environment in a form usable by Soar's production system. All layers of a Soar architecture use the same representation, which is divided into short and long term memory.

Both short and long term memory violate R1. Short term memory is a collection of (attribute, value) pairs and so is not deictic. Such memory structures have the same drawbacks (see section 7.2.2) as described for the RAP system's memory [27]. Long term memory contains productions used in Soar's planning and learning processes. Since planning and learning are beyond the scope of this work, long term memory is not the type of memory I am concerned with here. The productions stored in long term memory store data about the possible results of actions to assist in determining possible future world states. This means there is extra data to sort through in order to determine the current world state (violating R1).

R2 is violated because Soar does not address the maintenance of representation. Soar is designed to be a general symbolic AI architecture that uses a database of facts (assumed to be correct) about the environment. The (attribute, value) pairs in short-term memory are assumed to be delivered there from an interface to the sensors, but there is no notion of feedback to that unit. That is, the sensors must have some method of generating the (attribute, value) pairs that Soar's engine needs because Soar does not tell the interface what to look for, i.e. what is important to the current task. There is no guidance on how to distribute the computational and perception resources to maintain the information efficiently. Section 3.6 of my methodology addresses such concerns in detail.

7.3. Evaluation of Design

Another way to evaluate the methodology is to look at its effect on the design process. The methodology is meant to provide a context in which the agent designers can work. By guiding designers with its questions, the methodology puts the designer in the right frame of mind to create efficient and effective representation systems to be used in the agent's perception and action component. Chapters 4, 5, and 6 provide evidence of the kinds of workable designs that emerge from designs carried out in this context.

Let us examine how the methodology directs the designer to consider the design issues of what to represent, how to structure that representation and how to maintain that representation. Recall that these were question I set out to answer in section 1.4. First, the methodology instructs the designer to consider only the objects that are important to each of the agent's tasks. This is done by the task-role concept, where important objects that the agent needs to act on (or with respect to) are used as the basis for thinking about the systems that will achieve the agent's goals. By having the designer consider the importance of each role in each task, the methodology encourages agent designs in which the agent considers only a small subset of its environment when executing any given task. This encourages the design of situated behaviors and limits the scope of the entities that might be represented by the agent. The final decision about representation will be based on roles and trade-offs in representing each role that are discussed in section 3.5 (e.g. computational expense of verification vs. penalty for selecting an action based on stale information).

By asking questions about the specific data that is needed by tasks (section 3.5) and about the data that is communicated between tasks (section 3.7), the methodology points the designer toward a structure for the role representation. Types of information that are

typically important for tasks handling perception and action are discussed in section 3.7.2. This is important because by directly addressing the information contained in roles, the designer is kept from representing arbitrary amounts of data. This is important not only to the efficiency of the finished agent, but it saves time in the design process by simplifying the design of the maintenance scheme.

The methodology addresses the design of the agent's representation maintenance scheme in section 3.6. Developing a strategy to assure that the stored information is up-to-date can be difficult and so considering the trade-offs presented by the methodology is important. Since maintenance is usually computationally intense, the methodology also suggests several optimizations to consider in designing the maintenance process (section 3.6). Considering the optimizations can greatly help the designer discover computational savings that are often necessary for the agent to operate fast enough to perceive important changes in the environment.

As the design process discussed in chapters 4, 5 and 6 show, the methodology provides a setting in which the designer can craft solutions to the three basic design issues (section 1.4). For example, the Bruce designer can use the task-role concept to decide what should be represented. In other words, Bruce should not represent every landmark on his search route in the move-to-landmark task because this task only requires information about one landmark at a time. This leads to Bruce's inter-level control structure where information about the layout of the entire environment is stored in one layer of the architecture and portions of that information are passed down to the PA layer only as needed.

The methodology also helps in creating the structure of Marcus' representation by having the designer consider information communicated between tasks, particularly the epis-

temological links between roles. Having the designer consider the relationship between the abstractions of the block structures used by different tasks helps determine what information needs to be computed by each task in order to keep the representations consistent with the state of the world. Considering these links also helps develop maintenance and communication systems where tasks maintain their representation and communicate that information to other tasks to help maintain the representations in those other tasks.

Finally, the methodology assists in the design of schemes to maintain representation, keeping it consistent with a dynamic world. For example, Spot's perception system is designed with the understanding of Spot's capabilities (stereo vision, limited field of view, etc.). Since the designer knows that Spot must keep track of multiple objects that will not all be within the field of view, the designer knows that Spot's focus of attention will have to be periodically shifted to different objects. By considering Spot's task, the "confidence component" of his representation was created as a trigger for when to shift the camera's view.

7.4. Summary

In this section I have evaluated my design methodology in two ways. The first is by comparison with other agent design methodologies. I have evaluated three other design methodologies with respect to the task performed by Marcus. Each has failed to meet the requirements for representation systems set forth in section 7.1. These analyses extend to cover many other current architectures/design methodologies because they all fail to address the design of representation systems.

The 3T architecture uses either arbitrary data structures or LISP predicates as represen-

tation. Any other architecture that uses one or the other, or both, will be at risk for similar problems. This includes most other layered architectures, such as ATLANTIS [29], RCS [3] and TCA [69].

The PURE architecture uses a structured representation called "markers", but does not limit the number of markers in the system nor the information that they can contain. The lack of an information hierarchy leaves open the potential to bog down the system with unnecessary details. Similar systems, such as Pengi [2], Sonja [19] or Brill's agent [14] would have similar problems if used for complex tasks.

The Soar architecture has two memory systems, neither of which meet the representation criterion. The short term memory consists of (attribute, value) pairs that have the same problems as the LISP predicate memory of 3T. The long term memory contains productions that are used to predict the outcomes of actions and not represent the agent's environment.

The second way the methodology was evaluated was by analyzing its effect on the design process. Following the methodology should put the designer in the right context to create effective and efficient systems of representation. I analyzed how the questions of the methodology provide this guidance by discussing the trade-offs involved in determining answers to these questions.

Chapter 8   *Conclusion*

This thesis presents a methodology for designing autonomous mobile robots. The main contribution of this methodology is in the design of representation that can be used efficiently and effectively by the agent architecture. The robots for which this methodology is appropriate are expected to operate in dynamic, uncertain, physical domains. In order to be efficient and effective in such domains, the robot's control must have a close coupling of sensors and effectors and this constrains the design of representation.

The efficacy and validity of the methodology is documented through several agent designs that were successfully carried through to operating implementations. The key feature of these agents is the action oriented portion of their architectures and their system(s) of representation. This representation allows components of the architecture to be more efficient and effective at meeting their goals.

8.1. Action Selection Redux

The debate over the use of representation in the action oriented portion of the architecture (particularly the perception/action layer) has shifted away from an argument over whether or not perception/action layers should be stateless, reactive systems, to a debate over the form of effective representation. Work by researchers such as Agre and Chapman [2], Riekki and Kuniyoshi [65], and Brill [14], as well as this work, demonstrate that agent's can be successful with simple PA layer representation systems. A common feature of all these

works is that the representation is used in moment-to-moment effector control. That is, given current sensor values and a particular state of the agent's representation, a simple transform determines the behavior to engage in now. This work moves beyond previous work by examining the issues that effect the use of representation in action selection; namely what to represent, how to maintain representation and how to organize representation so that it can be used effectively throughout an agent architecture. I address each of these issues in turn in the next three sections.

8.2. Fundamental Representation Design

The most fundamental question in the design of a representation system is, what should be represented? In order to have a coherent design process for perception/action systems, one must have a task-oriented approach. A major implication of a task-oriented approach is the concept of task roles, i.e. the aspects of the environment that influence the agent's behavior during a particular task. These roles focus the methodology in developing representations that are efficient and effective for the agent's tasks.

Intertwined with the issue of the roles in a task are the questions: what entity (or entities) in the environment can fulfill those roles? and which (of the many possible) properties of those entities should be stored in the representation? While an obvious means of deciding what to represent is to base it on the data needed by a particular task, a less obvious basis is the agent's (computational) ability to verify the data stored in the representation as is required in a dynamic environment. I call this verification process, "maintenance". The computational complexity of general maintenance was the reason that representation-less agents were proposed in the first place: representation just took too much time to update in dynamic domains. My methodology calls for the designer to consider, in a structured way,

the task requirements and possible trade-offs associated with verifying representation.

The notion of what entity fulfills a role depends on the abstraction with which a task views the environment. The methodology encourages the designer to create role hierarchies (where the roles in the hierarchy represent different abstractions of objects in the world) because in decomposing the agent's task, the designer will often not just decompose the goals, but also the aspects of the environment that need to be manipulated to accomplish those goals.

The designer must also consider what data about the environment needs to be stored in a role's representation. The role hierarchies (and the layered nature of the agent architecture) allow properties with different levels of "usefulness" to be represented. Some data is used by tasks with sensor/effector control loops and so is used to determine moment-to-moment actions. Other data helps determine which of multiple tasks should be active and so is typically accessed much less often than the sensor/effector control data. The more often a piece of information is used, the higher the maintenance cost the agent should be willing to pay. Role hierarchies allow designs in which different information about entities is represented at different layers of the architecture. The methodology causes the designer to consider how quickly each task requires the agent to respond to changes in the world and role hierarchies allow the designer to create a maintenance scheme, for each layer of the hierarchy, that conforms to these requirements.

The major benefit to role hierarchies is that they allow information to be spread across layers of the architecture. Typically, the more information that is represented at a particular layer, the longer the tasks at that layer will take sorting through it. Even if the information is (relatively) inexpensive to maintain, if it is not pertinent to the layer's current task, it

slows down the layer's cycle time because tasks waste time examining the information to determine if it is relevant. A role hierarchy allows information to be stored outside of a layer, so that it will not reduce that layer's responsiveness. However, since the agent is storing that information somewhere, it can be passed back to a layer if that layer's tasks need it.

8.3. Representation Maintenance

Representation maintenance is fundamental to representation use and so the methodology examines the process of maintenance and how it can be made efficient. Having a representation in the action oriented portion of the architecture means maintaining it. Data that is merely stored will become useless in a dynamic environment. Given the necessity of maintenance, the key to efficient use of representation is limiting what must be maintained. The methodology advocates reducing the time required to maintain task roles by limiting the amount of information represented by the agent. This is done by limiting the *number* and *size* of the data structures used for representation.

The methodology encourages the number of structures to be limited with the role concept. The designer seeks to determine a "minimal" set of aspects of the environment that determine the agent's action(s) during a task. While a truly minimal set of roles is not required, the fewer entities that are represented, the faster the maintenance process can be and thus the faster the PA layer's cycle time.

The methodology also has the designer consider what information is stored in each representation, in order to limit the size of each structure. Position is most likely necessary, but other data can be in the Property component as well. The amount of information in a representation data structure presents the same trade-off as the number of data structures. The

more information, the more data the agent can use in selecting its actions, but the more expensive it is to maintain. Of course, having more data to use in selecting an action can also be a burden, as it means that process takes longer. In general, the designer will want to represent as few properties of the entities bound to the roles as possible. Those decisions will be based on usefulness (the importance of the information to the action selection process) and maintenance cost (including the required accuracy and the cost of shifting perceptual/computational resources to determine the information when it is needed).

The methodology further encourages efficiency in the maintenance process by having the designer consider how situated-ness (see section 1.3.2) can be used to computationally simplify the identification of the entities associated with roles. For example, Marcus can detect doors using vertical edges (see section 6.4) because the door detection routines are expected to be used in Olsson Hall.

Once the designer has selected roles and entities that can fulfill them for a task, the methodology suggests several more methods of optimizing the process of representation maintenance. First, the principle of situated activity can be applied by differentiating between initially binding an entity to a representation and maintaining that binding (see section 3.6). The example robot designs all use different methods to track objects associated with roles and to initially locate those objects. This allows the tracking routines to use context established by the locating routines. For example, in section 5.4, Spot makes use of the fact that his Locate routine places a proximity space on a human and so tracking can be done by the proximity spaces without them knowing what they are tracking (they just follow texture).

Another optimization is found by considering the level-of-detail needed for a particular representation. When it is possible for a task to operate with less than the maximum possi-

ble precision in its information, the designer should consider allocating perceptual and computational resources elsewhere. In fact, the limited perceptual fields of most agent's means that some tasks may not be able to perceive all important entities at all times with all sensors. The methodology encourages the designer to choose the entities to monitor based on the agent's sensor capabilities, the precision of entity information required by the tasks and the likelihood that the information stored about an entity will become stale while it is not being monitored.

The representation hierarchies developed using the design methodology allow the designer further control over representation maintenance. These hierarchies present the possibility that properties stored at one level of the hierarchy can be computed as an aggregation of the properties of representations at a lower level. The position stored in Marcus' arch representation is the aggregation of the positions of the blocks that make up the arch. This technique is useful when similar properties are used by different tasks, but those properties are at different levels of abstraction (such as Marcus' ego-centric vs. assembly-site relative coordinates from section 6.3). When such aggregation is possible, separate perceptual computation to determine the properties used by non-PA layer tasks are not necessary and automatic maintenance of non-PA layer representation is possible by propagating the results of PA layer maintenance to the non-PA layers [81]. Dis-aggregation [57], which refers to using domain knowledge to decompose an abstract entity into its components, allows one task to express domain knowledge to another task, in the form of default values. For example, since Marcus is designed with the knowledge that a stack is made up of blocks on top of each other, as well as, how big the blocks are, he can estimate the positions of the component blocks given the position of a stack [81].

Another benefit of role hierarchies to representation maintenance is that layers of an architecture are typically designed to update all representation at the same rate. This is because representation must be maintained as part of a task's main loop and it is presumed that all representation is pertinent to the current task (or else it would have been removed from that layer's memory). If all representation is maintained in a task's main loop, then it must all be maintained such that the representation that requires the most frequent updates can be maintained properly. The single maintenance rate at each level means that information about entities that is computationally expensive to determine impedes the responsiveness of the entire layer. The methodology encourages role hierarchies as a means of storing some information at a layer where it can be maintained at an appropriate frequency.

8.4. Representation Organization

The methodology assists the designer in structuring the agent's representation so that tasks have access to information they need and do not have access to information not needed. While this last part may seem obvious or trivial, the value of some information is rarely cut and dry, but rather involves several trade-offs, discussed below.

The designer is encouraged to examine role sharing and to build semantic links between the representation at different layers. This design allows the agent's tasks to communicate through an efficient channel. This includes both inter- and intra-layer communication. Consider intra-layer communication. If all tasks in one layer process representations of the same form, the representation forms an efficient communication medium since all tasks understand its structure and it contains only the information that is needed by the tasks (see section 3.7).

Semantic links between role representation in different layers allows for inter-layer communication. This communication takes the form of hierarchical control of lower layers by higher layers. Communication occurs when one layer sets the context for another by passing selected information down the task hierarchy to control what the agent treats as important and how it acts. By having the designer consider the semantic links between roles, the methodology aides the designer in discovering how the information in the representation of one layer can be converted to the information needed by tasks at a lower layer. This information can be placed by the higher layer, in the lower layer's representation, thereby controlling the actions of the lower layer.

8.5. Surprises

Having used the methodology to design several robots, there are some aspects of the design process that still surprise me and these are important to keep in mind. One surprising aspect is that general engineering concerns often appear to make it worthwhile to separate perception and action. It often seems like good engineering practice to divide up the robot design problem into separate components and then connect them together later. Particularly, it is tempting to create a perception system that is dissociated from the needs of the agent's actions. In other words, one is often tempted to create a perception system that generates information about the environment that is placed in some database that a separate action system can read and then decide what to do. The rationale for this separation is that the perception system does not have to know about the action system's current activity. The separation simplifies the communication between the two, but it means that the perception system must try and generate all information that the agent needs for all of its tasks, all the time. The difficulty is that since the perception system does not know what information is

needed now, it must generate all possible information so that the action system will have whatever it needs available. This means the perception system will be either inefficient or bogged down depending on how much information it is trying to generate.

My methodology tries to steer the designer away from this by first examining the task roles, objects that fulfill them, required information about those objects and representation for those roles before addressing perception. The intention is that when the designer has this information available with a task-based organization, the designer will make a perception system that is strictly task-oriented. By generating only the required information about the objects associated with the roles in the current task, the perception system can be made sufficiently efficient to be effective in rapidly changing environments.

Another surprising fact was that representation for the perception/action component of an agent's architecture is fairly independent of that architecture. In other words, the design of representation is mostly unaffected by the choice of an agent's architecture. I say "mostly" because, while nothing in the methodology explicitly refers to the form of the agent architecture, the representations I designed over the course of this thesis were designed assuming a layered architecture. The layered architecture assumption leads rather immediately to hierarchical systems of representation, as discussed in this thesis, with more complex object properties (that are more expensive to maintain) stored at higher layers and simpler to compute properties (possibly computed using only the agent's primitive capabilities) stored at the lower layers. However, the layered-architecture assumption is not required. Instead the methodology urges the designer to segregate the properties to be represented according to the difficulty of maintenance and include only the simple to compute properties in the perception/action component of the agent. Thus, the sensor and ef-

fector control process will not bog down with representation maintenance processing.

A further point is that there are many layered architectures in the literature and this representation design methodology does not depend on the definition of layer used by any one. Late in the design process, the designer must choose which "computational unit" (perhaps the architecture calls it a layer, perhaps not) will execute which tasks and this will depend on the unit's ability to maintain the representation used by the task. However, the representation has already been designed based on the needs of the task, not the architecture.

8.6. Contributions

This thesis contributes to the field of autonomous-robot software design by describing a task-oriented design methodology for agents operating in dynamic environments that require tight sensor/effector control loops. The specific contributions of this thesis are:

1. This thesis presents a design methodology for representation systems for the action oriented portion of an agent architecture. Specifically, the design methodology

   • presents the trade-offs between precision (benefit) and maintenance (cost).

   • assists the designer in considering hierarchies of representation in order to determine an appropriate allocation of computational and perceptual resources.

   • encourages the use of representation for communication between behaviors by role sharing.

   • encourages the use of representation for communication and control between layers of the agent architecture.

   • is quasi-architecture independent in that it can be used with any of today's multi-tiered architectures (e.g. [10]) or for any agent with a perception/action layer.

2. This thesis presents three physically implemented agents performing different tasks as validation that agents designed by this methodology can be implemented and operate effectively in a dynamic environment.

This thesis has presented a design methodology for creating systems of representation, for the perception/action component of an agent's architecture, that are efficient and effective for use in dynamic environments. The validity and efficacy of this methodology have been shown by its use in the design of multiple physically implemented agents. While I argue in chapter 7 that the methodology is superior to other methodologies in the area of representation design, the methodology should actually be seen as complementary to other work in autonomous robot architectures, e.g. [10] and robot development languages/environments [46][52]. Designers, even working with the constraints of current architectures, will only be successful if they incorporate efficient and effective representation and I contend that adhering to the methodology presented in this thesis will establish a context in which the creation of such representation is possible.

Appendix A   *Robot Implementations*

A.1. Bruce Implementation

This section provides details of Bruce's software that have been described more abstractly in chapter 4. In particular, I address the navigation/obstacle avoidance system, the color histogram matching of the perception system and the Task Executor's role in interpreting the search plan and doing the local-global coordinate system conversions.

A.1.1 Navigation System

The GoTo action, which Bruce places in the Action component of **landmark** markers as part of the search-for-opponent task, is used to navigate Bruce through the game environment. This action consists of 4 PA processes called move-toward-landmark, detect-obstacles, look-at-landmark and find-path that execute in (pseudo) parallel. These processes implement the move-toward-landmark and detect-obstacles tasks shown in the decomposition diagram of section 4.1. These PA processes are not shown as leaf tasks in the decomposition because, at design time, it was not anticipated that the navigation system would take this form.

In the actual implementation move-toward-landmark has two roles, **landmark** and **intermediate-destination**. Detect-obstacles has an **obstacle** role and look-at-landmark shares the **landmark** role with move-toward-landmark. Find-path has **landmark**, **intermediate-**

destination and **obstacle** roles. The first two are shared with move-toward-landmark and the last with detect-obstacles. All roles are represented by markers to facilitate inter-process communication.

The operation of these four PA processes is as follows. Move-toward-landmark controls the agent's wheels to servo toward the position stored in the **landmark** marker or the **intermediate-destination** marker, if it is instantiated. If this processes is servoing to the **intermediate-destination** marker, it also decreases the marker's confidence value. When the value drops below a certain threshold, the marker is deleted.

The detect-obstacles process checks the "objects" found in the groundline by the marker maintenance routine (section 4.4). If any of these objects is not the goal, is closer to Bruce than the goal and is within a path of one Bruce-width centered on the goal's azimuth, it is declared an obstacle (see [14] for a discussion of projecting the agent's path into an image to find obstacles). An **obstacle** marker is created with its position set to the closest point of the obstacle object to Bruce.

The look-at-landmark PA process turns the agent's camera to look at the **landmark**'s position, as long as this is possible. Since Bruce's pan/tilt camera mount can pan to view only an approximately 180 degree field in front of the agent, sometimes it is not possible to view the goal, particularly when avoid obstacles.

Finally, the find-path PA process does nothing until an **obstacle** marker is created by detect-obstacles. This **obstacle** marker causes the process to create an **intermediate-destination** marker along a path that is free of obstacles (see below). The move-toward-landmark task should move toward this destination for some time and then start going toward the landmark again. At this point, if the same object (or a new one) is an obstacle, detect-ob-

stacles will flag it again.

Any **obstacle** marker that is created by detect-obstacles is analyzed by find-path to determine if it needs a move persistent representation. As mentioned in section 4.3, obstacles are not represented unless they are near the edge of the field of view (and therefore likely to be unviewable soon). If an obstacle is close to the bottom edge of the image, its **obstacle** marker is preserved, otherwise it is deleted.

Find-path chooses a path for the agent to follow that is toward the goal, but around any obstacles. These paths are represented by creating an **intermediate-destination** marker that will become the agent's new destination (based on move-toward-landmark's behavior). The path, and hence the intermediate-destination's position, is chosen by performing a 3 image scan of the nearby area. That is, when an **obstacle** marker is created, find-path computes the groundline in three images, taken at three separate pan angles representing the 180° sweep "in front" of the agent. Using these groundlines, the agent can estimate the distance to objects and hence the free space, at every point. The agent chooses an azimuth for the **intermediate-destination** marker based on the goal's azimuth, the azimuth's of any represented obstacles (those that have not been deleted) and the amount of free space currently available along an azimuth. The chosen direction is the one that is closest to the goal, does not run through any obstacles, and has at least a certain amount of free space. Since the intermediate-destination is not the agent's real goal, it is not necessary that the agent reach that location, merely head in that direction. Therefore the radius stored in the marker can be any suitable large value. Move-toward-landmark will delete this marker after some time and so the agent will start heading toward the goal landmark again.

A.1.2 Perception System

Bruce's ability to distinguish landmarks and the opponent is based on color histogram intersection [72]. Histogram models of each landmark and the opponent were captured and stored in the Locate and Track functions of the various Identify components that the TE could place in a marker. These models were matched against histograms of the area between pairs of discontinuities in the groundline. This area was bounded on the left and right by the discontinuities and on top by the lowest (closest to the image bottom) point on the groundline between the left and right edges. The top was the top of the image.

Each pixel was hashed to a 9 bit value, made up of the 3 highest order bits of the red, green and blue image bands. The "degree of match" between a histogram h and a model m was measured by the formula of Swain and Ballard [72] as $\sum\limits_{i \in 2^9 buckets} min(h[i], m[i])$

I also use the iterative scaling technique of Terzopoulos and Rabie [73], shown below:

```
while ((d < .9) && (lastd < d))
      lastd = d
      for all i, m[i] *= d      // scale model
      d = sum_i(min(h[i], m[i]))// intersect
```

The idea here is that the scale at which the model was taken and the size of the current region being histogrammed are frequently not the same. Therefore, based on the results from the last histogram intersection, we scale the model down (the model histogram is created with the object close to the camera and so the model rarely gets scaled up) until either the match value becomes very high or the results start to decrease. This final match value for each region is one of the parameters used to rank objects and markers in the marker maintenance scheme of section 4.4.

It is worth noting that even though the segmentation and identification techniques that

Bruce used were sufficient for this task in this environment, they were fairly problematic. I believe there are several causes for this. First, the buckets in the histogram hashing function were uniformly distributed. However, the camera was always down on the laboratory floor where it is considerably darker than at the height where people stand or sit. This meant that colors tended toward the dark end of the spectrum and the highest order bit of any particular color band was seldom "on" unless the overall color was a fairly saturated white. This lead to fewer buckets being important to the histogram and more objects with seemingly different colors being only moderately distinguishable. In addition, the camera's automatic gain control was probably damping down the bright colors some, but it was necessary because otherwise the floor was too dark for Bruce to distinguish much of anything.

Bad segmentation, i.e. discontinuities in the groundline not corresponding to the "real" boundaries of objects, led to histograms of "incorrect" regions being matched against the model histogram (which was segmented from the background by a human). This further depressed intersection match scores.

A.1.3 Task Executor

For Bruce, the PA layer of the architecture can be adapted to perform other tasks by merely adding other PA processes and Identify routines. The TE, however, is not so flexible. Since Bruce has no route planner, the search route is encoded in the TE. It is encoded as a series of local coordinates to visit. I use the term "local coordinates" because the route plan is expressed as a series of steps that make assumptions about the agent's state when the previous step completes. For example, one step might encoded "given that you've just com-

pleted step 4, the garbage can is at (radius = 1520mm, $\theta = \pi/2$)".

The TE assumes that Bruce will be at a particular point when a certain stage of the plan completes. This point varies based on the step of the plan, but the local-space coordinate for the next point is with respect to this point. The TE will adjust the azimuth of the next landmark based on the azimuth stored in the marker of the current landmark, i.e. new landmark θ = old landmark θ + map θ value.

The Task Executors of other agents, for example Marcus, face more difficult problems with local to global coordinate conversion because their maps are truly encoded in a global coordinate system. However, those systems are more robust to error than Bruce.

A.2. Marcus Implementation

In this section, I present the details of Marcus' agent architecture. I discuss how the Task Executor converts between the global coordinates of its maps and the ego-centric coordinates used by the PA layer. I also discuss Marcus' pose detection system and the TE operations when beginning or ending an action in the PA layer.

A.2.1 Local-to-Global Coordinate Conversion (and vice-versa)

Marcus converts map coordinates into local-space (ego-centric) coordinates throughout the execution of the arch building task. The process takes two known local-space coordinates and locates the agent's position in the map of figure 18 in chapter 6. Then using that position, the local-space position of the next object of interest can be determined. Figure 25 shows an example.

next
destination

(dx, dy)

+y

θ_3

r_3

θ_2

(i, j)

(rx, ry)

θ_1

r_2

previous
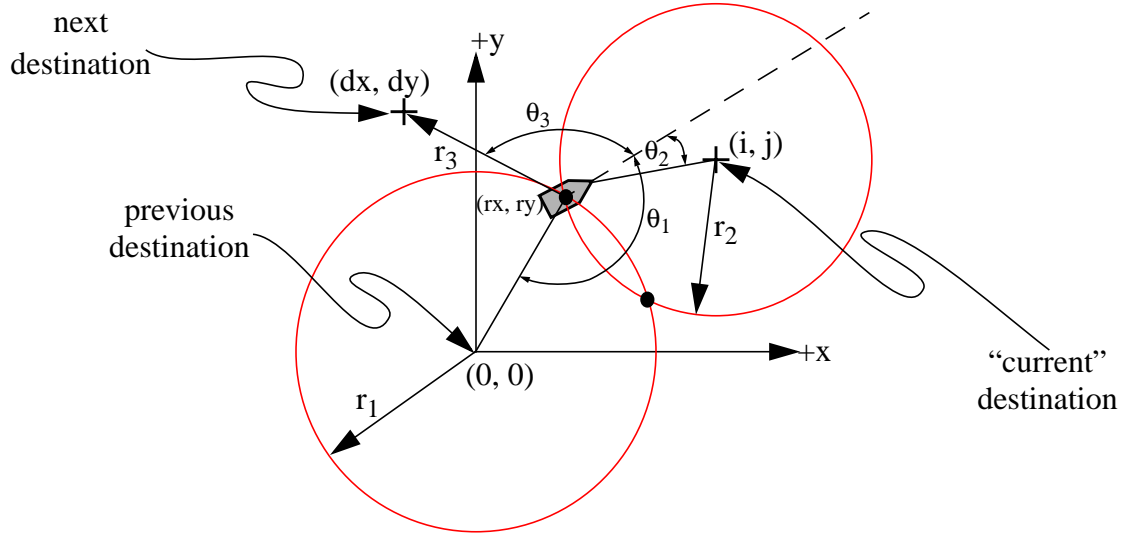destination

+x

"current"
destination

(0, 0)

r_1

Figure 25. Local-to-global Coordinate Conversion based on Markers

The grey pentagram is the agent and the dashed line is its heading. The agent maintains 2 markers, one corresponding to the location of the last place visited (the previous destination) and one corresponding to the landmark used as the current destination. The radii of these two markers (r_1 and r_2) put Marcus at one of two locations. Those locations are indicated by the block dots at the intersection points of the two circles derived from the radii. In global coordinates, Marcus defines the previous destination to be at (0, 0) and the current destination to be at (i, j). By solving some equations, he can determine the coordinates of those two intersection points in a coordinate system centered on the previous destination. It is a simple matter to translate that coordinate system to the map's origin. If the two circles do not intersect, then the intersection equation will fail. I allow r_1 to be made slightly larger or smaller in an attempt to get an intersection. R_2 could also be perturbed, but in general r_1 has been dead-reckoned while r_2 comes from visual data and so is likely to be more correct than r_1.

The question now is, which one of the two intersection points is Marcus' true location?

This point is called (rx, ry). Marcus uses some domain assumptions to resolve this issue. In particular, he uses the azimuths of the two markers and the coordinate (i, j). The azimuths of the markers are shown as θ_1 and θ_2 in figure 25. As with Bruce, Marcus defines θ to be positive in the counter-clockwise direction and negative in the clockwise direction.

If the point (i,j) is closer to the x-axis than the y-axis and $i > 0$, i.e. if ((abs(i) > abs(j)) && (i > 0)), then if $\theta_1 > 0$ Marcus is at the intersection point with the lower y value. If instead, $\theta_1 <= 0$, he is at the point with a higher y value. If (i,j) is still closer to the x-axis than the y-axis, but $i < 0$, Marcus will be at the opposite intersection points. That is, if $\theta_1 > 0$, then he is at the point with the higher y value and θ_1 is $<= 0$, then he is at the point with the lower y-value.

Instead, if (i, j) is closer to the y-axis, and $j > 0$, then if θ_1 is > 0, Marcus is at the point with the greater x value. If θ_1 is $<= 0$, he is at the point with a lower x coordinate. Finally, if (i, j) is closer to the y-axis, but $j < 0$, if $\theta_1 > 0$ Marcus is at the point with a lower x coordinate and if $\theta_1 <= 0$, he is at the point with a greater x coordinate. It is also possible for there two be only one intersection point. Although it is difficult for the values of r_1 and r_2 to ever be exact enough to yield this outcome, when the previous destination is almost directly behind the agent (θ_1 being close to π or -π), it is assumed to be the case. This means the agent is on a direct line from (0, 0) to (i, j) and (rx, ry) can be calculated as the point on that line that intersects the circle of radius r_2 centered at (i, j).

Once Marcus knows his location in global coordinates, he can calculate the local-space position of the marker for his next destination, (r_3, θ_3). The coordinates of the next destination are in the map and are expressed in "previous destination" coordinates as (dx, dy), so $r_3 = \sqrt{(dx - rx)^2 + (dy - ry)^2}$. The azimuth, θ_3, can be found by using the cosine rule on a tri-

angle with vertices of (dx, dy), (i, j) and (rx, ry). Once Marcus has calculated the position of the new marker, he stores the coordinate of his "current" destination in the task executor. This allows him to treat this point as (0, 0) next time and adjust the map coordinates of all other points relative to it. The reason I do this is just because the circle intersection equations are simpler if one of the circles is centered at the origin.

I chose to use this method for coordinate system conversion as opposed to other triangulation methods [21] because, even though it allows for two different (rx, ry) points, it is simpler than other methods. The domain assumptions that allow Marcus to disambiguate his location work well because his navigation system tends to follow a fairly direct path to the goal. That means he is rarely at the other possible location for (rx, ry) because to arrive there with the same values for r1, θ_1, r2 and θ_2 would mean Marcus took a more circuitous route. While this is possible due to obstacle avoidance, for example, the cost of keeping a third marker up to date is too high. This is especially true considering that in the situation where the two markers for previous destinations are the least expensive to maintain, they are the most inaccurate because they are being dead-reckoned.

A.2.2 Pose Detection

Marcus determines the pose of a block stack based on visibility of the stack's components. The implemented pose detection routine requires that there be three blocks. If the base block is said to have a footprint of size 4, one top block can be said to have a footprint of size 2 and the final top block of size 1. Figure 26 shows several views of an example stack. Based on the visibility of the various blocks in the stack, Marcus determines which of several viewing zones he must be in relative to the stack, the following overhead view
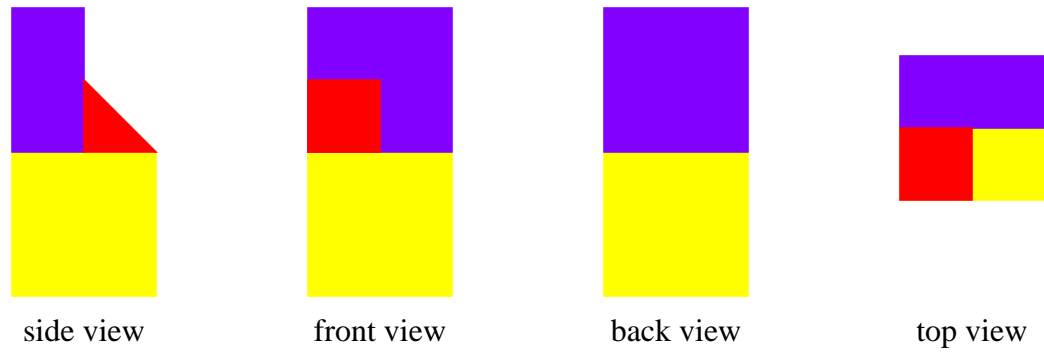
Figure 26. Example Stack Views

(figure 27) shows the different zones. The zones are differentiated by the relative size and position of the widths of the bounding boxes of the blobs associated with the different blocks. The solid black lines separate the major zones that define where the red block's bounding box falls within the yellow box's width. The dotted lines define additional tran-
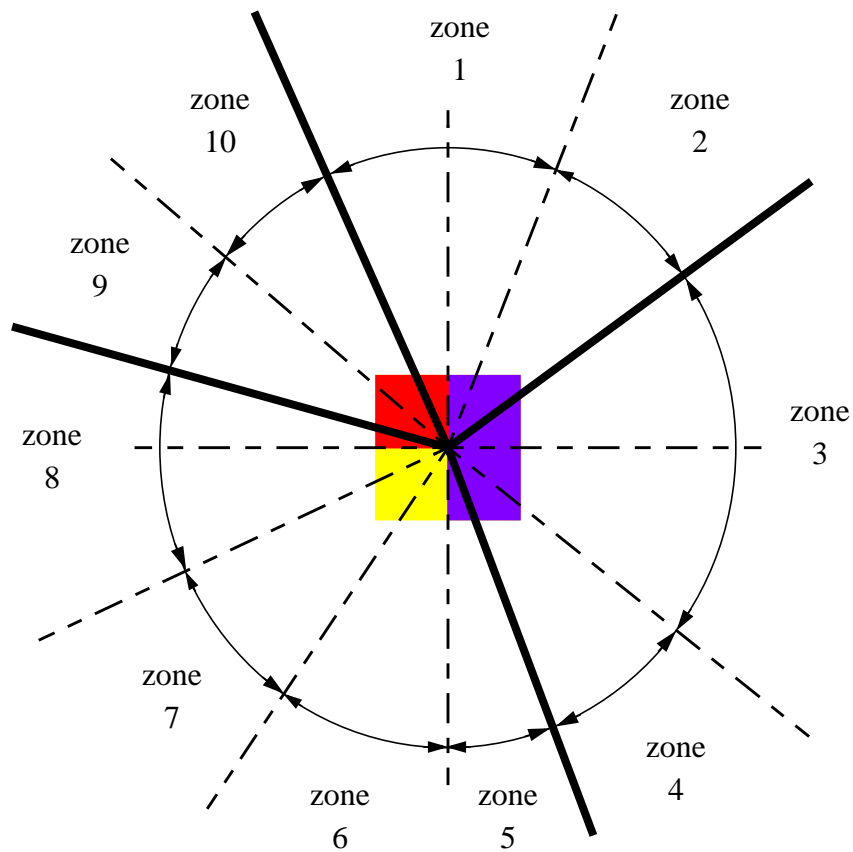

Figure 27. Zones Used in Determine-Item-Pose

sitions between areas where the relationship between the coordinates of the bounding boxes appear differently.

The perceptual conditions for determining which zone the agent is in, as well as what it means to be in that zone are summarized in table A1. The notation X_{min} of P stands for the screen coordinate of the left edge of the purple block's bounding box, while Y_{max} of Y would be the bottom edge of the yellow block's bounding box. Finally, "R is mostly left" means that the red block's bounding box occupies mostly the left portion of the area within the yellow block's bounding box width. These visibility criteria were determine empirically.

The angle returned by the pose detector defines zone 5 as the canonical orientation of the block stack. That is, this position is 0 degrees. The pose detection routine cannot distinguish between the stack being at a fixed position with the agent being at a different locations around it, and the agent being at a fixed position with the stack being rotated on its Z axis. In other words, the pose detector can only determine the local orientation of the stack with respect to the agent and not the agent's position in global coordinates. The agent uses the stack's pose to determine the orientation of a local assembly site coordinate system. If the plan does not care about the orientation of the construction in global coordinates, this ambiguity is not a problem. However, if the construction needs to be at a certain position in global coordinates, the agent must translate between the local assembly-site coordinates stored in the stack representation and the global coordinates of the map. In this case, Marcus must make an assumption about the pose of the stack. That is, he must assume that any changes in pose are due either to his position around the stack, or to the stack's own rotation. Marcus has a system for converting between local and global coordinates, so he uses

that to determine his position in the map. He then assumes that any change in the stack's

pose is due to the rotation of the stack on its own axis (entities in Marcus' domain are some-

times moved by devious users when Marcus' back is turned!).

Table A1: Summary of Pose Zones

Major Zone	Pose Zone	Condition	Angle
R is mostly right (at least 80% of R's width is to the right of the midpoint of Y's width)	1	(Y_{max} of P - Y_{max} of R) > -1	180
	2	(Y_{max} of P - Y_{max} of R) <= -1	-135
R is occluded (no portion of R is visible)	3	(X_{min} of P - X_{min} of Y) > 5	-90
	4	(X_{min} of P - X_{min} of Y) <= 5	-35
R is mostly left (at least 80% of R's width is to the left of the midpoint of Y's width)	5	abs(Xmin of P - Xmax of R) < 5	0
	6	X_{min} of P is within right 55% of R's width	34
	7	X_{min} of P is right of the left 20% and left of the right 55% of R's width	68
	8	X_{min} of P is within left 20% of R's width	90
R is in the middle (R is neither 80% left nor 80% right of the midpoint of Y's width)	9	X_{max} of P is within right 20% of R's width	124
	10	X_{max} of P is within left 80% of R's width	158

A.2.3 Detecting Adjacency

Marcus detects color adjacency using a technique called "springy dumbbells". Each

dumbbell consists of two square images regions separated by a certain distance along a line

segment. This is shown in figure 28 where the dumbbell ends are shown as black rectangles

connected by a black line. The line segment determines the kind of adjacency the dumb-
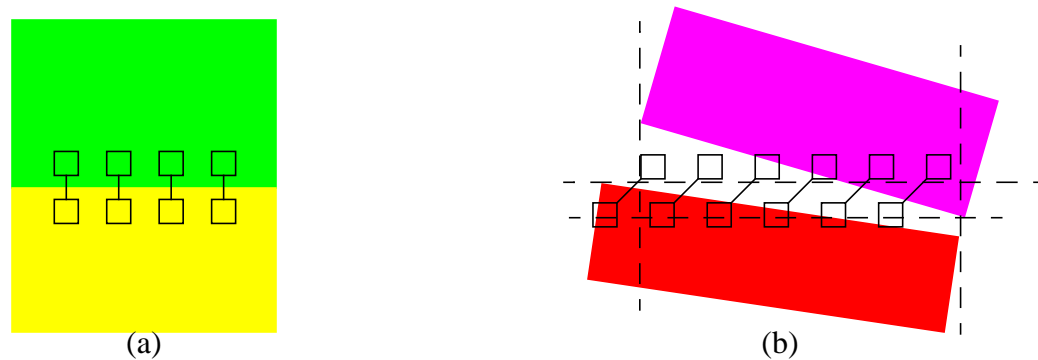
Figure 28. Example Adjacency Measurements

bells can detect. For example, a vertical segment checks for the "stacked" relationship, i.e. one block on top of another. A horizontal segment can be used to detect the "next to" relationship, i.e. two blocks side by side. In general, the line can be described by any means (e.g. slope/intercept), but Marcus only implements horizontal and vertical lines and only uses vertical lines for the arch building task.

Detecting adjacency begins with blob coloring the input image on the two colors (which I will call color1 and color2) for which Marcus wants to check adjacency. For example, if Marcus wants to look for a yellow block with a particular adjacency to a green block, he must first find all the green and yellow blobs. Dumbbells are placed at various locations within a search region. Once one dumbbell end is placed, it is a "fixed point" and the location of the other end will be determined based on the line segment. If enough pixels within one dumbbell end are of color1 and enough in the other end are of color2, then the dumbbell votes for the adjacency. Marcus' dumbbells are 10x10 pixels and at least 75% of them must be of the correct color (75 pixels). If a dumbbell votes for a relationship, another dumbbell is tried moving from left to right (at the same y-coordinate) within the search region. If a dumbbell does not vote for the adjacency, two different measures are taken. First, the

dumbbell separation is altered (hence the dumbbells are "springy"). The dumbbells shown in figure 28b are at the maximum separation, while those of 28a are not. If all possible separations have been tried and none has made the dumbbell vote for the adjacency, the "fixed" dumbbell end can be moved. The dumbbell moves up and down, left to right through the search region. This search pattern is used because when an adjacency is found, the next dumbbell starts at the same y-coordinate as the last one. This is useful when looking for mostly vertical adjacencies which are most common for Marcus' task.

The search region in which the dumbbell analysis is carried out is defined by the bounding rectangles of the colored blobs. The region extends from the larger of the two left bounding box edges to the smaller of the two right bounding box edges. It also runs from the smaller of the bottom of box1 and the top of box 2 to the other. The dashed lines of figure 28b illustrate the minimal bounding box region. This region is then extended, top, bottom, left and right by the size of one dumbbell end (10 pixels here).

The adjacency routine compares the number of positive voting dumbbells to the total number of attempted dumbbell positions. If more than a certain percentage (70%) voted for the relationship, then Marcus believes it exists. The dumbbells of figure 28a all vote for their adjacency and so these blocks would be called stacked. The dumbbells in figure 28b do not vote for their adjacency and so the blocks would be considered separate.

A.2.4 Handling Completed Actions in Marcus' Task Executor

Table A2 summarizes what happens when the PA layer completes an action and passes a marker to the TE. Based on the current plan step and the result of the completed action, the TE will modify its own state, the state of the protomarker associated with the returned

marker and possibly other protomarkers and markers as well. Note that some completed actions are treated differently depending on whether the TE is involved in sequencing a set of PA layer actions for a single plan step. Columns with "in the midst of..." in parenthesis indicate how completed actions are handled during an ongoing sequence. Unless otherwise stated, the Completion result for the action is SUCCESS (the universal code for the action completed as expected). The ordering of the table is the order in which the completed action and its result code are examined.

Table A2: How TE Handles Completed PA Layer Actions

Completed PA Layer Action	TE's Response
align-with-hallway (in midst of find-and-go-through-hall-door)	Change marker's Action component to center-in-hallway.
center-in-hallway (in midst of find-and-go-through-hall-door)	Create a **destination** marker in middle of hallway on same side of Marcus as the door and put a rotate-base-to-item action on it (put **door** marker in its dependency list).
rotate-base-to-item (in midst of find-and-go-through-hall-door)	Change marker's Action component to find-hall-door.
find-hall-door (in midst of find-and-go-through-hall-door)	This marker should be a **destination** marker with the **door** marker and another **destination** marker associated with a position inside the detected door, in its dependency list. Put an align-base-and-turret action on the inside the door destination marker and delete the marker for the completed action. Also update the current and previous destinations for the local-to-global conversion described in section A.2.1. The current destination will be the inside **destination** marker and the previous destination will be the **door** marker.
align-base-and-turret (in midst of find-and-go-through-hall-door)	Change marker's Action component to go-through-hall-door
go-through-hall-door (in midst of find-and-go-through-hall-door)	The action is complete. Increment the plan step counter to move to the next plan step.

Table A2: How TE Handles Completed PA Layer Actions

Completed PA Layer Action	TE's Response
delete-marker (in midst of find-and-go-through-hall-door)	Same as delete-marker below. This is handled separately in here because all completed actions during a find-and-go-through-hall-door are diverted to a separate handler.
move-to-door with result NO_DOOR (in midst of find-and-go-through-door)	Create new **destination** marker at a new position and place a rotate-to-item action on it. The new positions will be progressively larger angular offsets from the door's original position, alternating in both directions. This is how rotate-to-view-door is implemented.
rotate-to-item (in midst of find-and-go-through-door)	Set the **door** marker's Action to find-door again and delete the **destination** marker. The agent will now look for the door again and hopefully the image will be better aligned.
move-to-door with result SUCCESS (in midst of find-and-go-through-door)	The action has completed and Marcus should now be in the hallway. Adjust the current and previous destinations for the local-to-global conversion. Current destination should be the **destination** marker in the hallway (the one estimated by the baseboard's position) and the previous position should be the **door** marker. Increment the plan step counter.
delete-marker (in midst of find-and-go-through-door)	See delete-marker (in midst of find-and-go-through-hall-door)
delete-marker (in midst of delete-marker plan step)	Remove this marker from the proto-marker dependency list that it is in. If protomarker for this marker has multiple markers then they will all be deleted, so don't increment the plan step counter until the last one comes back with a completed delete action.
search-for-item (in midst of span-item1-and-item2-with-spanner)	Change marker's Action component to look-at-item so that it gets centered in the view and add a monitor to wait for the **topblock** marker of the stack to get instantiated.

Table A2: How TE Handles Completed PA Layer Actions

Completed PA Layer Action	TE's Response
stack-item1-on-item2 (in midst of span-item1-and-item2-with-spanner) **OR** marker's Index is **topblock** **AND** the protomarker for this marker has an Index indicating it is item1	Either the item1 stack was just re-stacked and so the **topblock1** marker is instantiated, or the monitor function just fired (also meaning the **topblock1** marker is instantiated). Now, place a search-for-item action on the base marker for the protomarker specified by the plan step as representing item2.
stack-item1-on-item2 (in midst of span-item1-and-item2-with-spanner) **OR** marker's Index is **topblock** **AND** the protomarker for this marker has an Index indicating it is item2	Either the item2 stack was just re-stacked and so the **topblock2** marker is instantiated, or the monitor function just fired (also meaning the **topblock2** marker is instantiated). Put the **topblock2** marker and the **spanner** marker in the **topblock1** marker's dependency list and put the span-topblock1-and-topblock2-with-spanner action in its Action component.
put-down-item (in midst of put-down-item or put-down-item1-relative-to-item2) with result CANT_FIND	The putdownloc is not in view and so the agent must rotate to view it. This happens because the move-to-good-putdownloc-view task does not account for the azimuth of the point it is trying to view (it simply figures out what image Y coordinate the putdownloc should be at and if its not far enough from the bottom of the image, it backs Marcus up). This is compensated for by putting a rotate-to-item action on the putdownloc marker.
rotate-to-item (in midst of put-down-item or put-down-item1-relative-to-item2)	Put a put-down-item action on the marker.

Table A2: How TE Handles Completed PA Layer Actions

Completed PA Layer Action	TE's Response
put-down-item (in midst of put-down-item or put-down-item1-relative-to-item2) with result SUCCESS	If this marker's protomarker represents a stack, make sure that the two markers are in the PA layer and have the correct Identify routines (their associated objects can now be seen by the camera). Set all the markers in the protomarker to uninstantiated and increment the plan step counter.
delete-marker	This is an "out of order" deletion completing, i.e. the current plan step is not a marker deleting step. Look through the Dependency List of each protomarker and delete this marker if its there. Don't increment the plan step because this is out of order and so doesn't represent the end of a step.
put-item-in-toolbelt	Set the position in the Property component of the protomarker to "in toolbelt" and increment the plan step
pick-up-item	Set the position in the Property component of the protomarker to "in hand" and increment the plan step.
go-to-destination	Set the "previous" destination to be this marker. Increment the plan step counter.
stack-item1-on-item2 (in midst of a determine-item-pose)	Marcus has just completed a restack that was needed because determine-item-pose could not find a stack as expected. Create a marker for the fastener block if it doesn't already exist and place a determine-item-pose action on it. Place markers for top and bottom blocks in its dependency list. All three markers should be in the protomarker's dependency list.

Table A2: How TE Handles Completed PA Layer Actions

Completed PA Layer Action	TE's Response
stack-item1-on-item2	Set the Index of the marker for item2 to **top-block** and set its Property to the same position as the bottom block. Then set the Identify routines of both the **item1** and **item2** markers to be the appropriate XonY Locate and Track routines. Set Index of one of the protomarkers (either for **item1** or **item2**) to 'stack' and place both markers in its dependency list. Delete the other protomarker since there is now a single, multi-part entity associated with the stack protomarker. Increment plan step counter.
rotate-to-item	Increment plan step counter.
determine-item-pose	Store the result code in the Prop field of the Property component of the protomarker. This is the pose returned by the action and it should be stored in Prop because that is where data known about the associated object, but not maintained by the PA layer is stored. Increment plan step counter.
all other actions with result SUCCESS	Increment plan step counter.
any action still unhandled	Abort program and halt agent.

A.2.5 How the TE Starts a New Plan Step

Table A3 shows what actions the TE takes when beginning any of the various plan steps. Before doing any of these actions, the DetermineAction step of the TE's main loop creates protomarkers and markers (including the global-to-local coordinate conversion) for the step, if they are needed. If markers are created, they are added to the PA layer. Note that when markers are created for a stack, the bottom marker is placed in the dependency list first and will be the marker that all actions are placed on unless otherwise specified.

Table A3: TE Actions when Starting a New Plan Step

Plan Step Wants This Action to be Executed	TE's Actions
add	Create a marker at the agent's location to initialize the "previous" destination in the global/local converter.
delete-protomarker	Deletes the specified protomarker from the TE. Its markers (if any) should have already been deleted by previous steps.
go-to-destination (where destination is an entity at some location in the map)	Make the "current" destination be the "previous" destination and make the newly created **destination** marker be the "current" destination. From now on I refer to this as moving the local/global origin. The marker created at the start of the TE's DetermineAction step will have the correct ego-centric position and go-to-destination in its Action component.
find-and-go-through-door	Place find-door action on **door** marker. Protomarker needs to already have a **door** marker in its dependency list.
rotate-to-item	Place rotate-to-item action on this protomarker's marker and make that marker's Index **item**.
search-for-item	If the protomarker represents a stack, change the bottom block marker's Locate routine to YLocate and the top block marker's Locate routine to XonInstYLocate. Set up a monitor to watch for the top block marker's instantiation. Put the search-for-item action on the bottom block marker (for stacks) or on the single block's marker (if the protomarker does not represent a stack). Set that marker's Index to **item**.
pick-up-item	The associated object should be in view, so verify the relationship specified by the protomarker. If the relationship is not present, then set the action on any detected component's marker to recreate the relationship. For stacks, the bottom block's marker will get a stack-item1-on-item2 action. If the protomarker's relationship is verified, set the appropriate block's marker's Action to pick-up-item (so the bottom block for stacks, or just the block for singletons).

Table A3: TE Actions when Starting a New Plan Step

Plan Step Wants This Action to be Executed	TE's Actions
put-item-in-toolbelt	If protomarker is a stack, verify that the blocks are still stacked. Then place a put-in-toolbelt action on the appropriate marker.
put-down-item	Find the protomarker for the putdownloc. If it does not have a **putdownloc** marker, create one using the global-to-local conversion based on the agent's location and the position for the object specified in the plan. Add **putdownloc** marker to appropriate object marker's dependency list and set that marker's Action to put-down-item. Add **putdownloc** marker to PA layer.
put-down-item1-relative-to-item2	Find protomarker for item2 and calculate the putdown-loc position based on item2's location and the relative offset specified in the plan step. Then convert this to a local space coordinate and create a **putdownloc** marker. The remainder is the same as put-down-item.
make-from	This action is used to initialize the local/global conversion system. The "add" action initializes the first "previous" destination and make-from initializes the first "current" destination. It creates a new **from** marker at the same position as the marker in this step's protomarker.
delete-marker	Put a delete-marker action on all the markers in this step's protomarker's dependency list.
stack-item1-on-item2	Find protomarkers for item1, item2 and the fastener block. The agent needs to have a fastener in hand or in the toolbelt to do the stacking action, but the PA layer does not look for it to know that the stack is built properly). Set the Identify routines for the markers of item1 and item2 to be XonYLocate and XonYTrack and put them in each other's dependency lists. Place the stack-item1-on-item2 action on item1's marker. Note that while it should be possible for this action to be taken on any block structure, Marcus only has it implemented for stacking two singleton blocks.

Table A3: TE Actions when Starting a New Plan Step

Plan Step Wants This Action to be Executed	TE's Actions
find-and-go-through-hall-door	Determine which side of the hallway the door is on (left or right relative to agent). This is based on agent's position and position of the door in the map. Set **door** protomarker's marker's Action to align-with-hallway.
determine-item-pose	Find protomarkers for bottom and top blocks and verify that the relationship between then still exists (remember this action only works on stacks). If the stack has a fastener marker, put the determine-item-pose action on it, otherwise create the marker and then place the action on it.
span-item1-and-item2-with-spanner	Find protomarkers for item1 and item2 and create markers for them if necessary. If the **base** marker for item1 is uninstantiated, put a search-for-item action on it. Otherwise, if the **base** marker for item2 is uninstantiated, put a search-for-item on it. If both are instantiated, and the **topblock** markers are also instantiated, put them in the **spanner** marker's dependency list and put a span-item1-and-item2-with-spanner action on the **spanner** marker. Note that the implementation does not allow the spanner to be a compound object.
say	Add the say action to the protomarker's marker with the "string to speak" specified in the plan step as a parameter. This step is just used to say "plan completed successfully" at the end of plan execution.

Appendix B   *Marcus Task Figures*

B.1. Marcus Task Flow Diagrams

   This section presents task flow diagrams for the task decomposition shown in chapter 6, figures 21 through 23. The "double arrows" (see figure 30a) show a continuation of control flow from one sub-diagram to another. The flow diagram for the assembly plan shown in chapter 6, figures 19 and 20 is not shown here because those tasks simply execute sequentially, in alphabetic order.
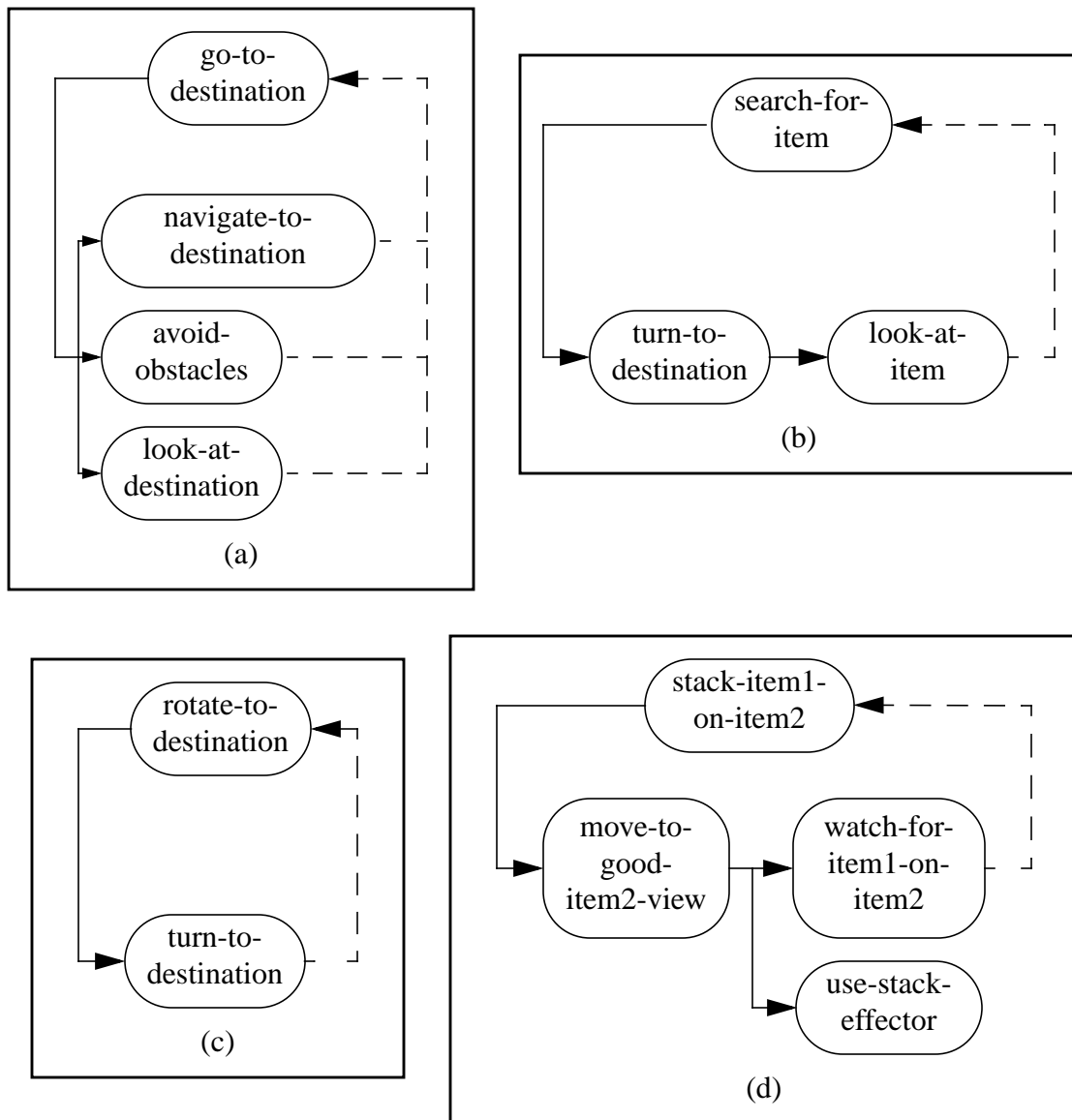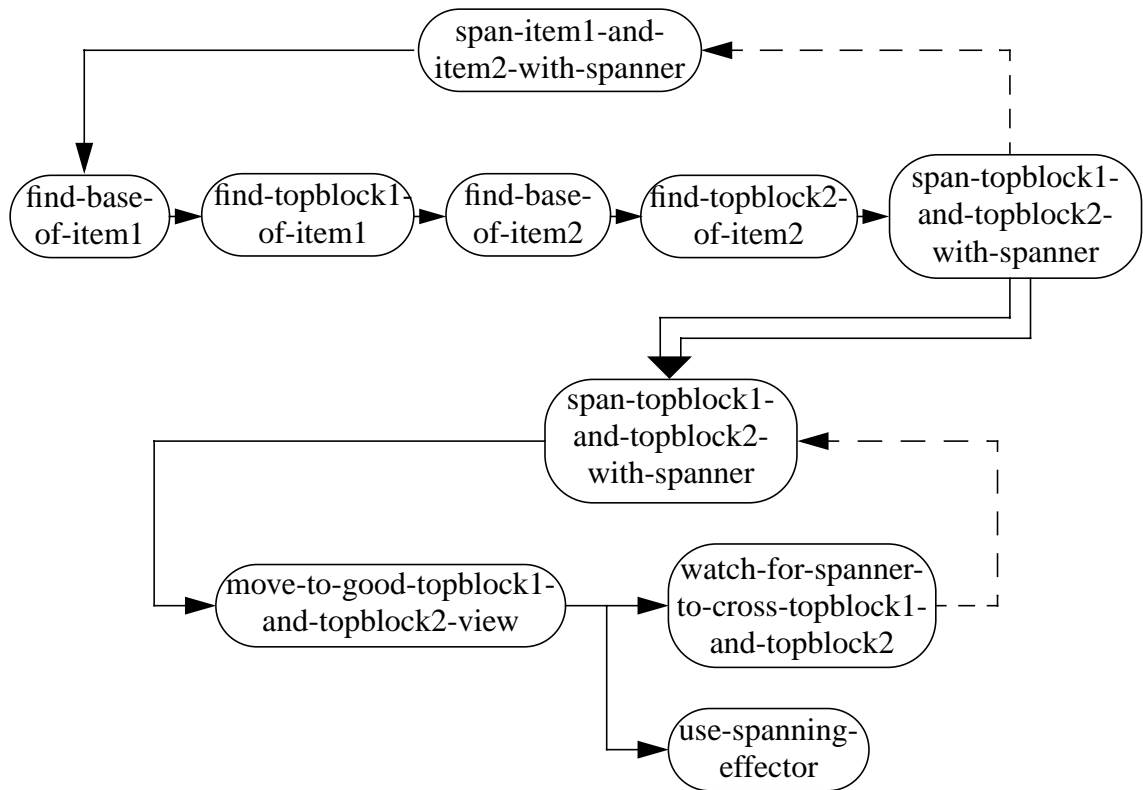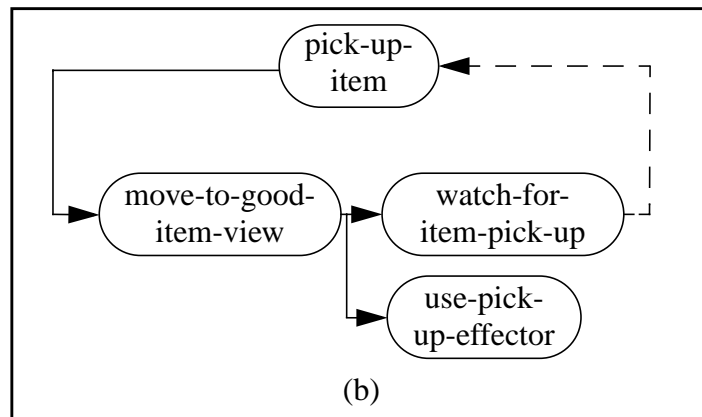
Figure 29. Marcus Task Flow Diagram (1)

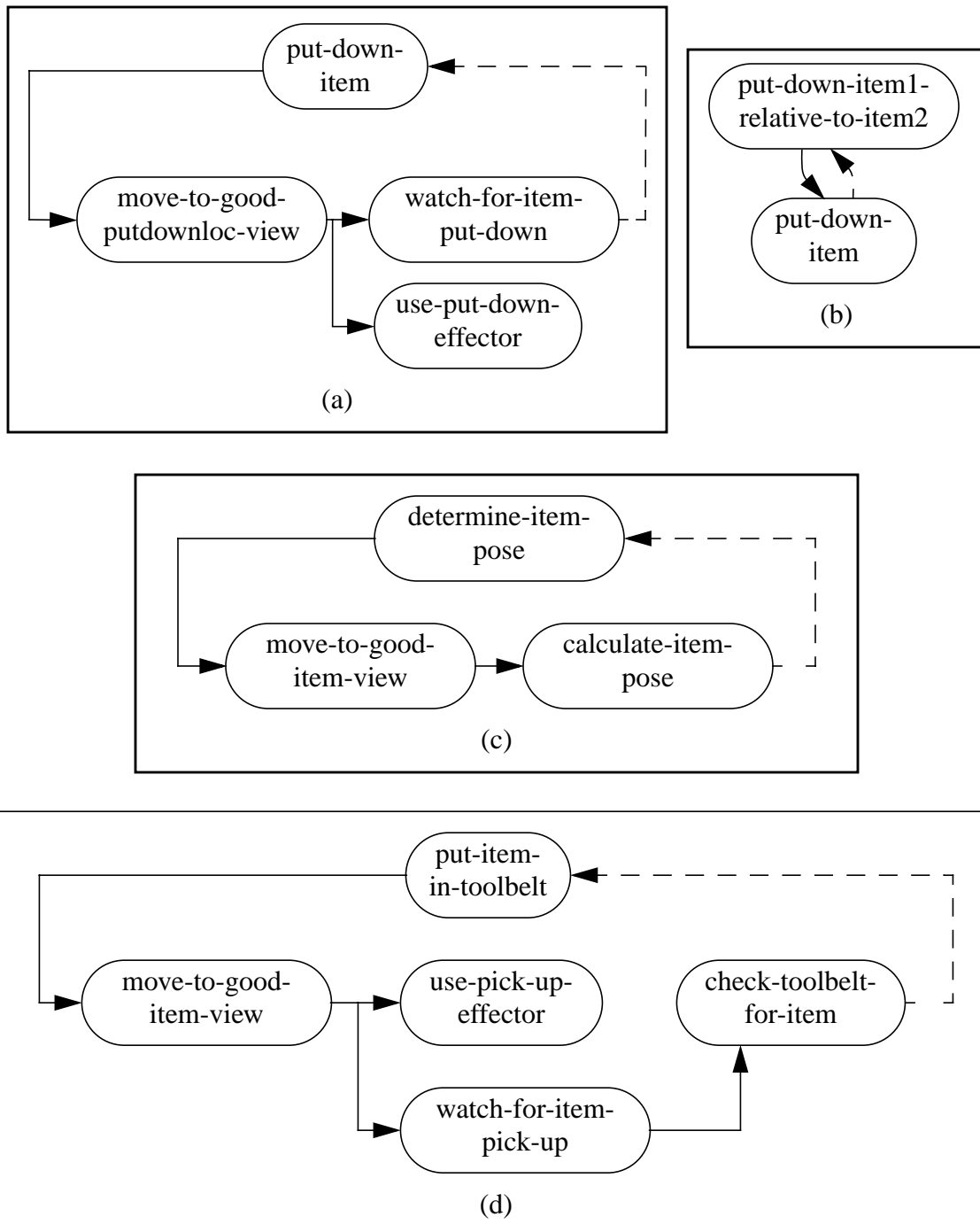Figure 30. Marcus Task Flow Diagram (2)
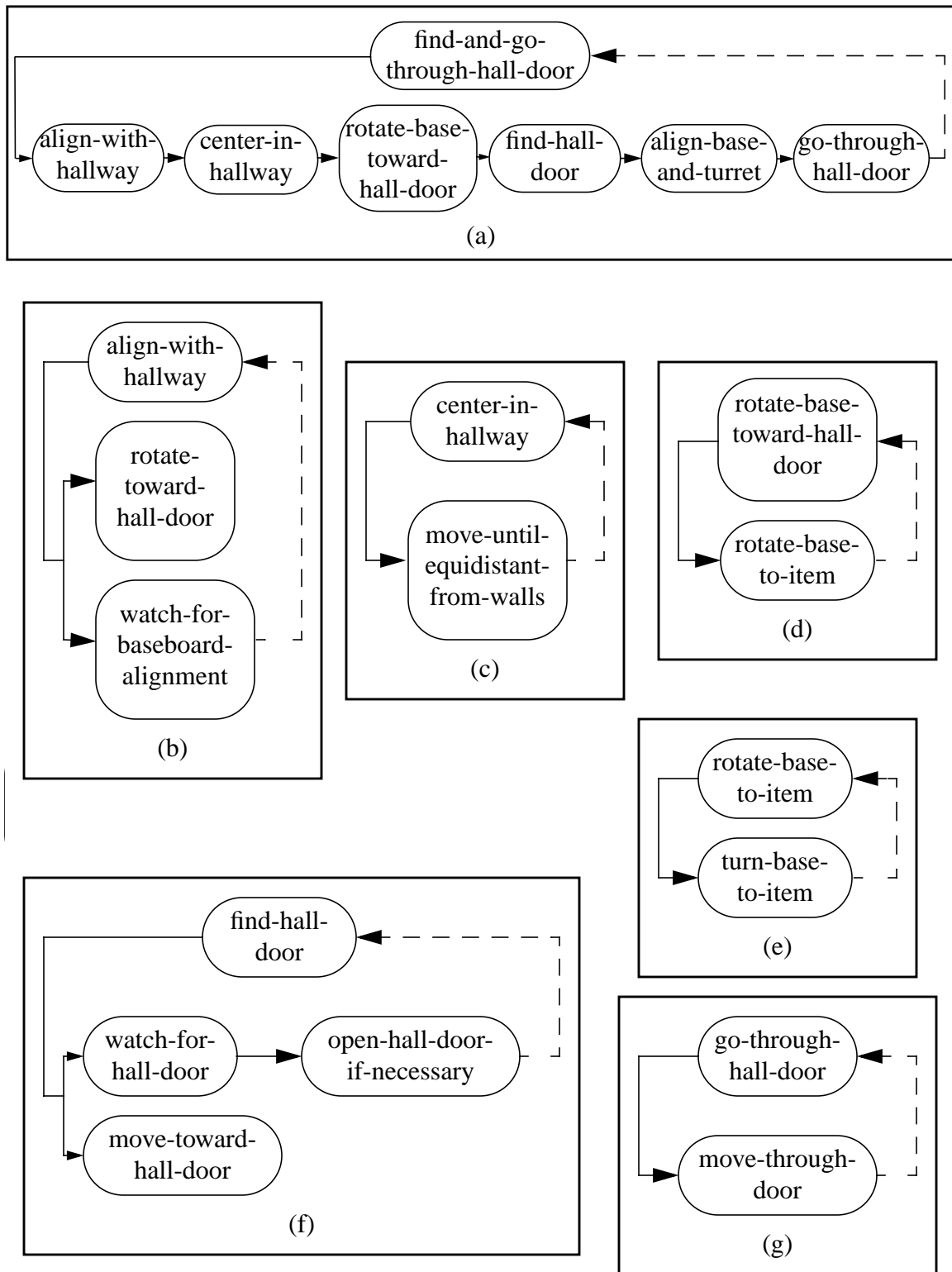
Figure 31. Marcus Task Flow Diagram (3)

Figure 32. Marcus Task Flow Diagram (4)
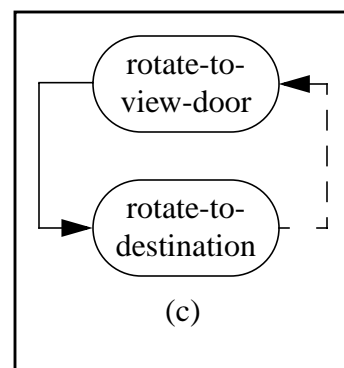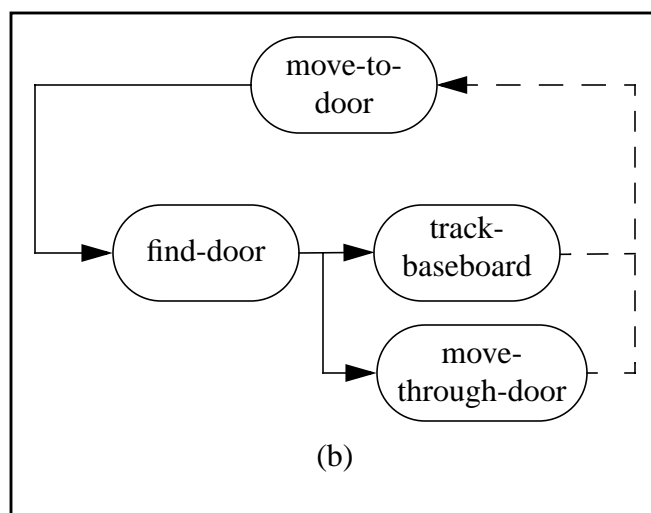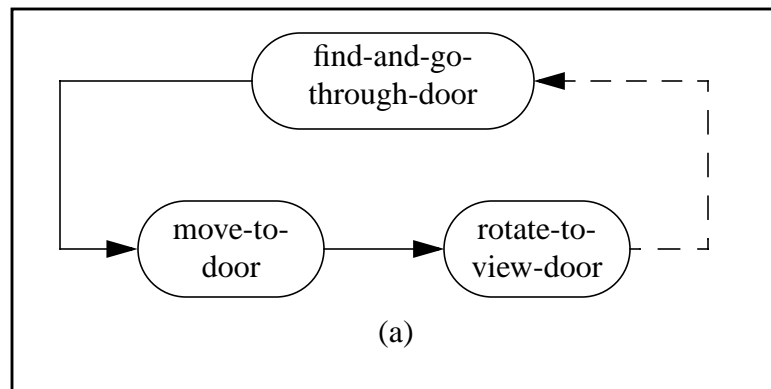
Figure 33. Marcus Task Flow Diagram (5)

(a)

go-to-**block**

search-for-
base-block

go-to-**base-
block**

pick-up-
base-block

(b)

search-for-
fastener

go-to-
fastener

put-**fastener**-
in-toolbelt

(c)

search-for-
assembly-site

put-down-
base-block

Roles
base-block
assembly-site

(d)

go-to-**top-
block**

pick-up-
top-block

go-to-
assembly-site

stack-**top-block**-
on-**base-block**

(e)

go-to-
fastener

rotate-to-
door

go-through-
door

(f)

search-
for-**stack**

go-to-**stack**

pick-up-
stack

go-through-
hall-door

Figure 34. Top Level Task Roles for Marcus

(g)

search-for-**stack**   go-to-**stack**

(h)

determine-**stack**-pose   put-down-**stack1**-relative-to-**stack2**

(i)

search-for-**spanning-block**   go-to-**spanning-block**   pick-up-**spanning-block**

(j)

search-for-**fastener**   go-to-**fastener**   rotate-to-**stack**   go-to-**stack**

(k)

determine-**stack**-pose   span-**stack1**-and-**stack2**-with-**spanning-block**
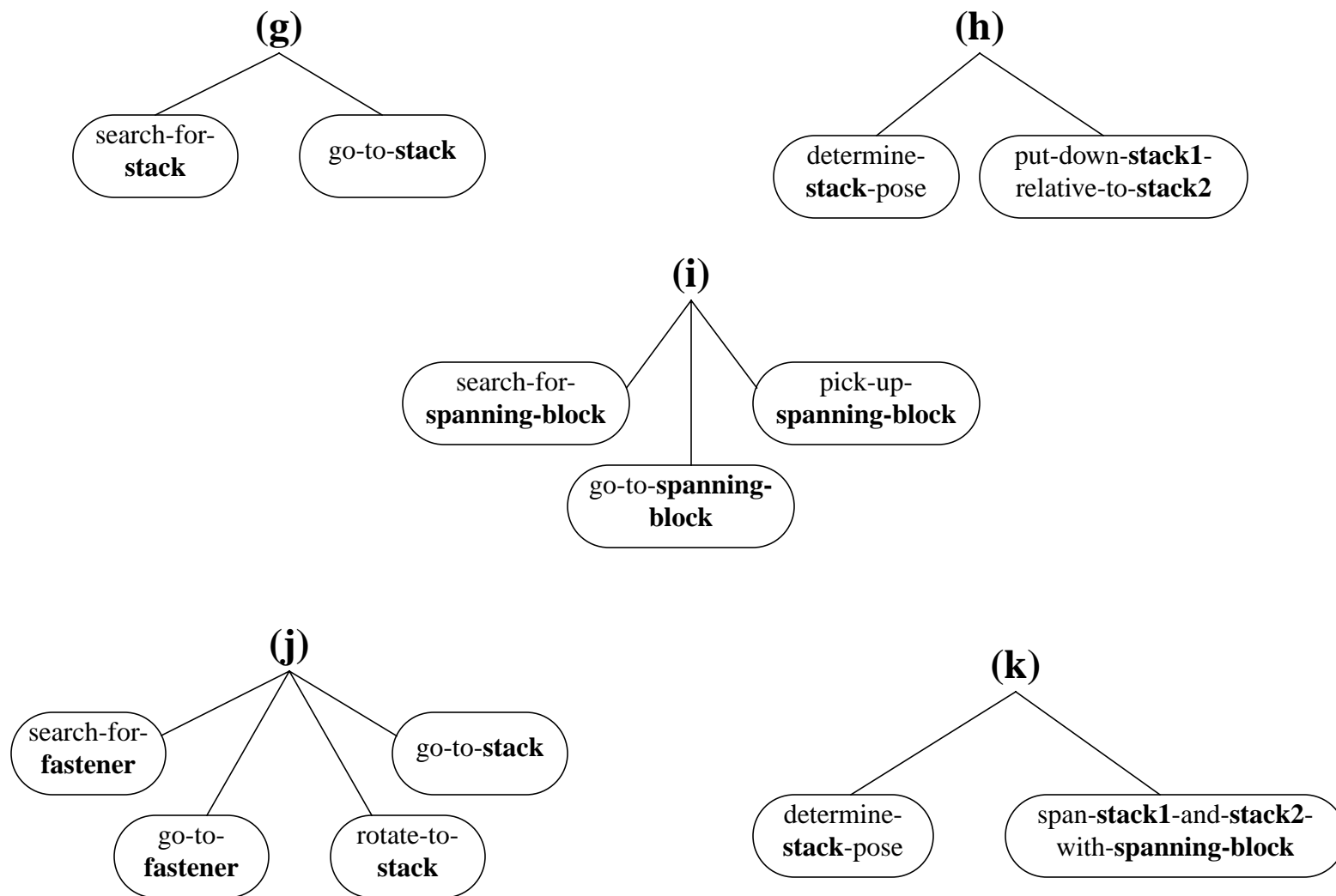
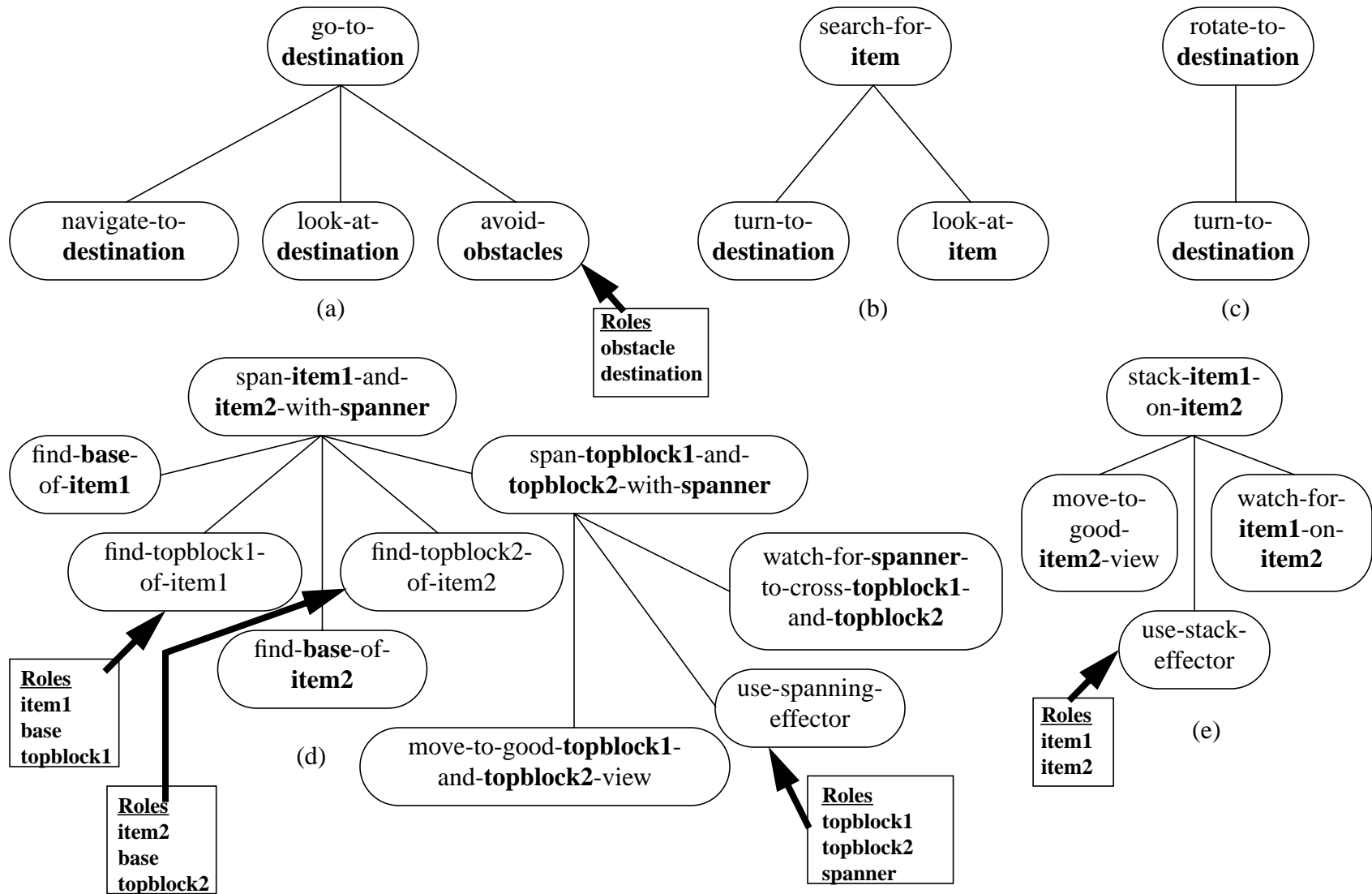Figure 35. Top Level Task Roles for Marcus (continued)

Figure 36. Locomotion and Building Task Roles for Marcus
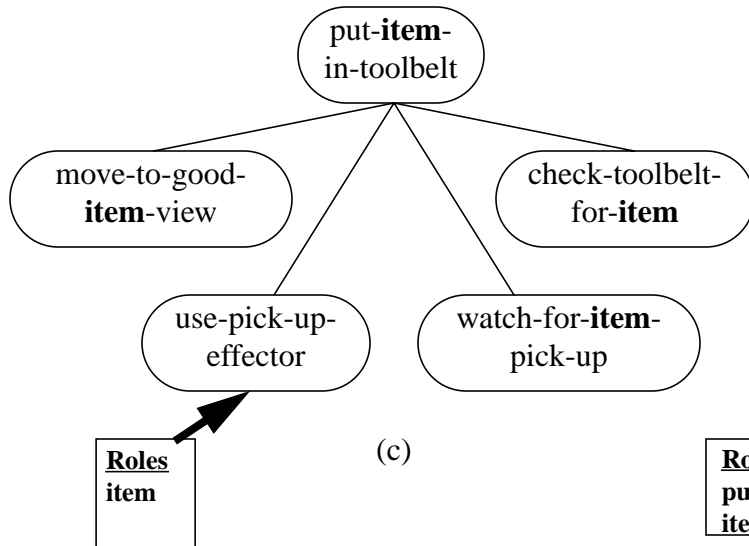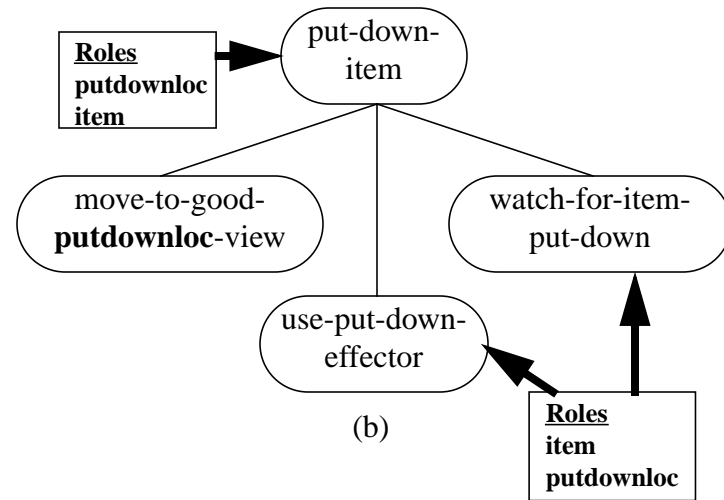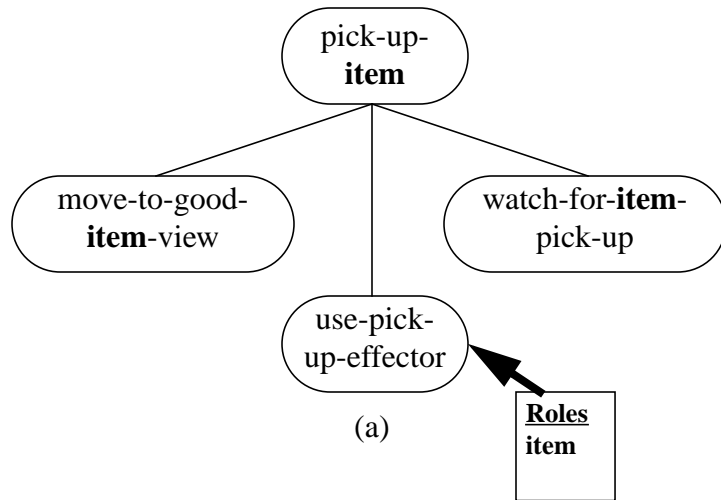
Figure 37. Manipulation Task Roles for Marcus
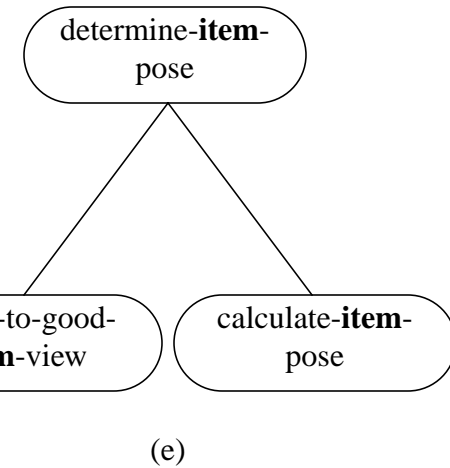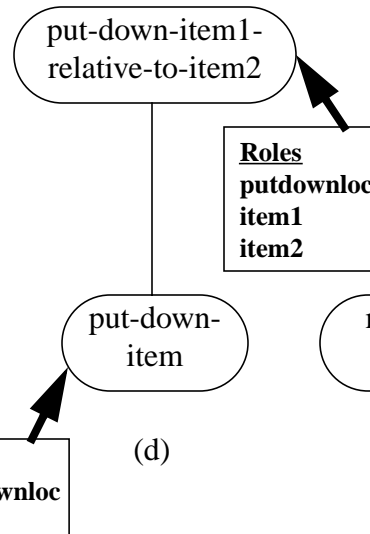
Roles
hallway
hall-door

find-and-go-through-
hall-door

Roles
hall-door
destination

align-with-
hallway

center-in-
hallway

rotate-base-
toward-**hall-door**

find-**hall-
door**

align-base-
and-turret

go-through-
hall-door

rotate-
toward-
hall-door

watch-for-
baseboard-
alignment

move-until-
equidistant-
from-walls

rotate-base-
to-**item**

watch-for-
hall-door

open-**hall-door**-
if-necessary

move-through-
door

Roles
front-wall
back-wall

turn-base-
to-**item**

move-toward-
hall-door

(a)

Roles
door
destination

find-and-go-
through-**door**

Roles
hall-door
left-edge
right-edge

move-to-
door

Roles
door
destination

rotate-to-
view-**door**

(b)

Roles
door
destination

move-through-
door

track-
baseboard

find-door

rotate-to-
destination

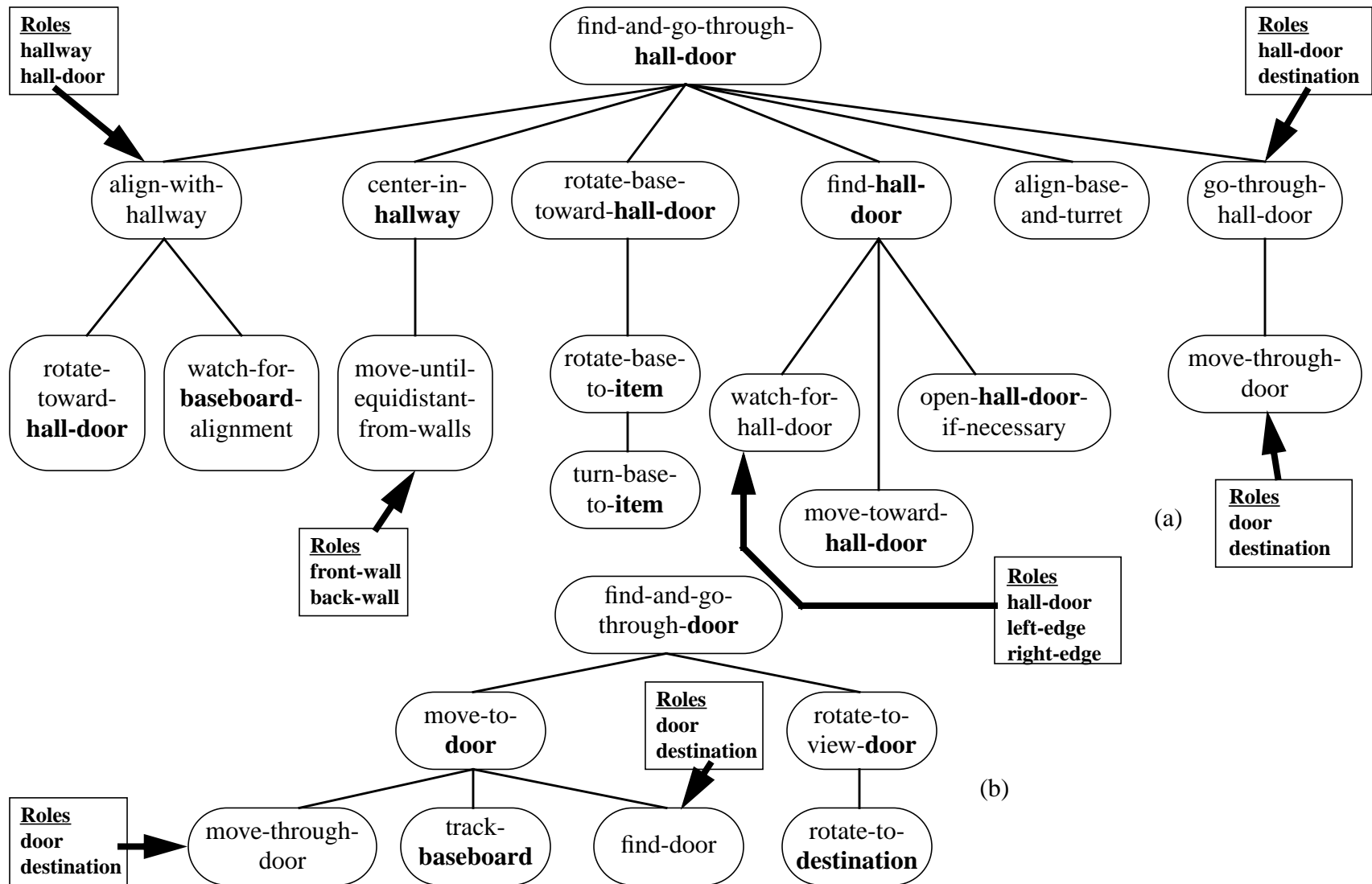Figure 38. Door Task Roles for Marcus

B.3. Marcus Role Mappings

This table summarizes the mappings between the roles in the steps of Marcus' assembly plan and the roles in the tasks that implement it.

Table B1: Role Mappings

Plan Step	Implemented Using	Plan Step Role	Mapped to role
go-to-block	go-to-destination	block	destination
search-for-base-block	search-for-item	base-block	item
go-to-base-block	go-to-destination	base-block	destination
pick-up-base-block	pick-up-item	base-block	item
search-for-fastener	search-for-item	fastener	item
go-to-fastener	go-to-destination	fastener	destination
put-fastener-in-toolbelt	put-item-in-toolbelt	fastener	item
search-for-assembly-site	search-for-item	assembly-site	item
put-down-base-block	put-down-item	base-block	item
		assembly-site	putdownloc
go-to-top-block	go-to-destination	top-block	destination
pick-up-top-block	pick-up-item	top-block	item
go-to-assembly-site	go-to-destination	assembly-site	destination
stack-top-block-on-base-block	stack-item1-on-item2	top-block	item1
		base-block	item2
go-to-fastener	go-to-destination	fastener	destination
rotate-to-door	rotate-to-destination	door	destination
go-through-door	find-and-go-through-door	door	door
search-for-stack	search-for-item	stack	item
go-to-stack	go-to-destination	stack	destination
pick-up-stack	pick-up-item	stack	item

Table B1: Role Mappings

Plan Step	Implemented Using	Plan Step Role	Mapped to role
go-through-hall-door	find-and-go-through-hall-door	hall-door	hall-door
search-for-stack	search-for-item	stack	item
go-to-stack	go-to-destination	stack	destination
determine-stack-pose	determine-item-pose	stack	item
put-down-stack1-relative-to-stack2	put-item1-relative-to-item2	stack1	item1
		stack2	item2
search-for-spanning-block	search-for-item	spanning-block	item
go-to-spanning-block	go-to-destination	spanning-block	destination
pick-up-spanning-block	pick-up-item	spanning-block	item
search-for-fastener	search-for-item	fastener	item
go-to-fastener	go-to-destination	fastener	destination
rotate-to-stack	rotate-to-destination	stack	destination
go-to-stack	go-to-destination	stack	destination
determine-stack-pose	determine-item-pose	stack	item
span-stack1-and-stack2-with-spanning-block	span-item1-and-item2-with-spanner	stack1	item1
		stack2	item2
		spanning-block	spanner

Appendix C *Task Constraints*

This appendix contains a list of constraints for the walk-the-dog task of chapter 3. These constraints also appear throughout chapter 3, particularly in figure 1 and task specification 1. However, they are repeated here for convenience.

C.1. Walk-the-dog Task Constraints

1. The agent's environment consists of 2 intersecting roads with sidewalks, a crosswalk, a park, at least one dog, moving cars, a mailbox on the sidewalk, tulips, roses, a ball and a leash.
2. The cars will always be in the street and moving.
3. The dog to be walked (as distinct from other dogs that may be in the environment) can move on its own.
4. This dog should be on a leash.
5. The agent should walk the dog to the park along the route specified in figure 1 (along the sidewalk on the right side of the street, across the one crosswalk, and then into the park).
6. The dog must be kept out of the street (except when crossing at the crosswalk).
7. Both the dog and the agent should avoid cars in the street.
8. The dog must be kept out of the flowers (tulips and roses).
9. The dog must be kept on the sidewalk.
10. The agent can only control the dog's position by using the leash, i.e. the agent cannot pick up the dog and carry it to the park.
11. The agent should play fetch with the dog after reaching the park.
12. The agent should use the ball to play fetch.
13. The dog will retrieve the ball when it is thrown.
14. The agent should stop playing fetch when the dog appears tired.

References

[1] Agre, P.E. and Chapman, D. 1990. What Are Plans For? *Robotics and Autonomous Systems* 6: 17-34.

[2] Agre, P.E.; and Chapman, D. 1987. Pengi: An Implementation of a Theory of Activity. *AAAI-87*: 268-272.

[3] Albus, J.S. 1997. The NIST Real-Time Control Architecture (RCS): An Approach to Intelligent Systems Research. *Journal of Experimental and Theoretical Artificial Intelligence* 9: 157-174.

[4] Aloimonos, J. 1988. Active Vision. *International Journal of Computer Vision* 1(4): 333-356.

[5] Arkin, R.C. and Balch, T. 1997. AuRA: Principles and Practices in Review. *Journal of Experimental and Theoretical Artificial Intelligence* 9(2): 175-189.

[6] Attneave, F. and Farrar, P. 1977. The Visual World Behind the Head. *American Journal of Psychology* 90(4): 549-563.

[7] Ballard, D.H., Hayhoe, M.M., Pook, P.K. and Rao, R. 1997. Deictic Codes for the Embodiment of Cognition. *Behavior and Brain Sciences* 20: 723-767.

[8] Ballard, D.H. 1990. Animate Vision. *Artificial Intelligence* 48: 57-86.

[9] Ballard, D.H. and Brown, C.M. 1982. *Computer Vision*. Prentice Hall. Englewood Cliffs, NJ.

[10] Bonasso, R., Firby, R., Gat, E., Kortenkamp, D., Miller, D. and Slack, M. 1997. Experiences with an Architecture for Intelligent, Reactive Agents. *Journal of Experimental and Theoretical Artificial Intelligence*, 9(2): 237-256.

[11] Booch, G. 1991. Object Oriented Design with Applications. The Benjamin/Cummings Publishing Company, Inc. Redwood City, CA.

[12] Borenstein, J. and Koren, Y. 1991. The Vector-Field Histogram - Fast Obstacle Avoidance for Mobile Robots. *IEEE Transactions on Robotics and Automation* 7(3): 278-288.

[13] Brill, F.Z., Wasson, G.S., Ferrer, G.J. and Martin, W.N. 1998. The Effective Field of View Paradigm: Adding Representation to a Reactive System. *Journal of Engineering Applications of Artificial Intelligence* 11: 189-201.

[14] Brill, F.Z. 1996. Representation of Local Space in Perception/Action Systems: Behaving Appropriately in Difficult Situations. Ph.D. Dissertation. Department of Computer

Science. University of Virginia.

[15] Brooks, R.A. 1991. Intelligence Without Representation. *Artificial Intelligence* 47: 139-159.

[16] Brooks, R.A. 1990. Elephants Don't Play Chess. *Robotics and Autonomous Systems* 6: 3-15.

[17] Brooks, R.A. 1986. A Robust Layered Control System for a Mobile Robot. *IEEE Journal of Robotics and Automation* RA-2(1): 14-23.

[18] Bryson, J. 1999. Hierarchy and Sequence vs. Full Parallelism in Action Selection. submitted to *Cognitive Science* 99.

[19] Chapman, D. 1992. Intermediate Vision: Architecture, Implementation and Use. *Cognitive Science* 16: 491-537.

[20] Chapman, D. 1987. Planning for Conjunctive Goals. *Artificial Intelligence* 32: 333-377.

[21] Cohen, C. and Voss, F.V. 1992. A Comprehensive Study of Three Object Triangulation. *Proceedings of SPIE Mobile Robots VII* vol. 1831: 95-106.

[22] Connell, J.H. 1992. SSS: A Hybrid Architecture Applied to Robot Navigation. *IEEE International Conference on Robotics and Automation*: 2719-2724.

[23] Dennis, J.B. 1980. Data Flow Supercomputers. *IEEE Computer* 13(1): 48-56.

[24] Duda, R.O. and Hart, P.E. 1972. Use of the Hough Transformation to Detect Lines and Curves in Pictures. *Communications of the ACM*, 15(1): 11-15.

[25] Feldman, J.A. 1985. Four Frames Suffice: A Provisional Model of Vision and Space. *Behavioral and Brain Sciences* 8: 265-289.

[26] Ferguson, I.A. 1992. Touring Machines: An Architecture for Dynamic, Rational, Mobile Agents. Ph.D. Dissertation. Computer Science Department. University of Cambridge.

[27] Firby, R.J. 1989. Adaptive Execution in Complex Dynamic Worlds. Ph.D. Dissertation. Computer Science Department. Yale University.

[28] Firby, R.J. 1987. An Investigation into Reactive Planning in Complex Domains. *AAAI-87:* 202-206.

[29] Gat, E. 1992. Integrated Planning and Reacting in a Heterogeneous Asynchronous Architecture for Controlling Real-World Robots. *AAAI-92*: 809-815.

[30] Gat, E. 1991. ALFA: A Language for Programming Reactive Robotic Control Systems. *IEEE International Conference on Robotics and Automation*: 1116-1121.

[31] Guiard, Y. and Ferrand, T. 1995. Asymmetry in Bimanual Skills. Manual Asymmetries in Motor Performance. Elliot, D. and Roy, A. Eds. CRC Press. Boca Raton, FL.

[32] Horswill, I. 1997. Visual Architecture and Cognitive Architectures. *Journal of Exper-*

imental and Theoretical Artificial Intelligence 9: 277-292.

[33] Horswill, I. 1997. Real-time Control of Attention and Behavior in a Logical Framework. *First International Conference on Autonomous Agents*: 130-137.

[34] Horswill, I. 1993. Polly: A Vision-Based Artificial Agent. *AAAI-93*: 824-829.

[35] Horswill, I. 1993. Specialization of Perceptual Processes. Ph.D. Dissertation. Department of Electrical Engineering and Computer Science. Massachusetts Institute of Technology.

[36] Huber, E. 1994. Object Tracking with Stereo Vision. *AIAA/NASA Conference on Intelligent Robots in Factory, Service and Space - CIRFFSS '94*: 763-767.

[37] Huber, E. and Kortenkamp, D. 1995. Using Stereo Vision to Pursue Moving Agents with a Mobile Robot. *IEEE Conference on Robotics and Automation*: 2340-2346.

[38] Jones, J.L. and Flynn, A.M. 1993. *Mobile Robots: Inspiration to Implementation*, A.K. Peters, Wellesley, MA.

[39] Kirch, D. 1991. Today the Earwig, Tomorrow Man?. *Artificial Intelligence* 47: 161-184.

[40] Knuth, D.E. 1976. *Marriages Stables*. Les Presses de l'Université de Montréal.

[41] Kosaka, A. and Kak, A.C. 1992. *Fast Vision-Guided Mobile Robot Navigation Using Model-Based Reasoning and Prediction of Uncertainties*. CVGIP: Image Understanding 56(3): 271-329.

[42] Kuipers, B. and Byun, Y. 1991. A Robot Exploration and Mapping Strategy Based on a Semantic Hierarchy of Spatial Representation. *Robotics and Autonomous Systems* 8: 47-63.

[43] Kuipers, B. and Levitt, T.S. 1988. Navigation and Mapping in Large-Scale Space. *AI Magazine* 9(2): 25-43.

[44] Kuipers, B. 1978. Modeling Spatial Knowledge. *Cognitive Science* 2: 129-153.

[45] Kuniyoshi, Y., Riekki, J., Ishii, M., Rougeaux, S., Nobuyuki, K., Sakane, S. and Kakikura, M. 1994. Vision-Based Behaviors for Multi-Robot Cooperation. *IROS-94*: 925-932.

[46] Laird, J.E., Yager, S.E., Hucka, M. and Tuck, C.M. 1991. Robo-Soar: An Integration of External Interaction, Planning, and Learning in Soar. *Robotics and Autonomous Systems* 8: 113-129.

[47] Lesser, V.R. and Erman, L.D. 1977. A Retrospective View of the HEARSAY-II Architecture. *IJCAI-77*: 790-800.

[48] Maes, P. 1990. Situated Agents Can Have Goals. *Robotics and Autonomous Systems* 6: 49-70.

[49] Marr, D. 1982. *Vision*. W.H. Freeman and Company. San Francisco, CA.

[50] Mataric, M.J. 1992. Integration of Representation into Goal-Driven Behavior-Based Robots. *IEEE Transactions on Robotics and Automation* 8(3): 304-312.

[51] McCarthy, J.M. and Hayes, P.J. 1969. Some philosophical problems from the standpoint of artificial intelligence. In Readings in Artificial Intelligence. Tioga, Palo Alto, CA, 1981, 431-450.

[52] McDermott, D. A Reactive Plan Language. Research Report YALEU/DCS/RR-864. Yale University. 1991.

[53] Miller, D. 1985. A Spatial Representation System for Mobile Robots. *IEEE International Conference on Robotics and Automation*:122-127.

[54] Moravec, H.P. 1996. Robot Spatial Perception by Stereoscopic Vision and 3D Evidence Grids. CMU Technical Report CMU-RI-TR-96-34.

[55] Moravec, H.P. and Elfes, A.E. 1985. High Resolution Maps for Wide Angle Sonar. *IEEE International Conference on Robotics and Automation*: 116-121.

[56] Musliner, D.J., Durfee, E.H. and Shin, K.G. 1993. CIRCA: A Cooperative Intelligent Real-Time Control Architecture. *IEEE Transactions on Systems, Man and Cybernetics* 23(6): 1561-1573.

[57] Natrajan, A. 1999. Consistency Maintenance in Concurrent Representations. Ph.D. Dissertation. Department of Computer Science. University of Virginia.

[58] Nilsson, N. 1984. Shakey the Robot. *Technical Note 323*. SRI International. Menlo Park, California.

[59] Noreils, F.R. and Chatila, R.G. 1995. Plan Execution Monitoring and Control Architecture for Mobile Robots. *IEEE Trans. on Robotics and Automation* 11(2): 255-266.

[60] Parnas, D.L. 1972. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM* 15(12): 1053-1057.

[61] Patalano, A.L. and Seifert, C.M. Opportunistic Planning: Being Reminded of Pending Goals. *Cognitive Psychology* 34(1): 1-36.

[62] Pearson, J.D., Huffman, S.B., Willis, M.B., Laird, J.E., and Jones, R.M. 1993. A Symbolic Solution to Intelligent Real-Time Control. *Robotics and Autonomous Systems* 11: 279-291.

[63] Pylyshyn, Z.W. and Storm, R.W. 1988. Tracking Multiple Independent Targets: Evidence for a Parallel Tracking Mechanism. *Spatial Vision* 3(3): 179-197.

[64] Rich, E. and Knight, K. 1991. *Artificial Intelligence*. McGraw-Hill. New York, NY.

[65] Riekki, J. and Kuniyoshi, Y. 1995. Architecture for Vision-Based Purposive Behaviors. *IROS-95*: 82-89.

[66] Sacerdoti, E.D. 1974. Planning in a Hierarchy of Abstraction Spaces. Artificial Intelligence 5(2): 115-135.

[67] Scheifler, R.W. and Gettys J. 1986. The X Window System. *ACM Transactions on Graphics* 5(2): 79-109.

[68] Sethi, Ravi. 1989. *Programming Languages*. Addison Wesley. Reading, MA.

[69] Simmons, R. 1990. An Architecture for Coordinating Planning, Sensing and Action. *Proceedings of the Workshop on Innovative Approaches to Planning, Scheduling and Control*: 292-297.

[70] Slack, M.G. 1992. Sequencing Formally Defined Reactions for Robotic Activity: Integrating Raps and Gapps. *Proceedings of SPIE Conference on Sensor Fusion*.

[71] Spector, L. and Hendler, J. 1992. Planning and Reacting Across Supervenient Levels of Representation. *International Journal of Intelligent and Cooperative Information Systems* 1(3 & 4): 411-449.

[72] Swain, M.J.; and Ballard, D.H. 1991. Color Indexing. *International Journal of Computer Vision* 7(1): 11-32.

[73] Terzopoulos, D. and Rabie, T.F. 1995. Animat Vision: Active Vision in Artificial Animals. *ICCV-95*: 801-808.

[74] Tracz, W. 1995. Confessions of a Used Program Salesman: Institutionalizing Software Reuse. Addison Wesley. Reading, MA.

[75] Tsotsos, J.K. 1995. Behaviorist Intelligence and the Scaling Problem. *Artificial Intelligence* 75: 135-160.

[76] Ullman, S. 1984. Visual Routines. *Cognition* 18: 97-159.

[77] Wasson, G., Kortenkamp, D. and Huber, E. 1999. Integrating Active Perception with An Autonomous Robot Architecture. *Robotics and Automation*.

[78] Wasson, G. and Martin, W. 1998. Multi-tiered Representation for Autonomous Agents. *SPIE Symposium on Intelligent Systems and Advanced Manufacturing*.

[79] Wasson, G., Huber, E. and Kortenkamp, D. A Behavior-Based Visual Architecture for Autonomous Robots. 1998. *CVPR Workshop on Perception for Mobile Agents*: 89-94.

[80] Wasson, G., Kortenkamp, D. and Huber, E. 1998. Integrating Active Perception with An Autonomous Robot Architecture. *2nd International Conference on Autonomous Agents*: 325-331.

[81] Wasson, G., Natrajan, A., Gunderson, J., Ferrer, G., Martin W. and Reynolds, P. 1998. Consistency Maintenance in Autonomous Agent Representations. Technical Report CS-98-06. Computer Science Department. University of Virginia.

[82] Wasson, G., Ferrer, G. and Martin, W. 1997. Systems for Perception, Action and Effective Representation. *FLAIRS-97*: 352-356.

[83] Wasson, G., Ferrer, G. and Martin, W. 1997. Perception, Action and Effective Representation in Multi-layered Systems. *GI/VI-97*: 73-80.

[84] Wirth, N. 1971. Program Development by Stepwise Refinement. *Communications of the ACM* 14(4): 221-227.

[85] Yantis, S. 1992. Multielement Visual Tracking: Attention and Perceptual Organization. *Cognitive Psychology* 24: 295-340.

[86] Zelkowitz, M., Shaw, A. and Gannon, D. 1979. Principles of Software Engineering and Design. Prentice Hall. Englewood Cliffs, NJ.

[87] Zilberstein, S. and Russell, S. 1996. Optimal Composition of Real-Time Systems. *Artificial Intelligence* 82 (1-2): 181-213.