An Architecture for Structured, Concurrent, Real-Time Action

by

Leon Rubin Barrett

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Jerome A. Feldman, Chair
Professor Dan Klein
Professor Thomas L. Griffiths
Professor Srini Narayanan

Spring 2010

An Architecture for Structured, Concurrent, Real-Time Action

Abstract

An Architecture for Structured, Concurrent, Real-Time Action

by

Leon Rubin Barrett

Doctor of Philosophy in Computer Science

University of California, Berkeley

Jerome A. Feldman, Chair

I present a computational architecture designed to capture certain properties essential to actions, including compositionality, concurrency, quick reactions, and resilience in the face of unexpected events. It uses a structured internal state model and complex inference about the environment to inform decision-making. The properties above are achieved by combining interacting procedural and probabilistic representations, so that the structure of actions is captured by Petri Nets, which are informed by, and affect, a model of the world represented as a Probabilistic Relational Model. I give both a theoretical analysis of the architecture and a demonstration of its use in a simulated robotic environment.

For my mother, who loved science and art, learning and doing, and above all, growth and life.

Contents

Acknowledgments

A great many people made my work possible and my life easier. First and foremost, Jerry Feldman and Srini Narayanan guided me through the maze of artificial intelligence research. They pointed out the paths of others before me and the directions where effort would be rewarded, saving me from disorientation and distraction. Without their knowledge and attention, I would have been quite lost. The rest of my committee has also been invaluable: working with Dan Klein gave me a wonderful opportunity to explore the data-driven world of statistical natural language processing, and I am grateful to Tom Griffiths for his particularly insightful comments as I drafted this document.

I have had the opportunity to work with many clever colleagues during my time here. I would particularly like to thank those with whom I have done joint research, namely Alberto Amengual, Slav Petrov, Romain Thibeaux, and Liam Mac Dermed. Also, the fellow students in my research group have provided key feedback that nudged my thoughts in interesting directions. Their numbers include Johno Bryant, Joe Makin, Nancy Chang, Steve Sinha, and Eva Mok.

I would like to acknowledge the undergraduate research assistants who have both written much of the code that realizes my designs and helped create the behaviors that rely on these tools. Andres Bonilla integrated the ATAN tool to make my software talk easily to the soccer simulator. Daniel Duckworth helped with RoboCup localization. Eric Hernandez aided in the localization, coded good support for observations, and helped design behaviors. Matthew Ekstrand-Abeug coded the essentials of math function support in the Petri net and provided substantial assistance in the creation of soccer behaviors.

My family, too, has earned my deepest gratitude: my father, who introduced me to the fun of computers; my brother and sister, who kept me challenged; and of course my mother, who showed me how to prioritize my work and still maintain a personal life. Also inspiring have been those muses who helped fill the non-research parts of my life: Kate, Megan, Caely, Todd, Cheryl, Alex, Ben, Juliet, Zach, Louis, Elena, Moorea, and more.

Finally, I proclaim my appreciation of my guinea pigs, who played the part of motivating examples whenever needed in a paper and provided boundless amounts of adorability at all times. Louis, Milo, and Chester, your contributions will live on.

Chapter 1

Introduction

The field of Artificial Intelligence often neglects two important issues: understanding the world works best when it is geared toward producing actions; and actions possess a great deal of structure. Consider humans: we are primarily environment-manipulating creatures, and we have good mental models for only the components we deal with. We tend to completely ignore aspects of the world that do not concern us, so we need not spend a lot of effort parsing and understanding them. (For instance, you probably have no idea how many times your shoelaces cross, even though you tighten them up every time you tie your shoes.) Similarly, we avoid rethinking our actions every time we perform them because we develop regular routines to exploit regularities, and we then use these routines as composable building blocks to make complex actions. (You would probably have to act out tying your shoes to figure out exactly what finger motions you make.) We thus neglect not only irrelevant stimuli but also irrelevant options and affordances. This use of routines streamlines both models and actions, keeping inference and planning from bogging down in irrelevancies. After all, knowing how you tie your shoes, and precisely what they look like, is not nearly so useful as just putting them on. Computerized methods for acting upon and understanding the world, then, will also benefit from taking advantage of these simplifying opportunities.

Continuing our example, think through the whole process of donning a pair of shoes. This action requires a number of components: for each shoe, slipping on the shoe, tightening the lace, and tying the lace. Each of these components can be further broken down into sub-components, eventually reaching some sort of "atomic actions," the smallest twitches a muscle can produce. Despite requiring structure, this action cannot be completely memorized, since the environment may be noticeably different each time (e.g. laces in different positions, laces slick with water, etc.). Furthermore, this sort of interaction with the world requires real-time responses to inferred

changes in the world, so that, for instance, we can recognize a sudden looseness as the sign of a slipped knot and respond appropriately.

This is precisely the sort of task that computers and robots find impossibly difficult. It cannot be scripted like an assembly job, or navigated with SLAM. The number of components makes reinforcement learning hard, and planning fails when it comes time to actually grasp a flexible shoelace. However, this is a simple procedure that humans master in less time than most Ph.D. careers take.

I propose that the difficulty in this task requires a way of dealing with the problems of action representation. After all, data representation is central to anything computers do. In order to use any algorithm or pore through any data, you must first design a data structure that matches the inherent structure of the data. Only then will algorithms be efficient, able to focus on the important relations in the data and ignore the chaff. The same need is present in the domain of producing actions.

In this dissertation, I describe a data structure and complementary set of algorithms that are focused on producing and understanding actions. This includes a procedural, compositional description focused on the intricate structural properties of actions, which is coupled to a probabilistic, inference-oriented model of the world. This compound model uses the strengths of both components to extend their reach into the complex realm of real-world action. Also, by focusing on the production of actions, this will lead our world models to be minimal—just enough to allow the agent to make useful actions. The result is an action framework that is quite capable of describing complex actions.

1.1 Representation Components

I combine two models in such a way as to enjoy their strengths and overcome their weaknesses. On the one hand, Bayes nets are an admirable model of the world, but they lack a structured action model. On the other hand, Petri nets (a sort of procedural model) can easily describe coordinated actions, but they do not easily represent the world. A sensible combination is to join Bayes nets with Petri nets. I will now explain in detail why this particular combination of formalisms is appropriate to represent real-world actions.

1.1.1 Bayes Nets

In the past few decades, Bayesian networks have become the standard representation for probabilistic and causal information [59]. In a Bayesian network, random variables are represented

as nodes, and dependencies between them are represented as directed links, so that the structure of a domain with randomness is depicted as a graph. Then, by also giving conditional distributions that say how the related variables affect each other, these graphical models can give detailed information about the total probability distribution, and probabilities about a single variable given other information may be extracted via **inference**. These models are described in more detail in §2.3.1.

By virtue of being graphical, the structure that Bayesian networks describe is easy for humans to understand. Also, because of their carefully-analyzed underlying mathematics, they are easy to manipulate via computational means. It is clear why these advantages make Bayes nets the standard representation and why I have chosen to use them. However, I use not mere Bayes nets, but the object-oriented extension, called Probabilistic Relational Models, which I describe more in §2.3.4. Thus, the world model used in this thesis consists of random variables attached to objects, and the objects may have various relationships, which affect the dependencies of the random variables. Based on these dependencies and on observed variables, it is possible to infer likely values for hidden variables in the world.

1.1.2 Petri Nets

The most basic procedural representation, the Finite State Machine (FSM), is simple, commonly used, and well understood; why not select it? The problem with FSMs is that they do not easily represent concurrent events. We wish to represent a world with several different components, each changing somewhat independently, but we must represent a single world state, which therefore grows exponentially. In a sense, all of computer science can be seen as a way to separate different sorts of states into independently analyzable chunks. So, we need a representation that inherently represents concurrent activity.

At the opposite extreme of representational complexity lies actual computer code. It is extremely flexible, and can easily separate world state into many different variables, while also representing concurrent execution of different tasks. The problem with such a powerful representation is that it is extremely complex and can vastly over-represent action. Analyzing complex representations becomes prohibitive or impossible; in fact, it is undecidable to determine whether an arbitrary program will eventually terminate. So, we would prefer to choose as simple a representation as possible,

Petri nets are essentially like concurrent FSMs [63]. They have markings in multiple places at once, and expand arcs by adding **transitions** to manage the coordination of those multiple markings. (They are described further in §2.2.1.) The result is that Petri nets lie in the

pleasant middle ground between the extremes of FSMs and computer code. They are inherently concurrent and split world state into weakly-interacting clusters. At the same time, some simple conditions allow Petri nets to be far more analyzable than computer code. Indeed, Petri nets have a long history of analysis, and there are standard algorithms for determining state reachability, deadlock, reversibility, coverability, and so on [58]. Therefore, Petri nets are an eminently suitable procedural representation for action.

1.1.3 Combination

Though these two separate tools have complementary features, it is not entirely obvious how they best fit together. Nor are they useful for our purposes entirely unaltered. Thus, in Chapter 2, we see the full details on how they can be extended and interfaced to produce a good representation of action.

1.2 Background

In order to completely understand the work presented here, it is useful to see how it fits into contemporary research in the general area. Here, I will attempt to present the shape of such other, related work.

- Planning

  Researchers have long tried to tease useful real-world activities out of computers and robots. Some of the first approaches were the most direct, and the most notable of these is **planning** [64, 26]. In planning, the world is described as a logical statement, with another logical statement as the goal state of the world. Similarly, every possible simple, **atomic** action is listed, along with the preconditions that must be true for it to be possible and the postconditions that will be true after it has been performed. Then, a search is made through the sequences of atomic actions to find a path from the source to the goal; once it is done, the robot executes this sequence. Unfortunately, this is most useful with actions that include very little uncertainty, such as package routing or airplane usage optimization. However, for real-world activities, this front-loaded decision-making can bog down an agent.

- Hierarchical planning

  Planning has been extended in a hierarchical fashion [52, 73, 74, 16]. The agent, rather than coming up with a sequence of atomic actions, instead finds a sequence of **high-level**

actions. Then, it breaks this high-level action down into lower-level actions, with just the same planning method. By doing this recursively, it eventually develops a full sequence of atomic actions that achieves the goal. However, because the search space at every level is smaller, and because later activities need not be fully planned-out until they are needed, such a hierarchical plan can be created and used much more efficiently.

- Reinforcement learning / MDP

  Another general framework for describing agents acting in a complex world is the Markov Decision Process (MDP). Here, the world has a state that randomly changes in discrete time, with each time step depending only upon the last state of the world and the agent's action (i.e. it has the Markov property). The agent gets scalar rewards from the environment, and it wants to figure out a **policy** that will show it how to act at every time step so as to maximize its expected reward. The process of solving an MDP is known as **reinforcement learning** [68, 41]. This is a well-explored field, with many extensions. For instance, one that I myself have worked on was to consider the case where the agent gets a vector of rewards; even then, reinforcement learning can find all possibly optimal policies [7].

  Unfortunately, reinforcement learning has a few major flaws for creating and describing complex actions in the real world. First of all, it learns policies as preferences for actions in every possible state (or, equivalently, as values of actions in every possible state). While ultimately this will approach optimality, the description of these preferences is often overwhelming in real-world environments, largely because it fails to take advantage of the many regularities found in the real world, where useful actions are often sequential or procedural, and optimality is not always essential. Also, reinforcement learning does not deal well with unobserved components of the environment, or with concurrent tasks. Thus, it is not highly effective at the sort of real-world actions that draw my attention.

- Hierarchical RL

  Just like planning, reinforcement learning can be extended with a hierarchy of increasingly complex actions. Again, the goal is to select policies at each level that are optimal (given the level they work at) and achieve the goal of obtaining as much reward as possible. Then, because the improved structure allows for simpler search, policies can be found for much more complex problems. Much of that work treats the structure as given by a human designer [49, 71, 50], while some actually attempts to learn that structure automatically [35, 36]. Not all of it works on concurrent actions, but notably [49] does. In principle, those methods could

give an interesting starting point for learning behaviors such as described in this dissertation.

- POMDPs

  Since the MDP framework assumes that the agent observes the entire environment, it is unrealistic, so it has been expanded with the Partially Observable Markov Decision Process (POMDP) [42, 34]. Again, the world is described with a state that randomly changes in discrete time, with each time step depending only on the last step (i.e. it has the Markov property). However, the state is factored, and only some of the factors are observable by the agent. Then, the agent, whether or not it knows the probability distributions for the factors, or how they are connected, or even how many factors there are, must attempt to act optimally. This world description is used for a number of related tasks, including attempting to discover the hidden state factors and relations of the world, and also to achieve optimal control either knowing or guessing a model of the world. True optimal control, given the world model, is achieved by finding the maximum over hyperplanes, where each hyperplane is a value of the world under some uncertainty over what its exact state is.

- POMDP FSM learning

  One sub-field of POMDP research involves making procedural controllers that solve POMDPs. These controllers take the form of Finite State Machines with a fixed number of states, and it is possible to find the best n-state FSM that to solve a given POMDP. This can be done in various ways, including gradient ascent over Markov chains [54], stochastic gradient ascent [55], and direct search over policies [32]. It is a clever way to produce behaviors with internal state, as described in this dissertation; however, it does not produce controllers that take advantage of inference directly, and it is not clear how to extend it to handle concurrent, weakly-coupled environment components and tasks.

- Petri net synthesis

  It is also possible to synthesize Petri nets from given state graphs. If we know a Finite State Machine with labelled arcs, we can find an equivalent Petri net with transitions having those labels. That is, we can factor a given FSM into a Petri net, essentially by finding regions with matching arc labels and creating a place to describe each such region [22, 13].

- Subsumption architecture

  Another related method is known as the **subsumption architecture**, pioneered by Rodney Brooks [11, 12]. Its core idea is that robots should be made to produce reactions without

worrying a lot about creating an internal representation of the world. Actions are instead produced by a stateful, reactive system. The agent contains a number of such modules, interacting together, and together they produce the agent's behavior. Each subsystem is a Finite State Machine designed to accomplish a particular goal, and the system is debugged after each is added in turn.

That clearly shows some similarities to the method described here. After all, both are focused on using reactive, stateful, concurrent modules that can respond quickly. However, the subsumption architecture uses FSMs, which do not factor or handle concurrency very well. Also, this work deliberately ignores the possible advantages of representing the environment, which, while sometimes an unnecessary trap into which researchers can fall, can be critical to producing appropriate responses to noisy inputs.

- Pengi

  Agre and Chapman also worried that planning was too unwieldy to be effective in fast-acting situations, such as the video game *Pengo*. Instead, they designed a reactive system that behaved appropriately without using a world model [2]. This rule-based system attached labels to objects in the world, and these were then used to choose actions. However, it was not truly without a world model; instead, the world was assumed to be fully observed, so it was not necessary to track the world state through time. The result was a system with similar goals and concepts, but by working in a simpler environment, it neglected much of the complexity that I tackle here.

- Action metaphor

  Unsurprisingly, there is related work from my own advisors. In particular, Srini Narayanan studied how computers could model the complex ways humans use metaphors about actions. He used a Petri net to model the concurrent structure of a behavior, which was linked via a metaphor map to a dynamic Bayes net that represented some other domain. This allowed him to take a parsed sentence, simulate the action described, and transfer the result through the metaphor map to the target domain [60]. This provided interesting and effective understanding of linguistic metaphors. However, because his behavior model was never required to actually take real inputs or produce real outputs, it was not finely detailed enough to do more than just describe the general structure of an action. That said, much of the system I describe here is inspired by his work.

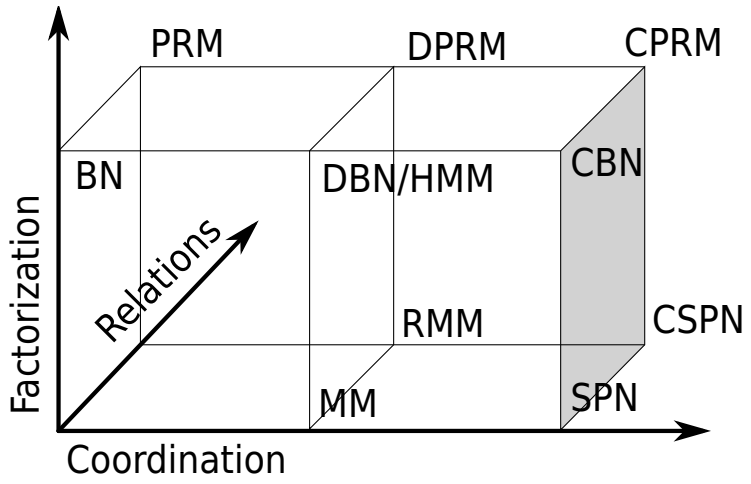


Figure 1.1: The meta-structure of probabilistic models on three different axes.

1.3 Coordinated Probabilistic Relational Models

There is another perspective that can be taken on the representation used here. I have said some related work, and described its features, but we can also envision how it and a number of related tools fit together with features on three axes. First, the full state of the world can be split into different, partially dependent chunks; we call this **factorization**. Second, the world has a temporal structure, which can be modeled coarsely or, in some cases, with finely structured concurrency and synchronization; this is called **coordination**. Finally, a model can use **relations** to describe the dependencies between chunks.

Of course, there are other dimensions by which we could categorize models, such as stochasticity and discreteness. However, we will focus on the three above axes, and simply take for granted the other axes, such as the stochasticity of our models and the models' flexibility with respect to discreteness. These three axes then comprise a 3D space, in which simpler models lie closer to the origin. Many combinations of these three features have been explored, and in Figure 1.1, we see the relationships between models with these different structures.

These three axes then make up a 3D space, where simpler models are closer to the origin. Many combinations of these three features have been explored. In Figure 1.1, we see the relationships between models with these different structures.

The simplest model of all is one consisting of a single world state, a single random variable. If we add factorization to this starting point, we get the Bayes net, which has several possibly-dependent state components [62]. If we add coordination to the start, we end up with a Markov model, where a single state evolves over time [10]. Of course, we can also combine these, and

obtain the dynamic Bayes net, with its many variables changing through time [59].

We can also add the third axis, relations, to any of those models. The relational form of the Bayes net is the Probabilistic Relational Model, where the state variables are lumped into "objects," which might be related [46]; these relations define dependencies between the variables of different objects. Similarly, a relational DBN is a dynamic PRM [65], which is a PRM with explicit temporal dependencies. The strangest is the relational Markov Model (RMM), which introduces relations not between variables (since there is only one state variable), but between the possible values of its state [5].

Each of the temporal models so far mentioned can still become more coordinated. As they are so far, they capture only the crudest sorts of temporal information: dependencies. Temporal processes in the real world can do such simple things, and operate dependently or concurrently, but in practice, most such processes interact in more interesting ways. They involve various parts that evolve sometimes alone, and sometimes in a coordinated way with other processes. They may produce resources that other processes use, or progress only when other processes are in certain states. Thus, the coordination axis continues further to the right, where we see models that capture some of these additional features. Models here are often based upon Stochastic Petri Nets, which extend Markov models by allowing such coordination within the evolution of the state [31]. They can be combined with Bayes nets to produce coordinated Bayes nets [61], or they can have tokens of various types, producing relational SPNs.

The model I present here combines the most complex parts of each axis. It allows the description, analysis, and modeling of related, factored, coordinated temporal processes. It can be seen in two ways: either it uses a Petri net to coordinate a Probabilistic Relational model, or it uses a PRM to add temporal, factored relationships to a Petri net, as described below. Because it is a coordinated form of a PRM, we may call it a CPRM. This representation is not the only possible structure that could lie on this vertex of the box in Figure 1.1, but it is interesting because of the way it extends existing representations into that space.

1.4 Example Domain: RoboCup

In order to show the usefulness of the representation I define, I apply it to a real task. I use the simulated RoboCup environment, in which simulated robots play soccer [43]. This is a commonly-studied problem designed to test a variety of robotic abilities. There have been a variety of RoboCup tasks, including a small, wheeled robot league, both real and simulated humanoid robot leagues, and the one I focus on: a simulated league. Here, the focus is not on

mechanics, vision, or robotic control issues, but is placed squarely on environment interaction and team play. Other than choosing actions and parameters, everything is taken care of, and the usual complexities of robotics are alleviated. To ensure that the focus is on creating sophisticated players that reason on their own, rather than simply being effective by unreasonably good coordination across the team, each player must be controlled by a separate process, and they can only attempt, without any guarantee of success, to send a few bytes at a time.

The basic format of the game is that of traditional soccer: 11 players per team, with a standard-sized field and reach for players. The world is two-dimensional, and the world is continuous in space but discrete in time. Time steps occur every 0.1 seconds, and in each, the state of the world is updated via stochastic difference equations. Each player can execute one action per time step, and that action modifies the state of the player or the ball. The agents are underpowered, so that with momentum they can gain speeds much faster than they can start or stop in a single time step. Furthermore, they have a slowly-regenerating reservoir of stamina, and if it is exhausted they may suffer long-term reductions in strength.

To make the game a higher-level task, the basic interfaces with the environment are also high-level. Instead of having to parse pixels into objects, or to effect changes by low-level joint angles, the players deal directly with the state of the world. That is, the player observes objects directly, and receives some (noisy) signal about the position and speed of objects. Similarly, the player outputs high-level actions like *turn* or *kick*, which have the semantics you would expect. This way, the designers have shortcut many of the complications of physical robotics and made it possible to dive directly into interesting behaviors.

So that the demonstrations below can be clearly grounded in this simulated soccer environment, I will now touch upon the finer structure of the environment. This description is based largely upon [14], and that is where you should look for an even more detailed look at the soccer simulator.

1.4.1 Simulator Detail

In this domain, the ball and players (11 per team) are represented as small circles whose position updates by a stochastic difference equation in discrete time. In particular, each of these objects has a two-dimension position p and velocity v. Every 0.1 seconds, these quantities update much like a Kalman filter would predict. The velocity decays by a small constant factor, and gains a small, proportional amount of uniformly-distributed noise. Then, the velocity is added to the current position to obtain a new position; no noise is added here. In the case of collisions, velocities are modified so that the two objects will not overlap. Note that this is not a perfect

physical simulation: in the case of very high speeds, objects can indeed pass through each other.

The players have some degree of control. In particular, during each time step, a player can emit only one action. These are atomic actions that have an effect only within a single time step. Unfortunately for the players, these actions are noisy—for every action, the actual parameter used is modified by an amount of noise distributed uniformly in a range proportional to the parameter's value.

A player can control its own speed by emitting a *dash* action, which has an associated parameter of *power*. This action is a discrete-time impulse, and it changes the player's speed by an amount proportional to the *power*. It accelerates (or decelerates) the player along the direction it currently faces.

The player may instead emit a *kick* action, which has two parameters: *power* and *direction*. This action changes the velocity of the ball, if the ball is within a defined radius. The actual change depends in a complex way upon the direction and distance from the player to the ball; in general, kicks directly in front of the player are the most effective.

The players also have an additional parameter, that of orientation. This comes with a restriction—their velocity can only be in the direction they face, so they cannot run in one direction and face in another. The players can emit the *turn* action (with parameter *moment*). This changes both the player's orientation and the direction of its velocity. However, again the actual amount of change depends in a complex way on the player's current speed; the faster a player moves, the less effective its turn actions.

In order to make the game more realistic (and prevent unreasonably frantic gameplay), the players use a complex stamina model. Each player has an amount of stamina that decreases when it accelerates and otherwise regenerates by a fixed amount in each time step. When the player's stamina runs low, not only is the effectiveness of the player's *dash* actions reduced temporarily, but it may suffer permanent reductions in both effectiveness and stamina regeneration rate. These factors are reset only at halftime—otherwise, the player's exhaustion is effectively permanent.

Goalies also get a special alternate action *catch*, with parameter *angle*. This has the expected semantics—if the ball is a small distance from the player, at roughly that angle, the goalie catches the ball. This is the goalie's main defense against scoring, as it is a more reliable way of stopping the ball than is kicking it.

The players have a detailed sensing model. Players see objects by receiving messages about the position, speed, and orientation of objects within their field of view. These messages identify the objects directly, thankfully obviating the need for any computer vision. The observed values are not modified by any noise, but instead corrupted by discretization errors, which increase

exponentially with the value of the parameter. Also, all positions and directions are given only relative to the player, so the player must factor in its own speed, location, and orientation in order to obtain global positions and speeds for the observed objects. However, this is of course more realistic, and it is almost always more useful. Similarly, to provide some realism, the player has a limited view region—objects outside of that region may be identified only by class (e.g. *teammate*, *ball*), or may be entirely unseen.

Finally, in one final gesture to physical realism, the player may turn its head relative to its body. This *turn_neck* command does not conflict with other actions as do *turn*ing, *dash*ing, *kick*ing, and *catch*ing, so the player can do it in the same time step as one of these others. This head direction changes only the player's field of view.

Each player also gets a certain amount of information about its own body. However, this is relatively limited. The only particularly useful elements are the player's available stamina, its neck angle, and a rough observation of its speed.

In order to let the player navigate in a general way, the player can see flags placed at various landmarks around the field. The sight model for the flags is just the same as that used for the players and ball. By combining the observations of the flags, it is possible for the player to have a rather detailed sense of its global position. Even better, the flagged landmarks include the goal boundaries and several nearby points, making the activities requiring the most attention to global position much easier.

So, this detailed, continuous-space discrete-time simulator is the backdrop for my demonstrations. It is a highly useful pre-built robotic simulated world, with an appropriate level of input and output.

It should be noted that my goal in demonstration is not to create a competitive soccer team. After all, this competition has been going on annually for more than a decade, and so hundreds of man-years have been spent, even on single teams (CMU, I'm looking at you!). Instead, I use this simulator to examine the effectiveness of my proposed representation for actions and behaviors, and this will not include a complete, soccer-playing team.

1.5 What to Expect in this Document

Because of the complexity of the action model under discussion, I will describe it incrementally. Therefore, to help maintain continuity and show how each piece adds important abilities, I will keep up with a series of examples. I will begin with the barest sketch of kicking a ball, and extend it both in detail and into more complex abilities, such as passing, including analysis, re-

covery from errors, and so on. Hopefully, this will help ground you, the reader, in what each piece means. Moreover, I will include a series of complete RoboCup behaviors that demonstrate the effectiveness of this framework.

The structure of this document is as follows: In Chapter 2, I give a detailed description of the components of my representation. I follow this with a look at analysis on this representation in Chapter 3. I continue by exploring how this model allows for recovery from unfortunate accidents in Chapter 4. I then summarize the results and implementation of this work in Chapter 5, and I conclude with a look back at everything and forward toward possible future work in Chapter 6.

Chapter 2

Representation of Action

Having motivated a full-featured architecture for action, it is time to say precisely what I propose and how it satisfies the demands placed upon it. The core idea of this representation is that we already have good tools for many of these pieces. In order to represent the structure of actions, we already have a number of tools of varying power. Similarly, there are already approaches that let us handle the unpredictable vagaries of the external world to greater or lesser effect. I have selected, modified, and combined two such tools that provide strong coverage of their own domains.

2.1 Conceptualization

Before jumping into the representation of action itself, let us first be sure to carefully frame the relevant concepts. Once I have shown what needs to be represented, it will be much easier to see why this representation is a good one. We must consider several factors, including components both internal and external to the agent doing the acting.

First, we are representing the behavior of an agent acting in an environment. The agent has control of some things in its world (for a soccer player, its body), and other things can be controlled only indirectly, if at all (its environment: the field and other players). Similarly, there are some things the agent observes directly (its senses), and others it cannot (most of the environment, and quite possibly much of its own body). Both control and perception are likely imperfect, so noise may be introduced in both input and output. A soccer player may not step exactly where it intended, nor are its eyes perfect cameras.

Also, both the agent's output (control) and input (perception) may have multiple channels. For instance, the agent's body will likely have multiple actuators, which we could view at the level of individual motors or muscles or at the level of muscle systems, such as limbs or muscle

groups. Similarly, the agent's inputs may include simultaneous visual, tactile, and auditory senses. This is one of the reasons that concurrent behaviors are so important.

Although the agent may only perceive a small fraction of it, the world's configuration may be very complicated. We call this configuration its **state**, and it may be broken down into different components. For instance, the state of a soccer game would include the positions and orientations of the players, their speeds, and so on. These components may interact, and it is precisely these interactions that make an environment interesting.

Of course, to interact, the state components must not be static, so we must deal with time. Unfortunately, dealing with time can be tricky, so we shall treat time as discrete. That is, time jumps in fixed quanta, and no events occur during these jumps, so we do not need to model anything with differential equations, but can instead use difference equations. That will improve the precision of the agent's internal models of the environment and also allow for more interesting analysis of its behaviors. However, though time is discrete, the environment need not be discrete in general. It may have discrete components, such as a light that is on or off, or another agent that is or is not aware of our agent's presence, but it may still have continuous quantities, such as the amount of liquid in a container or the position of our agent.

Similarly, the agent's inputs and outputs are divided into quantized chunks. The agent receives observations saying that a certain sensor has a particular value. In the same way, the agent's space of outputs consists of atomic actions that may have parameters. For instance, the agent could twitch a muscle with some force, move a motor some angle, or even kick a ball with a given force and direction, depending on the way the environment works.

Note that our agent need not live in a clockwork universe. Just because time ticks ahead in a regular manner, that does not imply that the world is predictable. Instead, we allow for randomness, not only in observations and actions, but also in the environment itself. The new state of each component of the world is chosen at random, in a way that depends on the state of the world at the previous time step. Of course, this also allows for deterministic evolution of the world, but it does not require it. This randomness means that our agent must choose its actions even though it does not know what will happen in the future.

So, with this setup, there are two main tasks for our agent. First, it must manipulate the environment to reach its goals. This will require taking certain actions at certain times, coordinating multiple actions, and reacting to what it learns about the environment. Second, it must infer the unobserved state of the environment from the observed state, which will allow it to react more appropriately to its observations. The representation presented here is designed to combine and extend tools for these separate tasks.

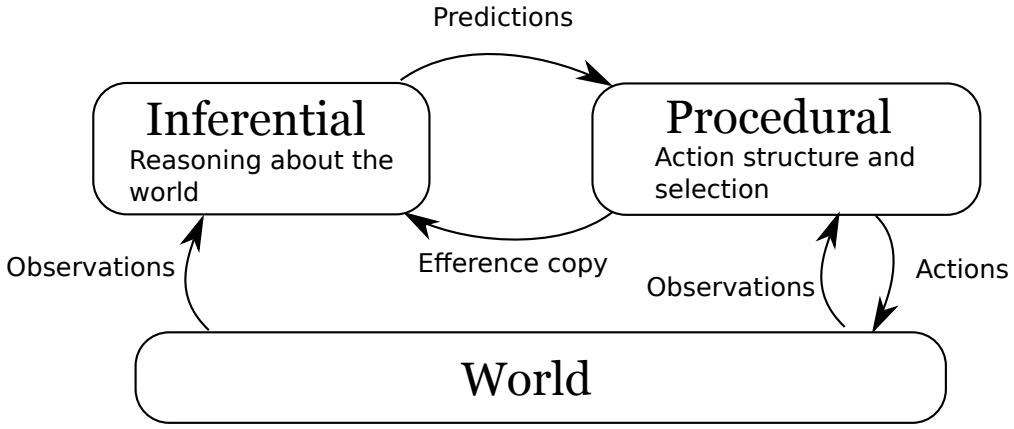


Figure 2.1: Here we see how the procedural and inferential models interact with each other and with the world in which they operate.

I use two components, one for each of these tasks. The first is a procedural model based on the Petri net, and it is good at representing and producing coordinated actions. The second is an inferential model called the Probabilistic Relational Model, which is superb at describing and inferring hidden information about the world. These two components interact with each other and with the world as shown in Figure 2.1. The procedural piece projects output on the world, basing its actions on the inferential estimates of world state and, for critically fast reactions, on direct world input. The inferential part maintains a model of the environment, updated by sensory input and feedback about action outputs from the procedural component. The workings of these pieces and their interactions are described in detail below.

2.2 Procedural Behavior

In order for our agent to deftly navigate and manipulate its environment, it must be able to execute the right actions at the right times. To describe such sequences of actions, we will use modified Petri nets.

2.2.1 Petri Nets

Petri nets are essentially extensions of Finite State Machines (FSMs) that, instead of keeping only one state token, allow more than one token at a time [63, 33]. Like FSMs, Petri nets also have circles (called **places**) that are used to indicate the state. Unlike FSMs, more than one of these places can have a token at a time; in fact, each place can have multiple tokens. So, the

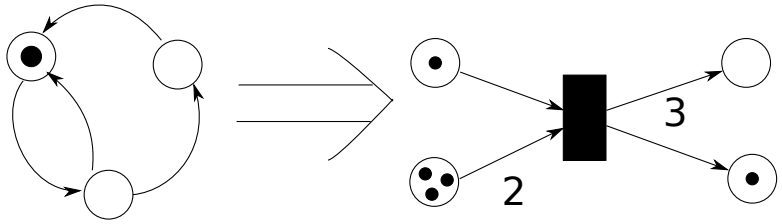


Figure 2.2: Finite state machine compared to Petri net

full state of a Petri net is a vector that lists the number of tokens in each place, and the meaning of the net is a combination of the semantics of the places and the number of tokens they hold.

Like FSMs, Petri nets require a way of moving tokens from one place to another. However, unlike FSMs, Petri nets have multiple tokens that should interact somehow, so we need more complicated semantics. Instead of just the arcs connecting states in FSMs, Petri nets have **transitions** that are connected to places. These transitions have **input arcs** (arrows pointed into the transition) and **output arcs** (arrows pointed out of the transition). The result is a bipartite directed graph, where transitions are connected only to places, and vice versa. Transitions can **fire**, and if they do, they consume tokens at their input places and produce tokens at their output places, in quantities given by the weights of the input and output arcs. The result is that tokens are created and destroyed; they need not be conserved. The tokens need not be affected one at a time, either; each arc can be labeled with a number saying how many tokens are created or destroyed when the transition fires. Of course, transitions that do not have enough available tokens to consume in their input places cannot fire and are called **disabled**; when they could fire, they are **enabled**. [58]

Historically, Petri nets were used to model asynchronous systems, which is done by assuming that any enabled transition may be fired by some external mechanism outside of our control. That is, any enabled transition might fire at any time, or might not fire at all. Then, the Petri net must be analyzed in a way that allows any enabled transition to fire, so rather than studying what does happen, we study what *might* happen. For instance, standard analysis includes state reachability, liveness, deadlock, and so on. For more information on Petri nets and their properties, see [58].

Let us now establish our running example. One of the most critical things a soccer-playing agent must do is to kick the ball, and it must do so in a variety of contexts. It must dribble, pass, shoot the ball, and steal the ball from an opponent. So, in Figure 2.3, we show the basic structure of a kicking action. As we describe the representation in more detail, we will refine this action.

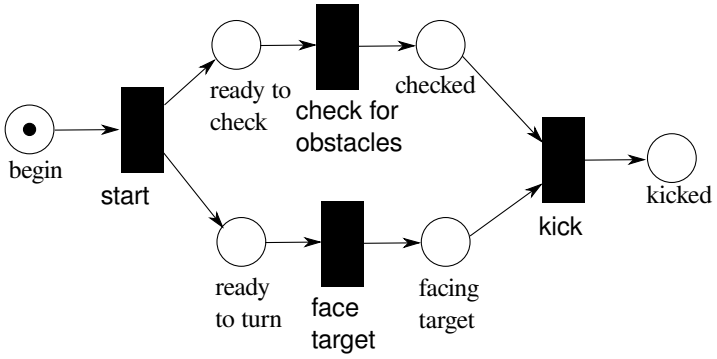


Figure 2.3: A simple example of a kicking event. Our agent will turn to face the target, make sure there are no obstacles, and only once those two concurrent tasks are complete will it actually kick the ball.

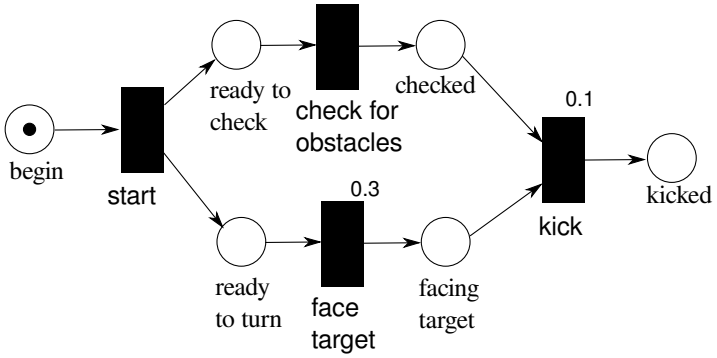


Figure 2.4: In our kicking example, the different parts of the action will take different amounts of time. For instance, kicking and looking are relatively fast, while turning is comparatively slow.

2.2.2 Time

We need our Petri nets to work in environments where time is critically important. Therefore, we use an extension for controlling the timing of transition firings, which includes **immediate** and **timed** transitions [70]. Immediate transitions always fire instantly, before any time has passed. Thus, all immediate transitions fire before any timed transition can fire. We do not, however, require the immediate transitions to fire in any particular order; instead, they are fired in an arbitrary order, so analysis in this case proceeds just as in the ordinary Petri net analysis. On the other hand, timed transitions fire after a fixed amount of time, which may be different for each transition. In Figure 2.4, we show how timed transitions fit into our kicking example.

See Figure 2.5 for a detailed example with an explanation of timing. This allows fine

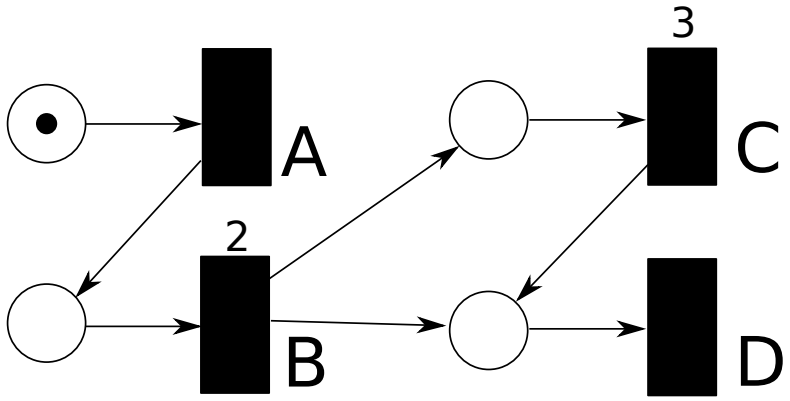


Figure 2.5: An example of timed and immediate transitions. At time 0, transition A will fire. Transition B will fire at time 2, enabling C and D. D will fire immediately, but C will fire 3 time units later, at time 5, followed immediately by D.

control over the timing of actions.

2.2.3 Control

The arbitrariness of transition firing in standard Petri nets can still be quite a confounding factor, particularly in cases where we care very much which particular transition fires. Therefore, we use standard extensions of the traditional Petri net definition, in particular the addition of some arc types that control transition enablement [23]. One such addition is the **test arc**, a modified arc going from a place to a transition that does not consume any tokens when the transition fires but still disables the transition whenever that input place does not have enough tokens. It is indicated by a dashed arrow. We also add a similar **inhibitory arc**, which disables the transition when the input place has more than a certain number of tokens. This is represented by a dashed line ending in a circle. (Note: a Petri net with inhibitory arcs and no limit on the number of possible tokens is Turing-complete, so we must be careful to place capacity limits on in the networks or analysis may become undecidable [23].) See Figure 2.6 for an example of these arcs.

In Figure 2.7, we show how this might apply to our running example of kicking. It allows us to make a clear decision whether to abort the kick or actually perform it based on the presence of an obstacle.

2.2.4 Input and Output

Of course, we know that the Petri net must interact with its environment. In particular, it must receive sensory information from the environment in a way that allows it to change its

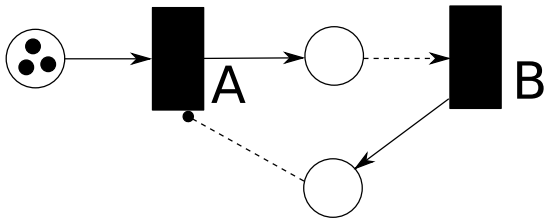


Figure 2.6: The dashed test arc means that transition B cannot fire until A fires; the dashed, circle-headed inhibitory arc means that once B fires, A cannot fire again, even if it has tokens remaining in its other input.

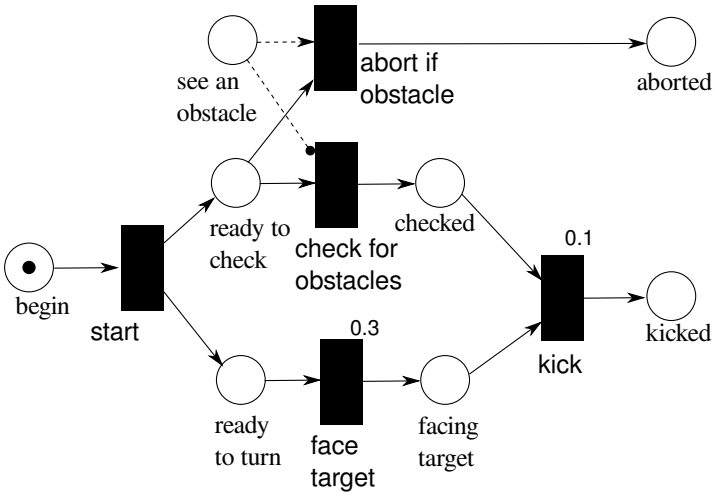


Figure 2.7: With this control scheme, we can make sure that our agent ends up in the right state based on whether it sees an obstacle in the kick path.

behavior, and it must be able to send commands to its body. Fortunately, the Petri net already has a way of changing its behavior—the markings of places determine what transitions fire. So, if observations from the world set the markings in some places, then they will affect the behavior. Similarly, we attach output events to the transitions of the Petri net.

Observations from the environment may be connected to places. An observation affects a place by setting how many tokens it has; this happens whenever the observation is received. For instance, see Figure 2.8: here one place receives an observation of an opponent's distance. This place controls two transitions; if the distance is high enough, the player will kick, and otherwise it will abort.

In the same manner, transitions may be connected to output messages. (This is similar to the way state changes in FSMs can be linked to emitting symbols, thus describing a language.) When the transition fires, then some atomic action command is sent to the agent's body, producing a result in the world. Depending on the level of detail of the model, this could be a very low-level action, such as a motor neuron firing, or a high-level one, such as a 'turn' command. Such a command may take parameters; if so, these are given by associated places. In Figure 2.8, we show how both inputs and outputs would be used in our kicking example.

2.2.5 Continuous Quantities and Mathematical Transformations

Because the agent's environment may include continuous quantities, our procedural model should be able to handle continuous inputs and produce continuous outputs, including making decisions based on these quantities. Therefore, we use **continuous places**, which are simply places that hold a real-valued quantity of tokens [17]. It is simple to see how these continuous places may take inputs from the environment and provide parameters to the atomic actions. However, using them as behavioral controls requires some care because we would rather not give up the analyzability of our Petri nets.

Therefore, we restrict the use of continuous places in the following way. Though they can be the source of test and inhibitory arcs, they cannot have tokens removed or added by transitions. We do, however, provide a mechanism, other than environment observations, to modify their values. We define **math transitions**, transitions which set their outputs to a function of their inputs. These math transitions make it possible to make decisions based on functions of several inputs. By making sure continuous quantities only affect the behaviors in carefully-defined ways, analysis does not need to consider their precise values, only whether they enable or disable transitions.

These new continuous places and math transitions are used in the running example in Figure 2.9.

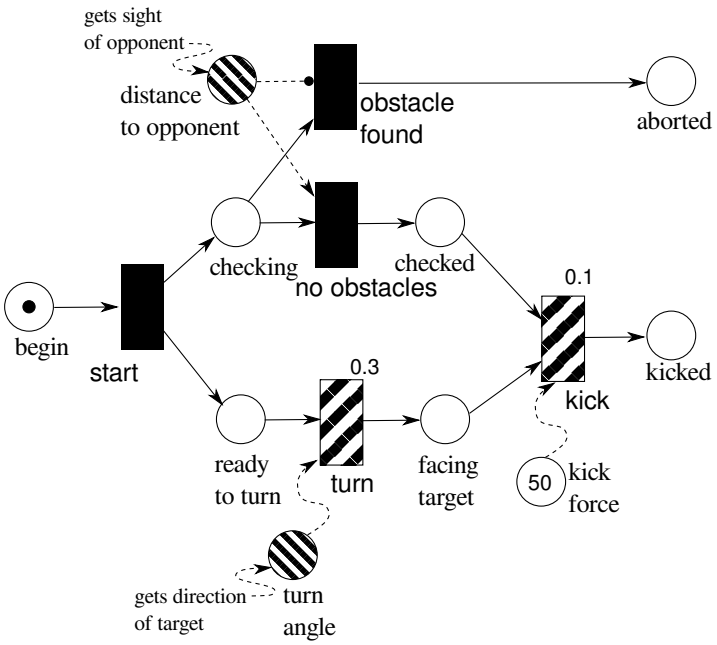


Figure 2.8: Here, we show how the kick network can receive inputs for seeing opponents and the direction of its target and send commands and parameters for the turn and kick actions. Inputs are shown as hashed places, and outputs are shown as hashed transitions, with parameters connected by wiggly dashed lines.

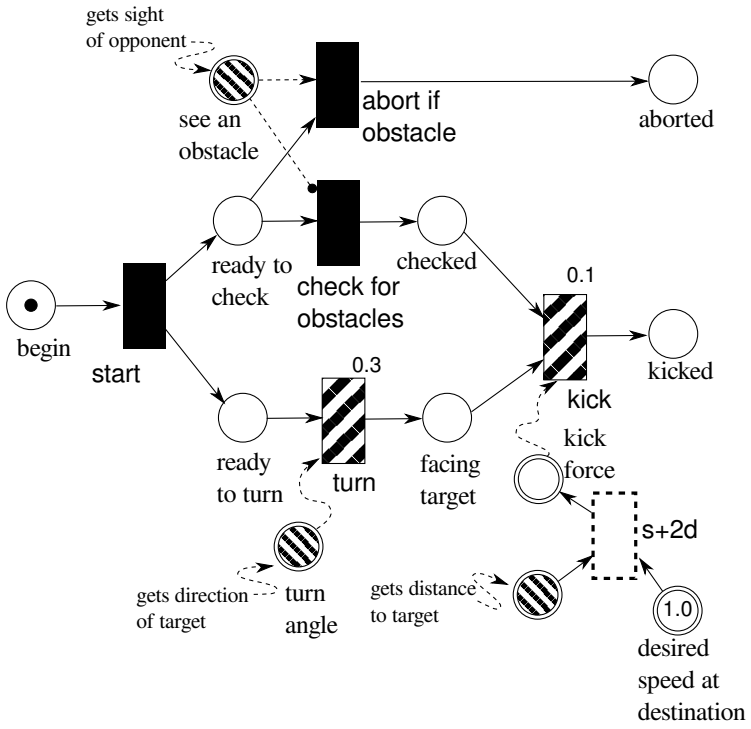


Figure 2.9: Here, we show how the kick network uses continuous inputs to make decisions and set its control outputs.

2.2.6 Modules

Consistent with the general computer science technique of "divide and conquer," a procedural representation that can be segmented will be easier to both design and analyze. For a Petri net, the simplest segmentation is just to label a set of places and transitions as being part of a sub-net; then, we can handle this sub-net separately. I will call such a sub-net a **module**. Note that in general, not all of the elements of a module are (directly) relevant to anything outside the module. Instead, some subset of its elements will form an interface with the rest of the full network. Then, when we design or analyze the larger net, we need only pay attention to the module's interface. Indeed, when representing the larger net, we can simply reference existing modules and then describe them separately. Of course, modules may be nested, so that one module contains and interacts with another module. Although it is not inconceivable that modules could reference each other in a recursive manner, so that a module contains a module with a reference to itself, I will generally proceed with the assumption that this is *not* the case, that instead the modules reference each other in a directed acyclic graph (DAG) structure.

This complete definition of modules gives us the ability to subdivide any existing Petri net. However, analysis is simpler on more restricted systems, and so we find it useful to define a restricted class of modules. For these modules, some elements of their interface have defined semantics. Let us start by defining **input** and **output** Places for the module. An input Place can only be modified by the outside network by adding tokens, not by removing them. An output Place can only be modified by the outside network by removing tokens, not by adding them. In particular, they have the following element semantics:

(Place) ***begin*** An input Place with capacity 1. This indicates that the module should start its activity.

(Place) ***ongoing*** An output place with capacity 1. When the module is operating, it leaves a token here, and does not remove it until it completes.

(Place) ***done*** An output Place with capacity 1. When the module has finished its processing, it will put a token here as it removes one from *ongoing*.

(Places) ***interrupts*** Input Places with capacity 1. Putting a token in one of these while the module is active indicates that there has been some sort of external problem and the module should take alternative action, possibly not ending in success.

(Places) ***errors*** Output places with capacity 1. If the module ends its activity without reaching

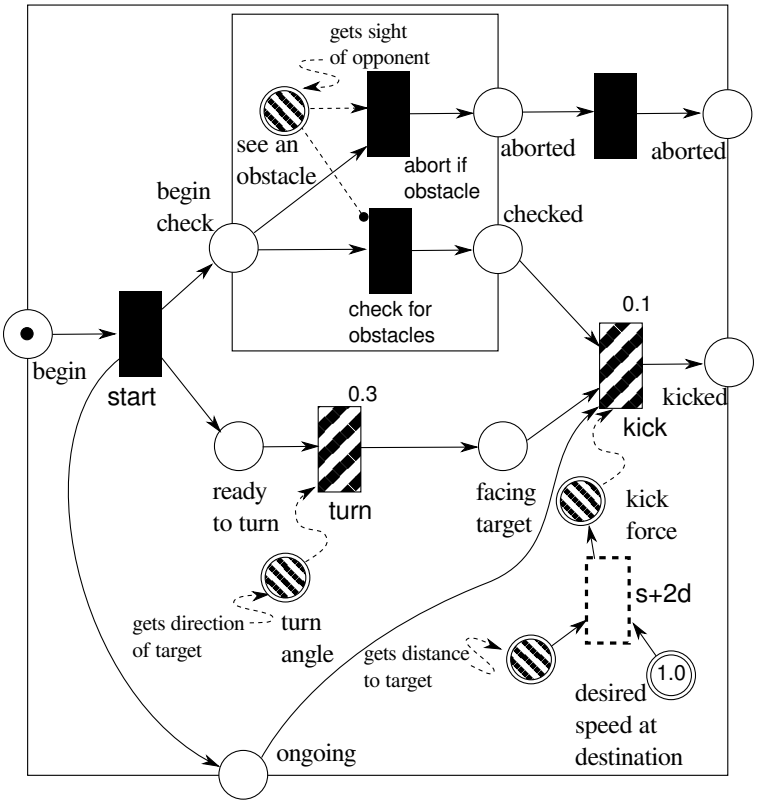


Figure 2.10: Here, we show how the kick network can be modularized. We extract a module that checks for obstacles, with one begin Place, one done Place, and one error Place. This way, the kick network can simply trust each component to work and use it the same way, even if the internals change. Similarly, we can describe an interface for the kick network that makes it a useful module for a higher-level behavior.

success, it will place a token in one of its error Places. Which Place receives a token indicates what sort of failure has occurred.

An example modularization of the *kick* network is shown in Figure 2.10. Of course, a module may provide more components than this on its interface, but defining these semantics means that analysis may be much simpler. For instance, we can determine whether a module may ever become active with traditional Petri net reachability analysis, or show that two modules will never be active at the same time by conservation analysis.

2.2.7 Demonstration: a RoboCup Behavior

Using only the reflexive, structured-procedure machinery we have so far described, not including the inferential reasoning yet to be discussed, we have enough tools to produce a useful behavior. With these enhanced Petri nets, we can create agents that coordinate hierarchical actions and choose their actions based on a combination of their past observations and their own goals. They cannot infer information about unobserved factors, but, as the effectiveness of simple life on our planet points out, that is not always necessary.

For instance, in this section I present a simple offensive soccer agent, made to function in the simulated soccer environment described in §1.4. This agent passes to its teammate, receives a return pass, dribbles toward the opposing goal, and takes a shot on the goal. The overall form of the behavior is shown in Figure 2.11.

First of all, this behavior relies upon the modularity of the procedural structure. The high-level module references many lower-level modules, each of which encapsulates some lower-level activity, and many of which are described below. For instance, the *kick_off* action begins by using a *setup* module to move into position before the game starts, and a *dribble* module to handle the various complexities of moving the player and the ball around the field. Then, the high-level behavior need only say how these components fit together. By using the lower-level pieces as building blocks, the overall action is much clearer in structure, making it easier to describe, analyze, and learn.

Furthermore, the high-level module counts on the structural abilities of our Petri net variant. It is organized both sequentially and hierarchically, with occasional loops. For instance, there is a primary sequence in which the player sets up, passes, runs forward, catches the ball, dribbles to the goal, and shoots. Each of these components is described as a separate lower-level module. However, in the case that the player loses track of the ball during catching or dribbling, it will loop back to finding and retrieving the ball. The clear structural mechanics of Petri nets make this possible.

We should also note that even the high-level behavior makes use of direct actions and observations of the environment. In particular, a precondition of the *go_to_ball* action is that the agent be facing generally toward the ball. Therefore, the high-level behavior will directly turn to find the ball before activating this action. This sort of turning could be encapsulated into a module, of course, but for simplicity and clarity of illustration, it is shown directly.

After the pre-game setup, the game begins with the player's kick-off pass. Here, the player passes the ball to a teammate, and the behavior is encapsulated into a *pass* module shown

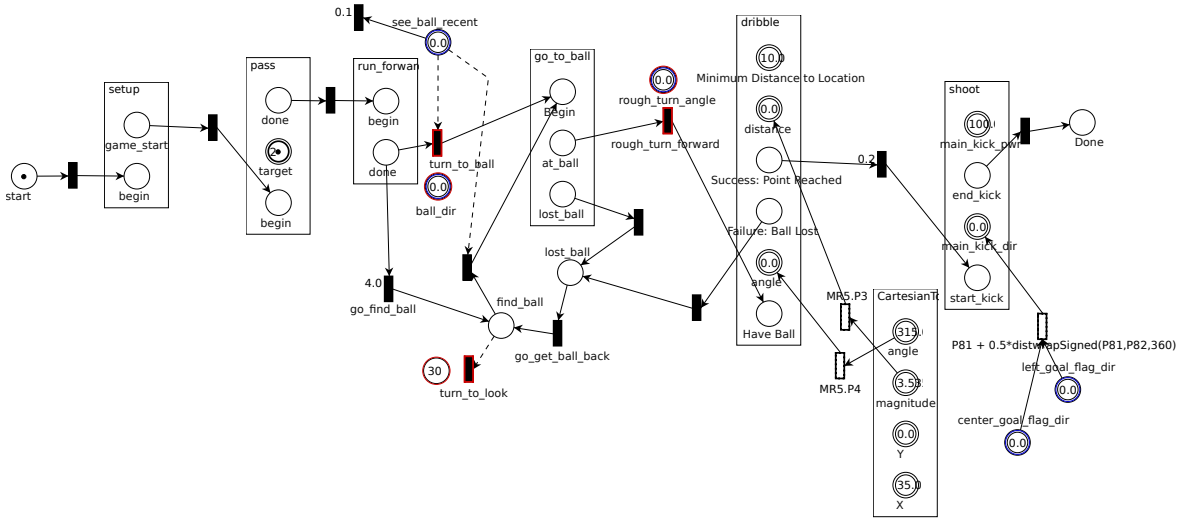


Figure 2.11: An offensive behavior that takes the player all the way from a kick-off pass to a shot on the goal.

in Figure 2.12. This module is highly linear; it takes a series of steps in order to achieve its goal. Note that the structure of the module reflects the demands of the environment. Some steps along this chain require confirmation before moving on, and some do not. In particular, the seeking portion of the behavior is not complete until the agent sees its target. On the other hand, the kick itself requires no confirmation of success. The agent can count on the environment in this way, so it can base its actions reliably on its previous actions.

A low-level behavior like this relies critically upon the direct input/output mechanisms of the Petri net extension. Those simple extensions to the standard Petri net formalism make this easy, and allow the agent to use those inputs both for its outputs and for determining the enablement of transitions. In order to determine the direction of a kick atomic action, the agent must base this output parameter on its inputs about the player's location, and it uses the *kick_dir* place to take this input and produce the appropriate output. Similarly, in order to decide whether it has found the ball and move on to further actions, it uses information from its senses via the *saw_teammate* place.

Furthermore, because the effectiveness of a kick action depends on both the angle and force parameters, the force the player uses must depend on the direction it aims. Therefore, it uses the mathematical extension to the Petri net to determine the appropriate value. It uses a math transition to set the value in *kick_pwr*, basing this calculation on both of the relevant quantities.

Note, however, that the agent's direct use of these noisy inputs is somewhat sub-optimal.

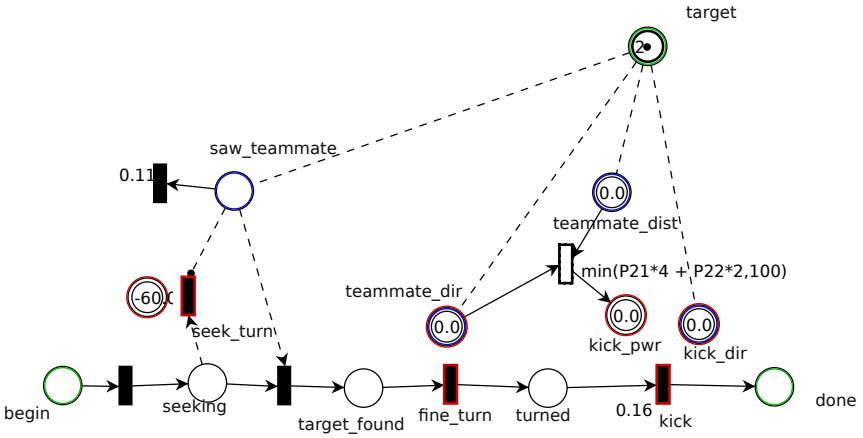


Figure 2.12: A module that performs a kick-off pass to a specific player.

Because the agent does not receive the facts about its teammate and the ball, but only an error-distorted signal, its output commands and parameters will be affected by those same errors. This distortion could be removed by inference that expects these errors and smooths the data to correct for them. A component to do this is described in §2.3.

In Figure 2.13, we see the module that navigates the player to the ball. It has one input place, *begin*, and two possible outcomes—one in which it has arrived at the ball, and one in which, for some reason (such as the actions of an opponent, or navigation errors because of noise) it has lost track of the ball. This module is used both in the high-level behavior and also in the lower-level *dribble* behavior, where the agent repeatedly kicks the ball and moves to keep control of it. It is very similar to the slightly simpler version analyzed in Appendix A.

This ball-seeking behavior, despite some complexity for reacting appropriately when the ball is lost, has a simple loop and conditional structure. Because of the restrictions of the RoboCup environment, in each time step, the agent can only accelerate or turn, not both. That means that at each time step, the player must choose which of these actions to take. Thus, the heart of this behavior is a loop that repeatedly iterates making this choice. Based on how closely it is facing the ball in state *begin_iteration*, the player either turns toward the ball more closely or accelerates toward the ball. If it turns, it turns toward the ball, with a small damping factor to prevent oscillation in the face of randomness. If it accelerates, the player chooses its speed by a formula depending on the ball's distance and speed, as well as how much stamina the player has available to spend on such activities. (Recall that if the player overexerts itself, it suffers substantial penalties.)

At the same time, there is a continuous monitoring process going on here. The player is not the only thing that might affect the ball, and should the player lose the ball, it must react

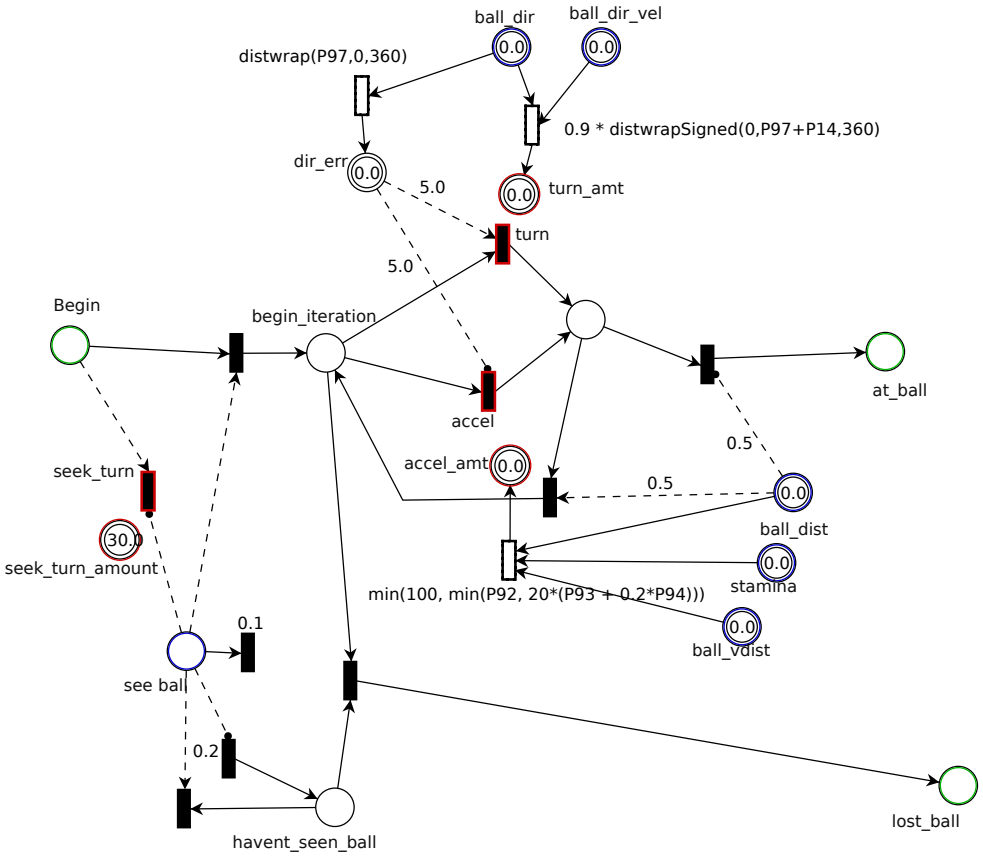


Figure 2.13: A reflexive behavior to go to a moving ball. It starts by looking for the ball if it cannot currently see it, then runs a main loop which repeatedly decides to turn to face the ball or to accelerate toward the ball. If it ever loses track of the ball, it goes to the *lost_ball* state, which is an output of the module, and thus passes the error to the higher-level module.

appropriately. What is appropriate depends on the circumstance, however, and such decisions are beyond the scope of this module. Thus, if the player ever loses sight of the ball for two consecutive time steps, long enough to believe there is something more than just a sensory error at fault, it aborts the main action loop and goes directly to the *lost_ball* state. Then, the higher-level behavior that is using this module can react as needed.

In Figure 2.14, we can see the middle-level behavior *dribble*. Its structure is primarily looping: it repeatedly kicks the ball toward its goal and then retrieves it. For the ball retrieval, it relies on the low-level module *go_to_ball* to accomplish this. It performs the kick itself, as it does this in a somewhat irregular way—it kicks the ball lightly in a particular direction, and this is not made into a separate module because it is highly particular to this one action.

Note that the *dribble* action is also continuously monitoring possession of the ball. If

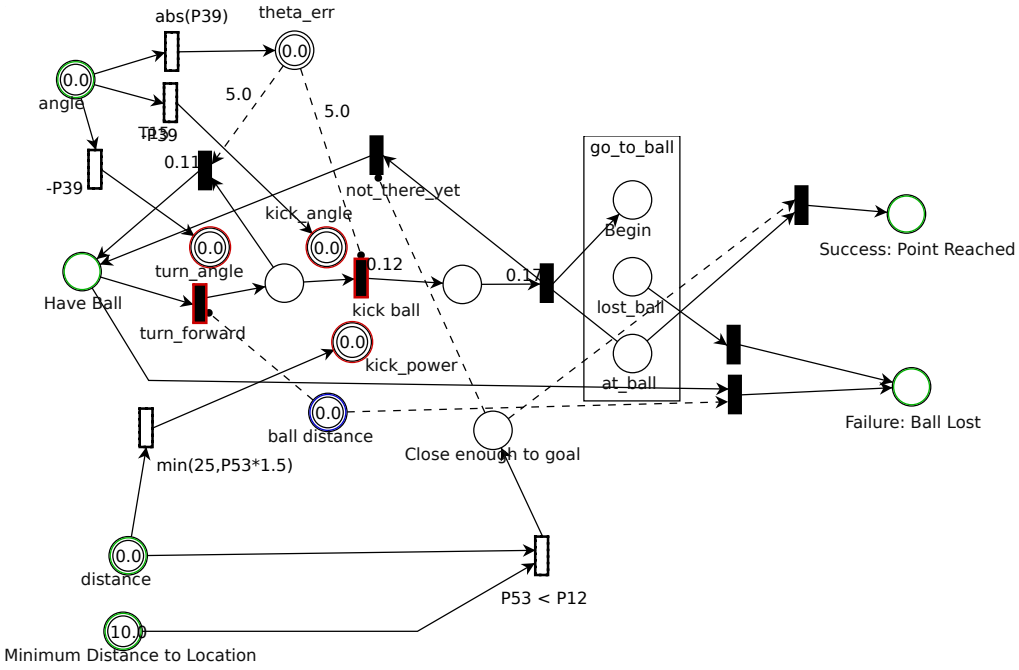


Figure 2.14: A reflexive behavior to dribble the ball to a destination.

it lets the ball get too far away, it has lost the ball, and presents this error to the higher-level behavior for appropriate action. In this case, it is supposed to let the ball get some distance away as it dribbles, so it does not check for this distance error just after it has kicked the ball.

This behavior is successful in its objective: given a simplified environment without opponents, it scores on the opposing goal. It is simpler than we might like because of its purely reflexive function. It does not do any inference on what is going on in the world, but, to the degree to which it can rely directly on its senses, this behavior is quite capable. It reliably goes to the opposing goal, shoots, and scores.

2.3 Inferential Reasoning

In addition to executing complex routines with interrelated components, agents must also cleverly interpret their environment and use that interpretation to inform actions. The procedural component we have discussed so far is far from ideal at that sort of reasoning. Instead, we will use probabilistic reasoning representations designed for precisely this sort of thing. The central representation is called a (dynamic) Probabilistic Relational Model (PRM) [28]. It is a generalization of the more standard Bayesian network (BN) [62]. To spare the casual reader from the vast ocean of literature on this topic, I will summarize the key points below. I will describe these tools

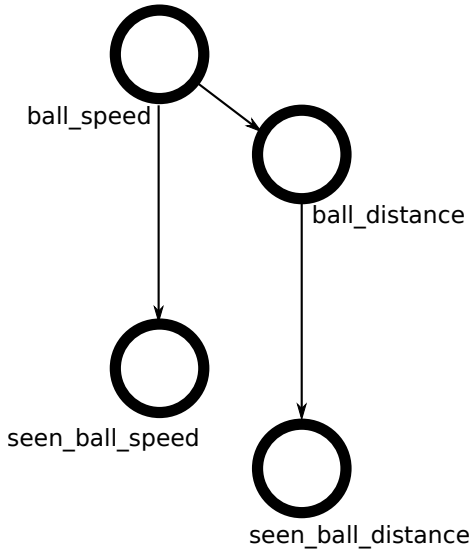


Figure 2.15: Here is a partial structure structure for the variable dependencies in a soccer world. The variable *ball_speed* is directly related to *ball_distance*, but not to *seen_ball_distance*.

starting with the simplest, so the explanations build on each other.

2.3.1 Bayesian Networks

Bayesian networks, also known as Bayes nets, are graphical (i.e. graph-based, not image-based) models of probabilistic data. The probabilistic idea is that the world contains **random variables**. Each random variable is a quantity that takes a value, but instead of that value being determined in some deterministic way, it is chosen randomly. The distribution from which the value is chosen may depend on the values of other variables. [62]

The graph part comes in as follows: Each variable is represented as a node in a graph. The nodes may be connected by directed edges, each of which represents a probabilistic dependency. So, a variable x with an edge going to variable y means that y depends on x. In other words, the distribution of y is affected by the value of x. Similarly, if there is no edge between x and y, there is no direct statistical dependence between them. An example structure for the soccer world is shown in Figure 2.15.

This graphical structure alone is not enough; we also require exact probabilities, which are given as conditional probability distributions (**CPDs**), like $P(y|x)$. Each variable has a distribution over its possible values given the values of its parent variables, where the parents are determined by the graphical structure. So, when one specifies a Bayes net, one must give both the (graphical) structure of the network, which indicates the dependencies, and the CPDs, which give

the precise distributions of the variables. These local conditional distributions, taken together, define a complete (**joint**) distribution over all the variables. That full distribution is given by repeated applications of the chain rule,

$$\mathrm{P}(x, y) = \mathrm{P}(y|x)\mathrm{P}(x)$$

So, by multiplying together all the conditional distributions for each variable given its parents, we can obtain a complete distribution.

For instance, in Figure 2.15, the full distribution is given:

$$\begin{aligned}&\mathrm{P}(\mathit{ball\_speed}, \mathit{seen\_ball\_speed}, \mathit{ball\_distance}, \mathit{seen\_ball\_distance})\\ &\quad = \mathrm{P}(\mathit{ball\_speed})\mathrm{P}(\mathit{seen\_ball\_speed}|\mathit{ball\_speed})\\ &\qquad \mathrm{P}(\mathit{ball\_distance}|\mathit{ball\_speed})\mathrm{P}(\mathit{seen\_ball\_distance}|\mathit{ball\_distance})\end{aligned}$$

We then eliminate the variables that are uninteresting by summing them out, so that we can get the distribution over just the variables that interest us. For instance, we may calculate $\mathrm{P}(y) = \sum_x \mathrm{P}(x, y) = \sum_x \mathrm{P}(y|x)\mathrm{P}(x)$. We might also do such inference to find the distribution of x given a known value of $y = y_i$: $\mathrm{P}(x|y = y_i) = \frac{\mathrm{P}(x, y=y_i)}{\mathrm{P}(y=y_i)}$ Of course, computing the full distribution can be quite time-consuming, and, unsurprisingly, there are algorithms for calculating these distributions without computing the full cross-product distribution. The two relevant methods are the Variable Elimination algorithm [75, 21] and the Junction Tree algorithm [38, 47]; the fundamentals of these methods are discussed later.

What we have considered so far is a way to extract information about how variables are likely to turn out in the world in general, but not in any specific cases. To extend the idea to specific cases, we add the concept that some variables can be **observed**. These variables have a known value; then, the idea is to find what values the other variables are likely to take, given what we have observed. In our soccer example, the observed variables will come from what the agent senses about the world. For instance, the agent sees something about the ball's distance and speed, though its sensors give noisy readings that are not always identical to the true value of these variables. Thus, in Figure 2.16, those observed variables are shown as shaded.

This representation is good for helping us model the agent's environment. The world is full of quantities that are related to each other. Furthermore, because these quantities are often related by very complex relationships, and often they depend on things the agent does not or cannot know, we are best off thinking of them as random. For instance, even macro-level effects like the bouncing of a ball, which are almost entirely deterministic, depend on tiny irregularities and starting conditions, so that a roulette ball seems random to us. Therefore, treating those

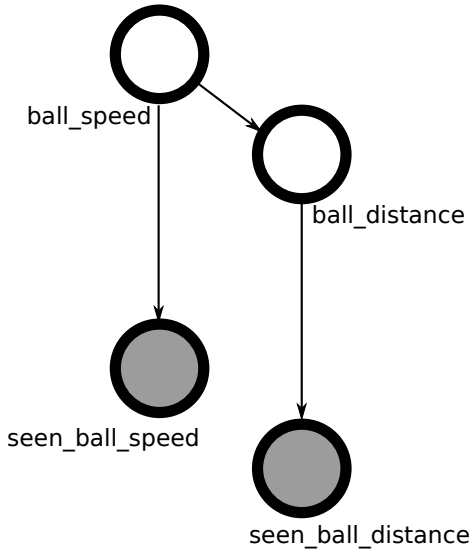


Figure 2.16: In standard Bayes net notation, observed variables are shown as shaded. Here, the soccer agent observes the variables *seen_ball_speed* and *seen_ball_distance*; based on this information, it can infer the distributions of the other variables.

quantities as random variables with relations provides leverage for inference. The agent observes some of these quantities, and can infer likely values for the others.

Connection to Petri Net

In order for this probabilistic component to be worthwhile here, it has to affect our agent's behaviors. It does our agent no good to compute a distribution over the position of the ball unless it can then use that to control the ball. Therefore, we need a way for the procedural Petri net to view the distributions of the random variables. The sensible way to do this is for it to see those distributions in its own terms. That is, to the Petri net, a random variable looks like a place, and its distribution looks like a number of tokens. That number of tokens may then determine whether transitions are enabled or not.

Specifically, the Petri net sees the influential random variables by including an equivalent place for each variable. The number of tokens in that place is given by a function of the variable's distribution. That is shown by a double-lined arrow from the variable to the place, as seen in Figure 2.17, though it may be elided in complex diagrams. Such functions might be the expectation, variance, argmax, or so on. The output of the function need not be a discrete number—after all, the Petri net can use continuous quantities for activating or disabling transitions. This inferred quantity may also be used as a parameter to actions, of course. This equivalent place cannot have

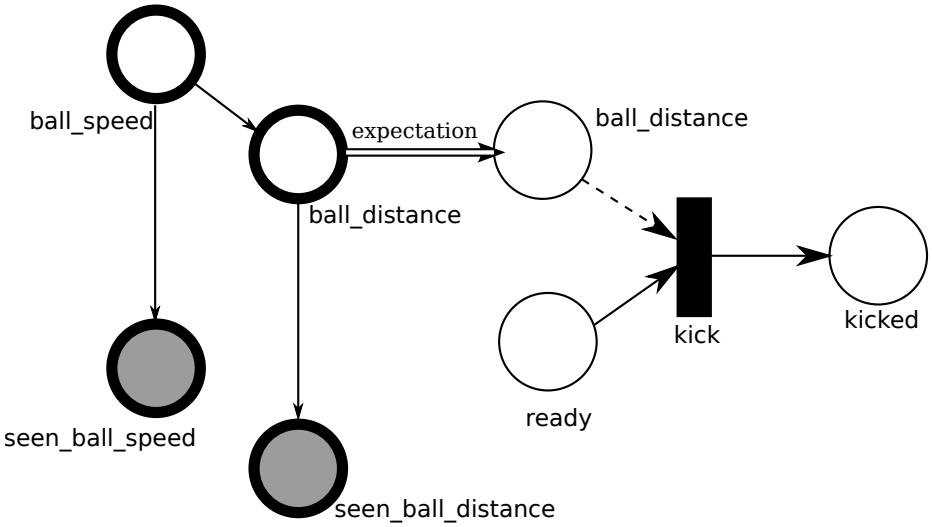


Figure 2.17: The *ball_distance* place has a token quantity equal to the expectation of the corresponding random variable.

its number of tokens set by the Petri net, so the only arcs affecting it must be test or inhibitory arcs, which do not modify their inputs.

For instance, in Figure 2.17, the Petri net sees the (inferred) expected distance to the ball. If that position is close enough for a kick to be effective, then the *kick* transition will be enabled, and the agent will kick the ball. If it is too distant, however, the agent will instead accelerate to pursue the ball.

Similarly, it is crucial for the inferential component to base its inference about the environment on the actions the agent itself has taken. Therefore, the inference needs to depend on the agent's procedural component. The way we connect them is to make the Bayes net view the Petri net in its own terms. Places have equivalent observed variables whose observed values are determined by the number of tokens. These observed variables have the usual conditional distributions; they may be causes or effects of other random variables. When inference is performed, then, these variables about the agent's internal state inform its knowledge of the hidden variables. Note that for this to be feasible with a discrete random variable, the place must have a limited capacity, so that its number of tokens is bounded to the number of values the variable could take.

For instance, in Figure 2.18, the Petri net's *kick* transition puts a token in the *kicked* place. This place corresponds to an observed variable in the Bayes net. That variable affects the distribution of the ball's speed. Therefore, if our agent has kicked the ball, it will factor that information into its estimate of the ball's speed.

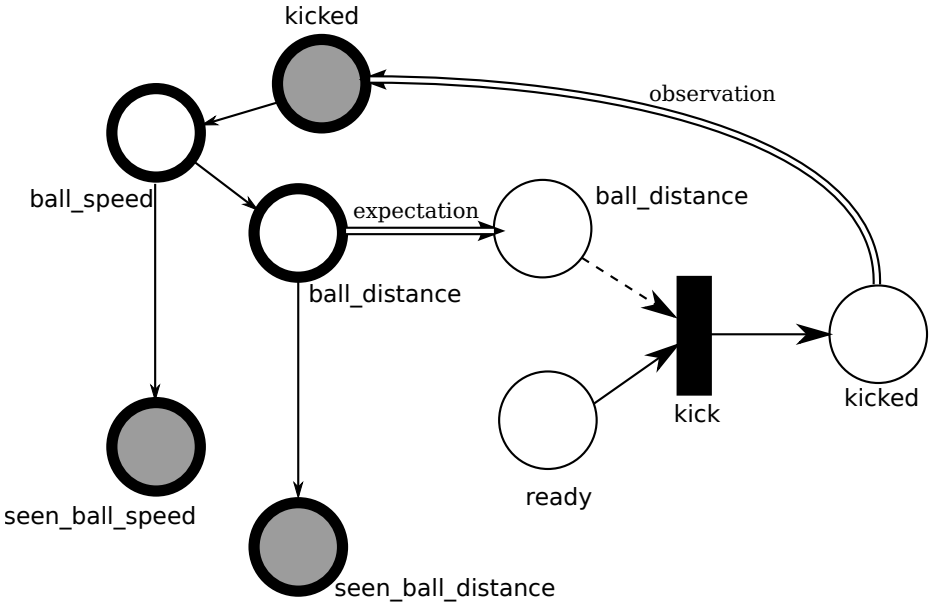


Figure 2.18: The *kicked* random variable corresponds to the equivalent place and helps to infer the ball's speed.

2.3.2 Bayesian Inference

In order to use inferred distributions, the agent must use some algorithm to calculate posterior distributions given its priors and observations. Here, we review two standard methods.

In the Variable Elimination algorithm, we remove one variable at a time [75, 21]. When we remove a variable, we must connect the variables that are related to it. For instance, if in Figure 2.19, we eliminate the variable w, that causes u, x, and y to become related, though it does not change the structure for z. The distributions for those newly-connected variables are computed by forming the joint distribution and then summing out w. If we do this for each uninteresting variable at a time, we will end up with just the variables we care about. The complexity of this method depends on the order of elimination. Some seemingly-simple Bayes nets can have exponential complexity because so many variables become related during elimination.

A related but more flexible method is the Junction Tree algorithm [38, 47]. This can give us the same sort of results as variable elimination, but for all variables simultaneously. Here, instead of eliminating the variables, we notice that a lot of the same information would be passed, no matter which variables we eliminate. So, we look at how the variables would be connected if we did eliminate them, to get the structure of that information, and then we compute that common information. Each cluster of connected variables is called a **clique**, and these cliques are connected

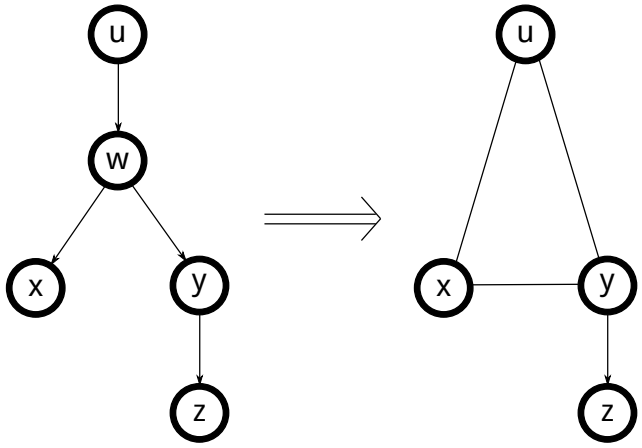


Figure 2.19: Eliminating variable w connects u, x, and y, but leaves z untouched.

according to the way the variables they contain are connected. This connectivity has the useful property that it can be acyclic—if two cliques are connected, their connection separates the clique tree in two pieces. So, we pass information between cliques, each summarizing everything going on its side of the tree. Then, with only a little extra effort, we can get the marginal distributions we want.

Note that both of these inference methods are exact, requiring discrete state variables and taking time exponential in clique size. There are other, inexact inference methods, such as particle filtering, which are very useful for estimating complex continuous distributions [6]. For environments of significant complexity, including the soccer domain discussed here, those methods are likely better. Not only do they make it easier to handle the continuous variables we may need, much better than discretization, but they also make it easier to manage the tradeoff of time complexity versus accuracy. For exact methods, a more complex model means that inference takes longer, so to make an agent that runs in realtime, it may be necessary to use a simplified model. This is means that these tradeoffs must be made when constructing the model. On the other hand, with particle filters, both the time and accuracy of the model depend on how many particles are sampled, which can be adjusted at any time. Thus, it is easier to make fast, appropriately-accurate inference systems with particle filtering.

Despite their advantages, approximate inference methods come with their own set of downsides. Most prominently, they require tuning to improve their accuracy. For instance, particle filters need to be designed to sample points in the right locations, and this may include the design of sampling distributions and rejection filters. Thus, although these methods would be a good match to a more practical implementation of the architecture described in this dissertation, this

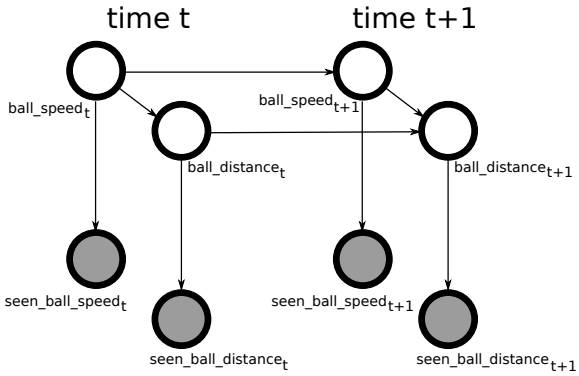


Figure 2.20: A simple dynamic Bayesian network.

proof-of-concept construction is better off done using the exact methods I have described here.

2.3.3 Temporal Inference

Since the world does not sit still, we expect the random variables that describe the world to change. This is a common case for Bayes nets to handle, and the standard solution is to view the world as a series of discrete time steps. That is, instead of just having a variable A, we have a series of variables A_t, where t is a time counter taking an integer value. All the variables tagged with t are in the same **time slice**. Then, the variables in different time slices depend on each other. For instance, in Figure 2.20, in every time slice t, the variable A_t depends on both its earlier value, A_{t-1}, and on the current value of B_t. These generic dependencies are defined with regards to the time variable t, then unwrapped into a series of variables, A_0, A_1, etc. This is not a new concept; it has been investigated in depth [59].

The key idea in inference is that we want to eliminate the variables from the past. We cannot afford to keep around all the past variables; otherwise, the inference complexity will depend on how long we have been dealing with our environment. So, at each time step t, we eliminate the variables at time $t-1$, encapsulating all our knowledge of the past in our inference about variables at time t—which is feasible because of the fact that the current time step separates the past from the future, also known as the Markov property. Then, we add new variables for time step $t+1$ and infer what we must about them from time step t.

We may do this inference with the junction tree algorithm. However, in doing so we must remember that the junction tree is constructed by finding what connections would be created by an elimination of all the variables. Because we want to eliminate variables from the past, then we find that this will connect variables in the future. In general, then, after eliminating many

past variables, there may be many variables in the present connected to each other. Exactly how they are connected depends on the structure of the graph in a slightly non-obvious way. Thus, the complexity of the temporal inference depends on the precise structure of the variables' relations. Of course, after iterating over a few time steps this process will converge, so that the set of cliques will be the same in every time step. Then, we do inference simply by computing distributions in all cliques and then, when moving forward in time, we simply shift the inferred distributions at time $t+1$ back to time t; then we can again compute distributions for the new $t+1$ slice.

Interaction with Petri Net

We now have two temporal components. First, the Petri net has delays that control how transitions fire over time. Second, the Bayesian network moves in steps of discrete time. How do these two concurrent components coordinate timing?

For the representation described here, these two components interlace in the simplest way possible. We define a length of time, t_B, that corresponds to the discrete time step of the Bayesian network. Then, with a period of t_B, the agent infers random variables' posterior distributions and shift the time slices back one step, eliminating the old variables and introducing new ones. This will change the numbers of tokens present in the variables' equivalent places, which may enable or disable transitions. Meanwhile, the Petri net, designed for concurrency, simply keeps concurrently firing transitions on their own schedule.

This contrasts with another sensible way to do inference: rather than having the Bayes net update with period t_B, we could instead link its update to the firing of a particular transition. That is an advantage because performing inference may be a lengthy task depending on the complexity of the variables' structure. By only doing complex inference when needed, we will spare possibly important resources. However, this is not highly effective with temporal Bayes nets. If our soccer player is tracking the ball using noisy sensors, it must update its estimate of the current location and speed frequently enough to smooth out the noise; it has no idea when the right time to do an update is. There are possible compromise methods, such as using different granularities of networks for frequent simple, temporal inferences and occasional deep inferences. However, we do not examine them here, as they are beyond the immediate scope of this project.

2.3.4 Objects

So far, we have not at all taken advantage of the fact that the world contains objects. Humans are able to infer and react very quickly in part because we use a simplified model of the

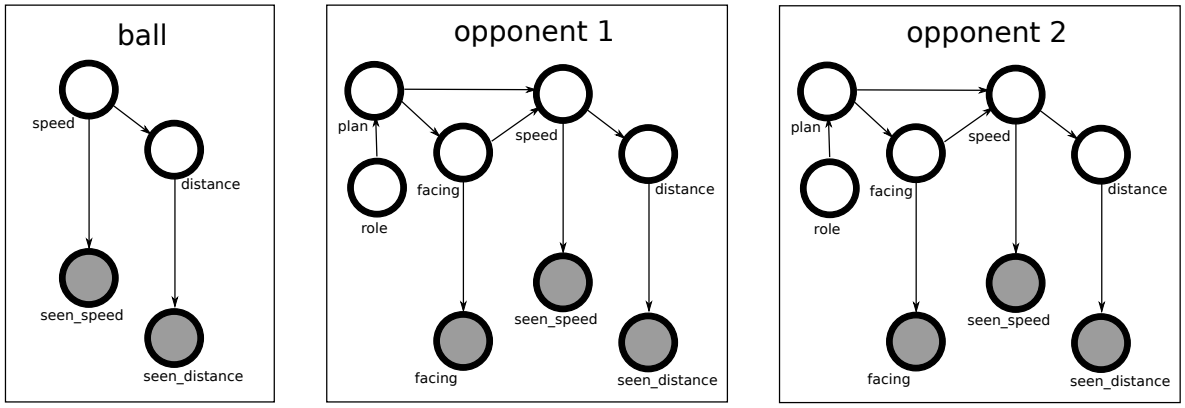


Figure 2.21: Some simple PRM objects.

world as much as possible. One example of that is that we link clusters of properties to particular objects. Objects are grouped into classes with the same sorts of properties. For instance, soccer players have positions and velocities, orientations and gaze directions. They also seem to have some hidden properties, such as roles, intentions, moods, and expectations. We need not consider the relations between a player's properties as separate for each player. Instead, the players have that in common; though their properties might take different values, the way they are related is the same. It would be good to capture that similarity in the probabilistic representation.

Fortunately, there is a way to represent objects with random variable properties. It is called a Probabilistic Relational Model [28], and the idea is very much as I have suggested so far. The universe contains objects, which themselves have properties, represented by random variables. The objects come in various types; objects of the same type have the same structure and conditional prior distributions for their variables. Thus, we can simply define the structure and distributions once, and they will apply to all the objects of that type. The objects' variables can be in time slices, just as ordinary variables would be [65]. For instance, in Figure 2.21, you can see several example objects and their (timed) random variables.

However, a universe of isolated objects would be rather sad and lonely. In our real universe, objects affect each other, and our representation of objects must be able to describe this as well. In the real world, not all objects affect each other—they only do so when they are related in some way. For instance, objects that are touching may push on each other, or a soccer player guarding another may move to stay near the player. So, the world also contains **relations** that describe how objects affect each other. If a set of objects is in a relation, then that relation can change how the variables are connected, including connecting variables in different objects and modifying their conditional distributions. For instance, in Figure 2.22, we see that the *possesses*

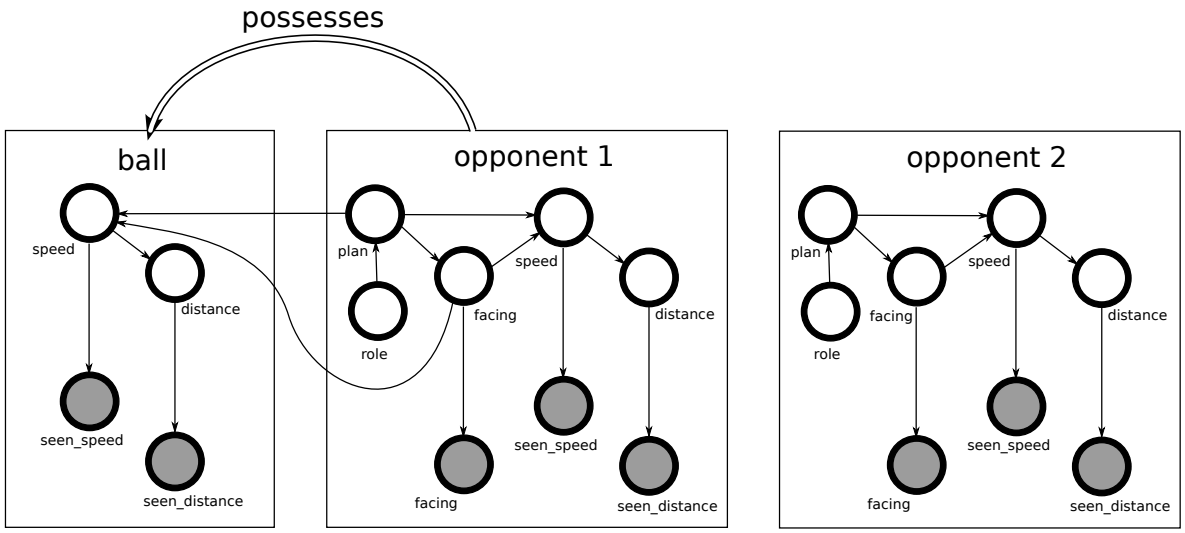


Figure 2.22: A simple PRM relation.

relation between a *player* and a *ball* makes the ball's position dependent on the player's position and orientation.

There are a number of approaches to working with PRMs. The one described here is a slight simplification of that given by [56]. That formalism allows for distributions not only over the variables of objects, but also over the number of objects in existence. It is also possible to do **lifted** inference with such a model, allowing the elimination of whole classes of objects at a time [19, 20, 57]. Alternately, we can eliminate the variables inside each object, so that then later inference can only depend on the object's interface [45].

We use a relatively simple method. We assume the world is closed, taking as given the full set of objects and hence the full set of random variables. So, we implement inference by **grounding** all the objects, creating a flat, conventional Bayes net corresponding exactly to the object-based description. The structure of the Bayes net is given by the structure within the objects and the active relations.

Even though we do not use the objects explicitly during inference, this use of objects is not merely a definitional aid. Yes, it does make it easier to define probabilistic networks. It may also help to learn them by capturing some of the important constants, making it easier to express complex relationships. However, it also allows for dynamic structure in the probabilistic model. Not only do the variables' values change over time, but the way they are linked may change over time. For instance, a relation such as *Possesses* may hold on *Player3* and *Ball1* at one point in time, but that may later change. This will change the way the variables in the Bayes net are

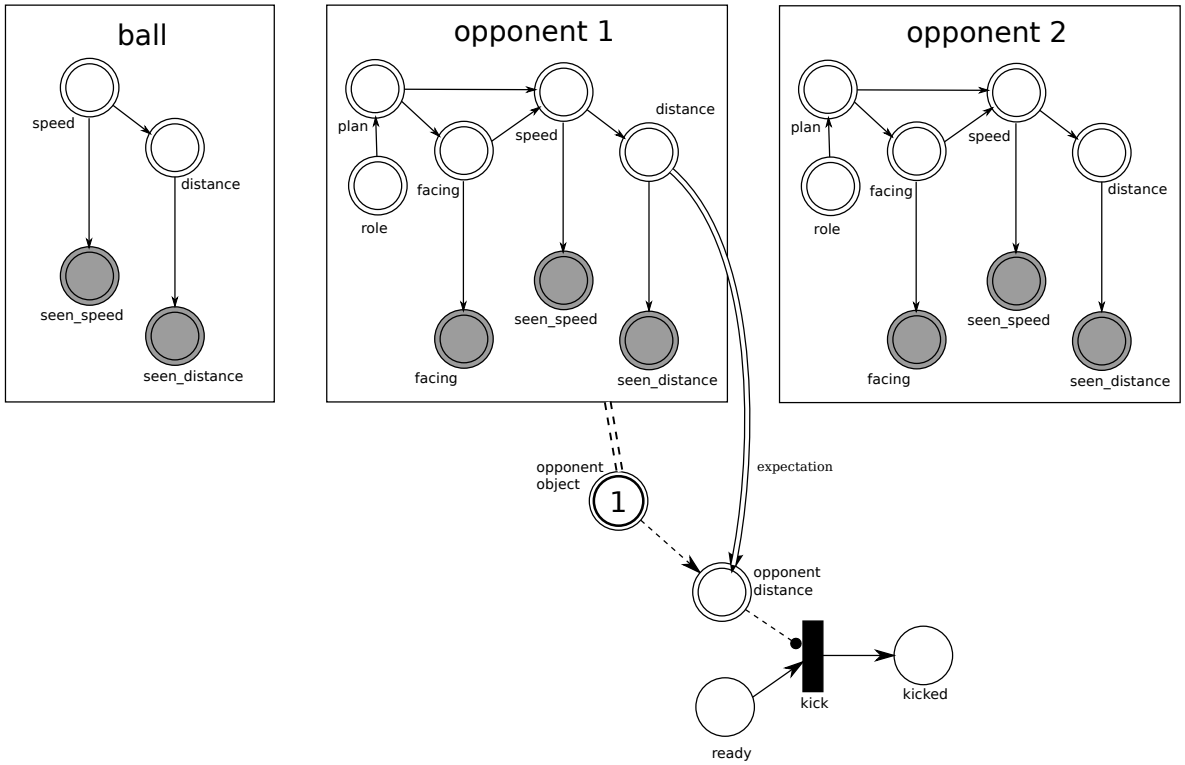


Figure 2.23: An object place (*opponent object*) makes the Petri net use the expected value of that particular opponent's distance to make its decision whether to kick the ball.

linked. This raises the question of how our agent knows which relations hold at any moment. That is explained below.

Objects and Petri nets

An activity in the Petri net may also be related to a particular object. For instance, the agent might want to pass to a particular player, to guard a player, to go toward a particular object, or so on. That is, it might not want to name ahead of time which variable would influence the behavior—instead, it might want a variable belonging to some object. Therefore, the Petri net needs to be able to pass around objects.

We allow this by defining **object places**, places which hold objects as tokens. For simplicity, each object place can hold only one object at a time. These objects are moved by transitions: if a transition that takes an object place as input fires, it puts that object in each of its output object places. (It would be possible to define more complex methods where multiple objects get moved around; however, that seems unnecessary and overly complex.)

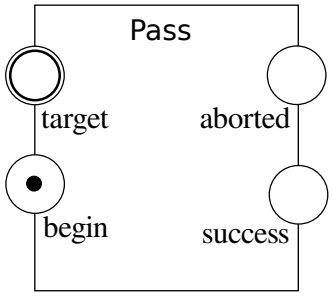


Figure 2.24: The external view of a Pass module that takes a target teammate object as a parameter.

Those object places are then linked to the Petri net's views of random variables, as shown in Figure 2.23. That means that the variables are variables belonging to objects—it is the copy of that variable that belongs to the object in question. Then, the agent can pass these objects as parameters to sub-networks, so that, as in Figure 2.24 for instance, the *Pass* network aims for the target's location.

Though we have described how to move object tokens around, how do these objects get into the Petri net in the first place? Object places can be given an expression to optimize. For instance, perhaps we want the agent to find the nearest opponent—then it could extract the object that minimizes the expression *opponent.distance*. Or, maybe we want it to find the second most open teammate, if one is sufficiently open; then, it would extract the object that has the second highest Distance(*teammate.position*, *opponent.position*), as long as it is at least 10 meters. In this way, objects can be pulled into the Petri net as object tokens.

Finally, our agent must determine what relations are active. We permit this by creating clusters of object places that together give the elements of a relation. If the object places are all filled, then the relation is active on those parameters; otherwise, it is not. This way, our agent can specify the active relations based on its activities. Note that this subsumes a purely Bayes net-based method, where, for instance, we say that the *Guarding* relation holds if Distance(*teammate.position*, *opponent.position*) < 4 meters. Because we can extract such related objects into the Petri net, we can use object tokens to make just that kind of relational definition, but we can also base the agent's relations on what it is doing.

This raises another use of relations—we can use relations to give varying grains of detail in probabilistic inference. Because it takes more resources to infer highly-linked variables, we want our agent in general to track unimportant objects with fewer links. This gives our agent a lower-quality estimation of their values, but it lets it focus its attention on the important objects.

Then, we want those important objects to have more densely-linked variables. Well, we can define relations such as *Attended*. For opponents which are not attended, then this relation does not hold, and the links present in *Attended* are not there. For an opponent the agent cares about, it can put that opponent in a relation element in the Petri net, causing the *Attended* relation to hold on it. Then, for this particular opponent, its variables will be linked at a finer grain, allowing our agent to spend more of its resources getting a better estimate of what this object is doing. Thus, we get improved control over what sort of inference is done.

2.3.5 Demonstration: A RoboCup Goalie

To demonstrate the use of the inferential component, and to show how all the extensions are useful, I now present another example soccer behavior: that of the goalie. Where the previous offensive behavior relied on a simple world without too much need for prediction or understanding of the surroundings, the goalie must predict the future in order to prevent the opponents from scoring goals. This is much more feasible with inference about the environment than with a behavior based purely on direct observations. Therefore, the goalie makes direct use of the inferential model.

The goalie has a probabilistic model for the world. For instance, it keeps particularly close tabs on the ball. The ball has a few observable measurements, in particular its *distance* and *direction*, plus the rates of change of these. These have precise values in the environment, but the agent only sees corrupted versions of them. They also evolve in a noisy way. Therefore, a probabilistic model like a Kalman filter is a highly appropriate model.

In Figure 2.25, we see dependency structure of the agent's model of the ball. This is a dynamic Bayes net with random variables for observed *distance* and *direction*, as well as their derivatives *vdist* and *vdir*, plus an observation of whether the ball was directly observed at all, *seen*. It also tracks two hidden variables, *dist* and *dir*, which are updated in time and hence have two temporal components, in time slices *t* and *t-1*. Each of these variables is discretized into about 10 different bins; the distributions between time steps are therefore somewhat complex. These hidden variables, which are smoothed out from the directly-observed noise, are then used in the goalie's Petri net component. This allows it to make decisions based on information which may not be directly observed, but predicted based on observations from times past. This illustrates the basic function of the inferential model.

Note that the network might be more accurate if it was more highly-connected. After all, the variable *seen* depends naturally on both *dist* and *dir*, as the visible region is a limited distance in front of the agent. However, because inference is performed exactly, via the Junction Tree algorithm, the time complexity of the inference is exponential in the size of the largest clique.

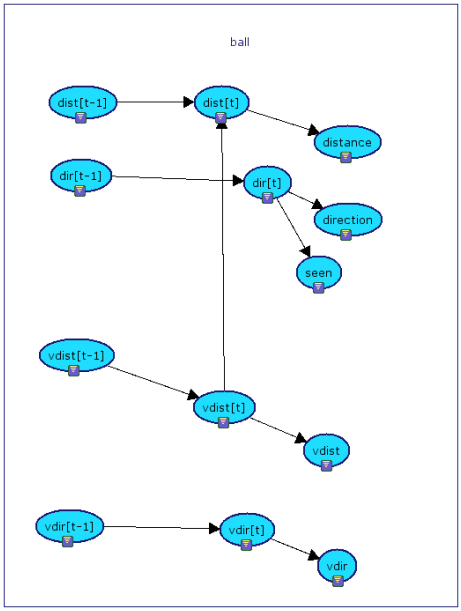


Figure 2.25: The probabilistic network for a ball object.

Therefore, it is important for making inference fast and efficient that we try to keep the variables in separate, unconnected components, so we avoid making both *dist[t]* and *dir[t]* parents of *seen*. This results in inference that is more useful because its results are more timely; however, the estimates of world values are necessarily less accurate, as is always a tradeoff when it comes to inference. For this reason, I suggest that future researchers investigate the effectiveness of particle filters in environments such as this, as they allow finer-grained control over the tradeoff between accuracy and computation time. However, for this first demonstration, exact inference makes a better proof-of-concept.

In Figure 2.26, the agent has a probabilistic model for opponents. These opponents contain similar attributes to the ball, such as distance, direction, and the rates of change of those. However, one thing to note is that there are many opponents on the field at one time. Therefore, the agent simply instantiates multiple instances of the opponent network. Without any relations to other objects, the agent does not connect each opponent's variables to those of other objects. Instead, by default the opponent's random variables are related as shown in the figure.

Note that, although it receives similar observations about all opponents as about the ball, the goalie does not care about these in quite the same way. Many opponents are not directly related to the goalie's immediate concerns, so the goalie need not track them in quite as refined a manner. Therefore, the probabilistic model for opponents in Figure 2.26 is noticeably simpler. Indeed, the agent does not even do temporal inference over the distance and direction to the player,

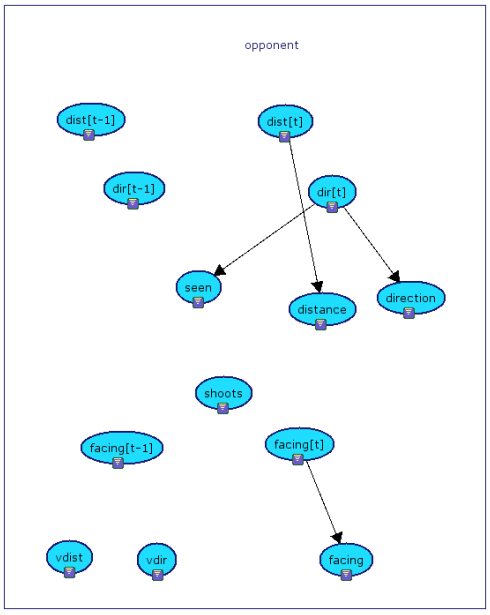


Figure 2.26: The probabilistic network for an opponent object. It is relatively unrefined, as the goalie does not receive a benefit from closely tracking presently-irrelevant opponents.

though it does use its priors to smooth the direct observations. This is similar to the way that we humans avoid tracking other people unless they are brought to our attention or relevant in some way. Once they gain this attentional focus, we can track them in great detail, but we will not do so until it seems useful.

To provide an attentional mechanism similar to that of humans, the agent can use a relation over objects. Consider the relation shown in Figure 2.27. This is a single-object relation, *attendedOpponent(Opponent)*, which is true only for the opponents that, for one reason or another, the goalie finds most relevant. Here, the goalie adds additional links between variables, imposing temporal inference on the variables *dist* and *dir*. Now, the agent carefully tracks and predicts these variables based on observations from previous times. Of course, this requires more computational resources. However, by carefully choosing which opponents it attends to, the goalie is able to spend more resources on those attended players than it could if it had to spread its resources equally. It thus tracks a few players more accurately, while sacrificing its estimation of the others.

Similarly, the agent has a lot to gain by making predictions based on the player that currently possesses the ball. Knowing what that player is likely to do will make it much easier for the goalie to make decisions on its future actions. Therefore, the goalie maintains another relation, this one *possessesOpponent(Opponent, Ball)*, describing an opponent possessing the ball. This relation is true if an opponent is sufficiently close to the ball to control its velocity. It allows

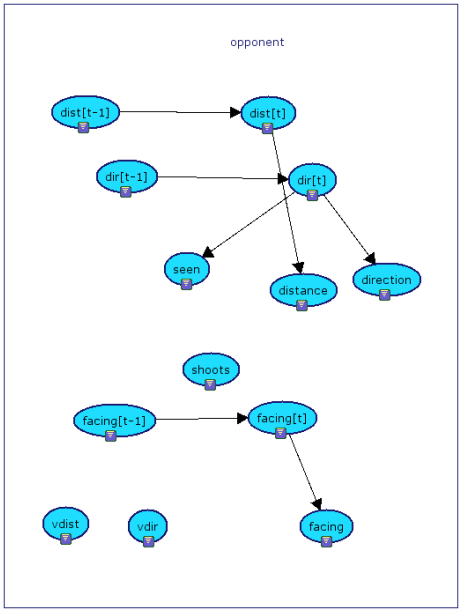


Figure 2.27: The probabilistic network for an *attendedOpponent* relation, which holds for those opponents to whom the agent is paying special attention. It is more connected and detailed than the ordinary opponent network, allowing the goalie to track such an opponent more effectively.

the player to use what it knows about the opponent to infer additional information about the ball's attributes.

In Figure 2.28, we see the structure of the goalie's probabilistic model for the *possessesOpponent* relation. Based on the opponent's distance and orientation, that opponent makes a decision on whether to shoot the ball; if it does, the ball will start moving quickly toward the goal. The precise details are given by the probability distributions over the variables, but the structure makes it clear what the general relationships are.

Given these inferential tools, the goalie is then able to react more appropriately than if it was basing its decisions on direct observation. In particular, in Figure 2.29, the goalie anticipates a shot on the goal, so it readies itself to block any such attempt. This module is largely linear, as the goalie has only one chance to actually use the *catch* action, which has a refractory period preventing overuse. The goalie therefore uses its estimates of the ball's position and speed to dash into the ball's way and precisely time its catch. It does not use the estimated values individually, of course; it instead derives quantities such as time to catch and expected point of closest approach, and then uses these highly relevant statistics to directly make its decisions. These derived quantities are used not only as test and inhibitory inputs to transitions, but also as parameters to output actions.

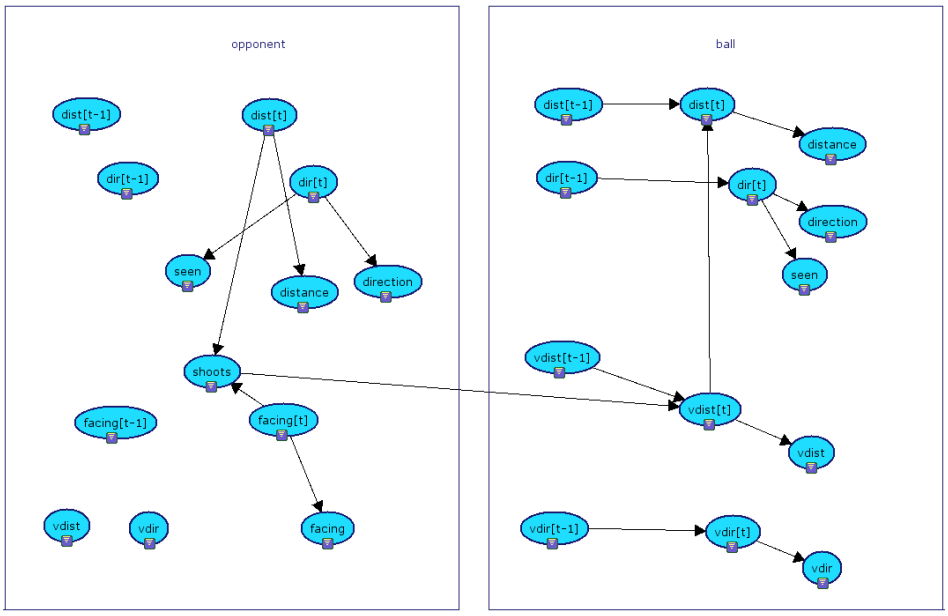


Figure 2.28: The probabilistic network for the *possessesOpponent* relation between an opponent and a ball object.

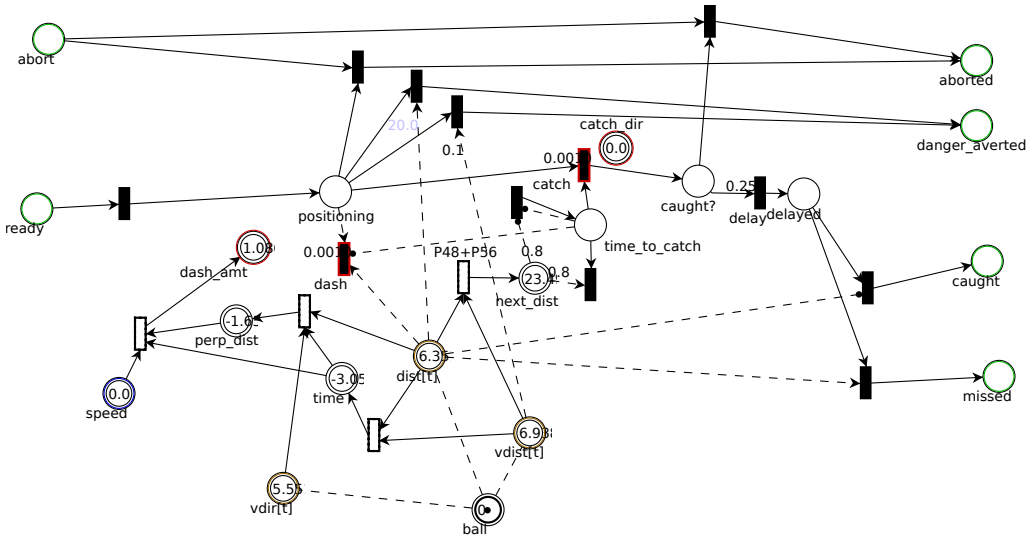


Figure 2.29: The goalie behavior for catching the ball.

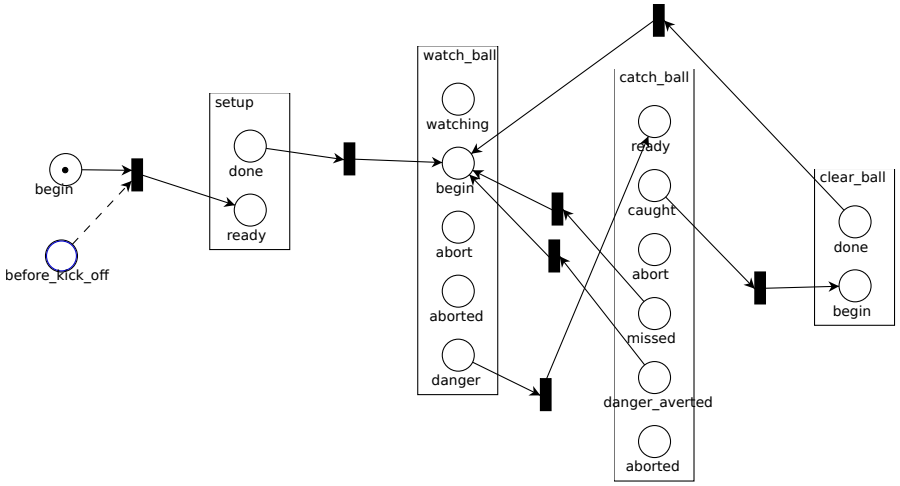


Figure 2.30: The high-level goalie behavior.

The overall behavior for the goalie is shown in Figure 2.30. This behavior is essentially cyclic, as befits the goalie's task. The goalie repeatedly catches the ball and clears it away from the goal, doing what it can to help its team. The finer detail is as follows: The goalie spends most of its time watching the ball; when it notices danger, it leaps into action to protect the goal. If that is successful, the goalie clears the ball and goes to a generally-useful goal-watching position and resumes its watchful waiting. Similarly, if the danger is resolved, or if the goalie fails to catch the ball but instead lets it past, the goalie resets its position again.

Finally, the goalie is effective. It is able to react appropriately and successfully stop shots against the goal it guards. By taking advantage of its environment estimation abilities, it goes to the location it needs to and does what it must to accomplish its task. Thus, we see that the goalie demonstrates the advanced probabilistic tools used in this representation.

2.4 Review

Having now examined all the pieces of this rather complex representation, let us recall them and consider their ultimate effect.

First, the procedural representation captures the structure of actions. By using a Petri net, it is able to describe how actions occur in sequence or in parallel. Breaking the Petri nets into modules allows for hierarchical structure, and linking them to the environment via inputs and outputs allows reflexive behaviors that react based on direct inputs.

Meanwhile, the inferential component allows reasoning to determine the course of our

agent's actions. Using cutting-edge first-order probabilistic tools, it takes ambiguous data and infers likely truths. It describes the structure of the world, including objects and their connections. It can focus its efforts on the most important objects.

Finally, these two components play nicely together. The system as a whole makes complex behaviors based on inferred information, with fast reflexes available in critical times. It can include the effects of its own actions on its guesses about the present, and it can conclude relationships about objects in the environment based on its observations and actions. All in all, this is a very capable structure for letting computers consider and produce actions.

Chapter 3

Analysis and Proofs

The largest motivation for representing procedures using Petri Nets, rather than something more expressive, such as arbitrary computer code, is that they permit analysis. We can examine a Petri Net and determine its execution properties, such as whether it may deadlock, without simulating every possible execution trace. In order to have confidence in behaviors we construct, we may want to analyze them to determine whether, or with what probability, they accomplish their goals. In particular, we might wish to analyze a behavior to determine whether (or with what probability) it will reach a particular goal, failsafe state, or error condition.

However, the main complication to any sort of analysis is that we cannot analyze any behavior in isolation—it will only function properly when coupled to the proper environment. Therefore, what analysis methods we may apply depend on the way we describe the environment, and their complexity depends on the difficulty of analyzing that environment. This is a well-known problem from control theory, as an exceedingly simple controller may control a complex system to achieve some goal, but proving this fact can be difficult or intractable depending on the system and goal [66].

Fortunately, the modular representation we use allows for decomposition of both behaviors and environment, which allows us to analyze sub-modules and then combine the resulting analyses to get an analysis of the complete agent. Those sub-systems may be analyzed either by further decomposition or by a variety of other methods:

- First, for high-level modules, we can use our analysis of the sub-modules to simplify our task. We need not examine every possible state, but instead we can use the modules' pre- and post-conditions to verify that the behavior will work as intended. §3.1
- For the general case, we can use a Markov model analysis method to determine detailed

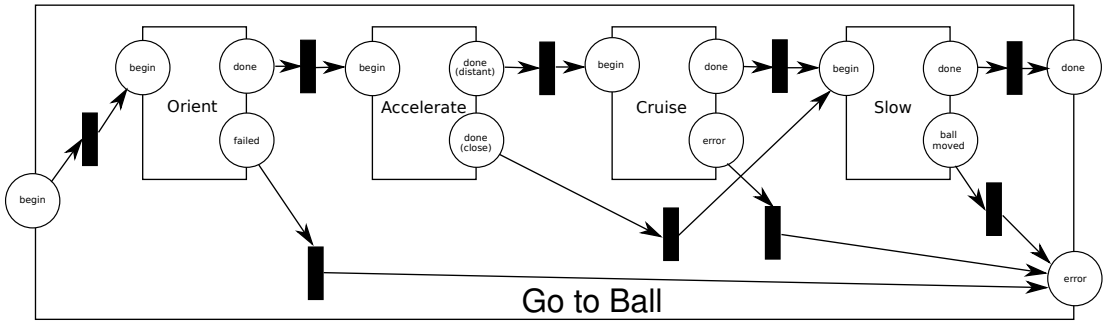


Figure 3.1: A behavior to approach a stationary ball.

probabilities of completion states. This is a pleasingly automated method, as Markov models have been well-studied. Unfortunately, it requires an extremely precise description of the environment in the form of a temporal Bayes net. (This precision may be relaxed to some extent, as I will describe.) Furthermore, in complex environments, when the agent's internal states may eventually couple otherwise unrelated environment variables, the time and space required to complete the analysis may be excessive. Thus, this method gives strong results, but it has a set of drawbacks. §3.2

- Sampling methods can provide guarantees sufficient for some purposes. In this case, we would sample initial states and executions of the behavior and use the statistics of the resulting states to approximate the true distribution over the behavior's effects. Alternately, we can back-trace from errors and then determine how likely those precursors of problems are. As always, sampling methods guarantee nothing absolutely, but if sampling is done correctly, then we can still get statistical certainty estimates about the analysis. The degree of certainty depends on the number of samples we take and the size and structure of the environment, so as always the analysis will get increasingly complex as we demand tighter bounds in a messier world. §3.3

- Finally, the most adaptable method is to use hand-constructed proofs to show that a given behavior reaches some state. These are the most flexible analyses in terms of the environment because the human analyst can work out how to relax the restrictions on the environment as much as possible while still keeping the proved result intact. Thus, the same behavior can be used in similar, related environments without re-analysis. §3.4

3.1 Analyzing Higher-Level Behaviors

The analysis of a complete behavior is built on the individual analyses of its components. If we can already summarize what each module will accomplish, then we can understand what their combination will do. We can do this just as is done in standard logical planning [51]. Each module has preconditions for it to operate correctly and postconditions it guarantees when it finishes. (It may also have alternate end conditions, such as if something unexpectedly changes in the environment to make its goal impossible.) By chaining modules, as shown in Figure 3.1, so that the postconditions of one action satisfy the preconditions of the next one, then we know that the second will complete successfully. This allows us to decompose the analysis of the behavior.

Algorithm 3.1 HIGH-LEVEL-ANALYSIS

Require: Module m

for all sub-modules $s \in m$ **do**

Output$[s]$ = PN-ANALYSIS(s)

P(Output$[s]_i$|Input$[s]_j$) = ANALYSIS(s)

end for

Verify that Precondition[FirstModule(m)] is satisfied by Precondition(m)

for all sub-modules $s \in m$ **do**

Use PN-ANALYSIS(m) to find what outputs go to s's inputs, $\mathbb{1}$(Input$[s]_i$|Output$[q]_j$)

Verify that Precondition[Input$[s]_i$] is satisfied by Postcondition[Output$[q]_j$]

end for

Verify that Postcondition(m) is satisfied by Postcondition[LastModule(m)]

For each sub-module, we both check its pre- and post-conditions and also analyze it to get the probability of each of its post-conditions. A detailed algorithm is given in Algorithm 3.1. For instance, in Figure 3.1, we would first analyze the *Orient* module to determine with what probability it ends in *done* and *failed*. Then, we make sure that the preconditions for *Accelerate* match the postconditions for the *Orient.done* case. Next, we analyze the probability of each of *Accelerate's* outcomes, and so on. Note that this will require checking two different postconditions of *Accelerate*: we must make sure that if the ball is distant, the postcondition matches the preconditions of *Cruise*, while if the ball is close, then we must immediately match the preconditions for the *Slow* module.

However, this simple method is insufficient when concurrency is present. If more than one module may be active at a time, then we cannot in general trust that they will work as intended,

since the actions of one may interfere with the other. (Remember that the world, in the worst case, may be completely coupled, so every action taken affects every variable.) Instead, either we must show that the co-active modules do not interfere, such as by showing that the actions each emits do not affect the variables the other monitors, or else we must perform another low-level analysis to see how the modules work together. We can identify which modules may be co-active via ordinary Petri Net reachability techniques, including conservation analysis. Then, having identified the groups that may operate simultaneously, we will analyze each of these groups with the low-level behavior analysis techniques described in the rest of this chapter. This modified algorithm is given in Algorithm 3.2.

Algorithm 3.2 HIGH-LEVEL-CONCURRENT-ANALYSIS

Require: Module m

Use PN-ANALYSIS on m to find all sets S_* of concurrently-active sub-modules

Construct m' identical to m but with one sub-module corresponding to each S_i

return HIGH-LEVEL-ANALYSIS(m')

So, that is the recursive case for analyzing hierarchical behaviors. Now, let us turn our attention to the methods used for analysis of lower-level components.

3.2 Markov Model Analysis

Together, the environment and behavior comprise a stochastic system. The environment moves in random ways, and the behavior moves in deterministic ways. Importantly, these changes only depend on these components' current state, not on past state. That means that for the full system, the Markov property holds—its new state only depends on its previous state. So, we can construct a Markov model of the full system, including both the environment and the behavior. Thus, we reduce our complex system to one that is quite analyzable and has been thoroughly studied. [15]

This most exacting analysis method, which requires the most detailed environmental description, also produces the most rigorous analysis. That detailed statistical description of the environment gives us useful guarantees about how our behavior can unfold. The complication is that the detail that gives us power can also bog us down. Also, in order to make sure that our model has finite state, we may need to place restrictions on our behavior's inferential component, because in general inference may result in continuous state. However, before discussing those issues, let us directly explore this method.

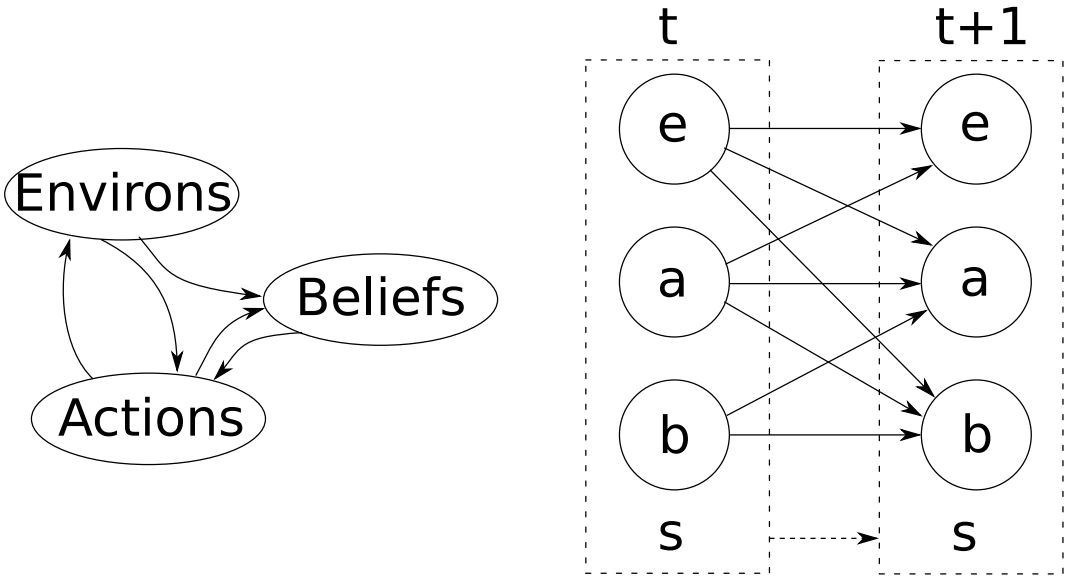


Figure 3.2: The Markov process model of our agent in its environment. On the left, you see the general ways in which the three components affect each other. On the right, you see the temporal relations between these three components, not quite fully connected, which together make up the full state s.

Because our Markov process is composed of more than one interlocking component (the environment and the agent's procedural and inferential components), its transition probabilities may be computed from those of the individual components. The full process, S, is composed of those three components: E (environment), B (belief/inferential), and P (procedural). In particular, in every time step, the environment changes according to some distribution that depends on its previous state and the actions taken, which are determined by the procedural state: $\mathrm{P}(e^{(t+1)}|e^{(t)}, p^{(t)})$. Similarly, the inferential component updates its state based on its previous beliefs, the observations from the environment, and the actions taken. It does this possibly in a deterministic way, but that is generalizable as the stochastic update $\mathrm{P}(b^{(t+1)}|b^{(t)}, e^{(t+1)}, p^{(t)})$. Finally, the procedural component shuffles some tokens based on the inference and the direct (reflexive) environmental inputs, moving it into a new state. Again, this may be deterministic, but we may generalize it with the stochastic distribution $\mathrm{P}(p^{(t+1)}|p^{(t)}, b^{(t+1)}, e^{(t+1)})$. Then the update over the full state is just the product of these three factors, so

$$\mathrm{P}\left(s^{(t+1)}\middle|s^{(t)}\right) = \mathrm{P}\left(e^{(t+1)}\middle|e^{(t)}, p^{(t)}\right)\ \mathrm{P}\left(b^{(t+1)}\middle|b^{(t)}, e^{(t+1)}, p^{(t)}\right)\ \mathrm{P}\left(p^{(t+1)}\middle|p^{(t)}, b^{(t+1)}, e^{(t+1)}\right).$$

Equivalently, we can rewrite this in terms of transitions between full states s_i and s_j.

$$\mathrm{P}\left(s_j|s_i\right) = \mathrm{P}\left(e(s_j)\middle|e(s_i), p(s_i)\right)\ \mathrm{P}\left(b(s_j)\middle|b(s_i), e(s_j), p(s_i)\right)\ \mathrm{P}\left(p(s_j)\middle|p(s_i), b(s_j), e(s_j)\right)$$

A Markov process may be analyzed by describing the distribution over its states as a vector. [15] Specifically, if we have a Markov process with a state $s \in s_1, s_2, \ldots$, then we define π such that $\pi_i = \mathrm{P}(s = s_i)$. Similarly, we can construct a matrix M such that $M_{ij} = \mathrm{P}(s^{(t+1)} = s_j|s^{(t)} = s_i)$. Then, we compute transitions over time as $\pi^{(t+1)} = \sum_{s^{(t)}} \mathrm{P}(s^{(t+1)}|s^{(t)})\mathrm{P}(s^{(t)}) = M\pi^{(t)}$. That gives us the happy result that $\mathrm{P}(s^{(t)}) = \pi^{(t)} = M^t\pi^{(0)}$. In this way, we can calculate future distributions very easily. Note: to calculate just $\mathrm{P}(s^{(T)})$, we can do this in $O(\log(T))$ time by subdividing the matrix exponentiation; or, we can get all $\{\mathrm{P}(s^{(t)}) : t \in \{1, \ldots, T\}\}$ in time $O(T)$.

Now that we have captured the basic structure of our behavior, let us also include its outcomes. We make our goal and failure states absorbing, $\mathrm{P}(s^{(t+1)} = s|s^{(t)} = s_g) = \mathbb{1}[s = s_g]$. Then, we can calculate the probability that the goal is achieved, $\lim_{T\to\infty} \mathrm{P}(s^{(T)} = s_g)$, as the g^{th} element of $\lim_{T\to\infty} M^T\pi^{(0)}$. (In practice, we can approximate that infinite limit by some finite T.) Thus, by a series of matrix operations, we can analyze our behavior precisely and determine how likely it is to succeed. This analysis will work for any behavior and environment with finite, discrete states.

This method has some noticeable advantages. First, we can use it to analyze any behavior with a finite state; we need not restrict ourselves to any particular formalism, such as Petri Nets.

Second, because it puts tighter restrictions on the environment, it also provides stronger claims about the results of the behavior. Indeed, it gives an exact accounting for the results of the state that is perfectly correct (if the environment is actually as described). Finally, it can be performed in an entirely automated way, requiring no human assistance for analysis.

There are two main drawbacks to this analysis method. First, it requires a very detailed description of the environment, which may be difficult to obtain for real environments. Not only is it hard to obtain, but if that description changes slightly, we must re-do our analysis. If we prove something about a behavior using less restrictive assumptions about the environment, then it will apply to many environments; however, that will not be the case when we define the environment so strictly. Therefore, in §3.2.2, we explore a version of Markov analysis that works on a whole range of environments at once.

The second main drawback of this method is that it has complexity $O((|E|\,|B|\,|P|)^2)$. On the one hand, that seems to be merely quadratic. However, if any of those components can be subdivided, then the size of the component is the size of those components' full cross product, which is exponential in the number of sub-pieces. For instance, if the environment E contains n variables that can each take m states, then $|E| = m^n$. We can combat this growth by taking advantage of independences between these variables, in the same way as people use independences to make belief propagation efficient, and that is explored in §3.2.1.

3.2.1 Complexity and Subdivision

Suppose that the environment is described by a dynamic Bayes net that contains several groups of variables. The variables in each group are dependent on each other, but independent of the variables in the other groups. Write e for the full environment state, and $e_{(j)}$ for the state of the j^{th} group of variables. Then the probability of the full state may be decomposed, so

$$\mathrm{P}(e^{(t+1)}|e^{(t)}) = \prod_j \mathrm{P}(e_{(j)}^{(t+1)}|e_{(j)}^{(t)})$$

However, when we add in the action of the agent, these groups of variables may become coupled. For instance, suppose that one action the agent takes affects a variable in both of two groups of variables. Then we cannot simply decompose the probability into this product, because the variables are not independent. Therefore, we must assume not only that these variables are decoupled in the environment, but also that they are decoupled even given the action of the agent. Similarly, we may extract decoupled components of the agent's procedural and inferential systems. If this condition of independence holds, then we can analyze each of those components in isolation. In this way, we can decompose a complex analysis into several simpler analyses.

Now, this is a stronger independence than required in Bayesian inference. After all, in Bayesian inference, we may divide variables into groups ("cliques") that are independent of other variables given an interface. Then, we can use an algorithm such as the junction tree algorithm [38, 47] to do inference, with the time dominated by the size of the largest clique. That means that the inference cost does not grow exponentially with the number of variables, just with the size of the biggest clique. At first glance, it seems like this same technique could be applied here.

However, it turns out that during temporal inference, clique-based methods do not work as well as they do in fixed Bayesian networks. The problem comes because during temporal inference, we keep eliminating variables from the past. After all, we do not want to keep them around indefinitely, as then the inference cost would grow over time. But, as we eliminate those variables, the previously indirect connections between current variables become explicit, increasing the size of the present's cliques. This process can only extend so far, and eventually it converges to some stable limit. Unfortunately, that limit includes explicit connections for all related variables, meaning that only very strong independence, such as described above, will help. So, techniques for doing inference in DBNs have the same complexity issues as the methods given here. [69, 59] [1]

Also, note that because the environment variables are coupled not only through the environment's DBN but also through the behavior, we cannot necessarily use arbitrary methods for DBN approximate inference. This additional coupling makes the problem noticeably more complex. This makes exact and sampling methods much more appealing.

3.2.2 Bounded Markov Process Analysis

One main drawback of the Markov process analysis given above is that it requires a very detailed description of the environment. In particular, we must know all the environment transition probabilities, $\mathrm{P}(e^{(t+1)}|e^{(t)}, p^{(t)})$, with great precision. In reality this is rarely the case; in fact, the Markov process is likely to be an approximation of the underlying physical system. Instead, it would be better to perform such an analysis to show success over a whole range of possible environments, so that as long as the environment is similar to our approximation, we can be assured of success.

So, suppose that instead of using the precise transition probabilities $\mathrm{P}(s^{(t+1)}|s^{(t)})$, we know bounds on these probabilities. For instance, we might have transition probabilities bounded by L and U:

$$\forall_{i,j}\ L_{i,j} \leq \mathrm{P}(s_j|s_i) \leq U_{i,j}$$

[1] Note that it may be possible to define factorable distributions (or approximate factorable distributions, via variational methods [40]) that make inference feasible even in the case of some kinds of dependencies.

We can still analyze this situation, too. After all,

$$\begin{aligned} \mathrm{P}(\text{reach goal}|s_i) &= \sum_j \mathrm{P}(\text{reach goal}|s_j)\mathrm{P}(s_j|s_i) \\ &\geq \sum_j \mathrm{P}(\text{reach goal}|s_j)L_{i,j} \end{aligned}$$

Then, we will use the same matrix analysis technique, but with a new matrix based on the bounds. Construct a new matrix N such that $\forall_{i,j} n_{i,j} = L_{i,j}$. Then, define a new state s_u to represent an unknown state and expand N to have a row and column for u. Define transition probabilities for u by setting $\forall_i n_{i,u} = 1 - \sum_j L_{i,j}$ and $n_{u,u} = 1$. (This makes sure that N represents transition probabilities that sum to 1, which is necessary for useful analysis.) Now, we compute a lower-bound distribution $l^{(\infty)} = N^{\infty}\pi^{(0)}$ that gives a lower-bound on the final distribution of states. That is, $\mathrm{P}(\text{end in state } j|\pi^{(0)}) \geq l_j^{(\infty)}$. We will also end up with a probability mass we do not know how to distribute, $l_u^{(\infty)}$. We can use this to also provide upper bounds on the probability of outcomes, since $\mathrm{P}(\text{end in state } j|\pi_0) \leq l_j^{(\infty)} + l_u^{(\infty)}$.

So, we can analyze our behaviors even for ranges of environment descriptions. This makes analysis stronger because we can prove effectiveness once, and have that analysis hold even if the environment changes. An important side effect of this is that it makes our analysis apply even for real, physical environments, where transition probabilities must be measured or even estimated. Then, even if we are not entirely sure of our environment parameters, this method can give useful guarantees anyway.

3.2.3 Discrete Inference State

In order to write a transition matrix for the states of the Markov process made by the environment and a behavior, we require a finite number of states. The environment, composed of a dynamic Bayesian network of finite, discrete random variables, has a finite state. Similarly, the Petri net procedural component of the behavior has finite state, since its state is made up of places with finite, discrete numbers of tokens. Unfortunately, the inferential component of the behavior can in general have infinite state.

That infinite state may seem surprising. After all, the inferential component has the same structure the environment: it is a dynamic Bayesian network. However, while the environment's DBN is used in a generative way, assigning variables random values, this one is used for inference. The agent does not just keep known values of variables—it keeps inferred distributions of variables, based on all the evidence seen in the past. Those distributions are usually continuous quantities, since even a binomial distribution has a continuous parameter describing the probabilities of the

two outcomes. This is the same problem that makes solving POMDPs so complex; the state space may grow in a way that can flummox simple algorithms [42].

There are three main solutions to this problem. First, we can act as though the environment variables the behavior relies on are known. After all, when we are analyzing the behavior, we are tracking the full state of the environment, so we really do know those variables' values. However, this will lead to unfortunate outcomes of the analysis, where we are analyzing a behavior that knows much more than it would if it was acting in the real world. This is almost certainly a mistake in all but the most easily observed environments, where this difference in inference requirements does not matter much.

The second solution is to restrict the inference component's state representation. Instead of letting it track continuous distributions, we can discretize its state space. After all, the inferential component only affects the behavior by enabling or disabling transitions at certain thresholds, and we know those thresholds. We can make discrete state spaces based on those boundaries and use those to approximate our agent's behavior in the world. That approximation will sadly not be perfect, but it can still produce useful analyses.

The last solution is to use a more flexible sort of model. Our analysis need not use a discrete Markov model—there are also continuous Markov models. For instance, it is easy enough include one or more Gaussian random variables. That is, in fact, the basis of Kalman filtering. That will change some of the Markov analysis method described above, since we will now track some Gaussian components. These will also be approximations to the behavior's true belief state, so they will not quite duplicate the behavior of our agent in its true environment. Still, that allows a sort of inference that will be useful in certain conditions.

3.3 Sampling Analysis

When an exact analysis grows too complex, it is still possible to approximately analyze a stochastic system. Simply by sampling from the statistical model, we can explore the parts of the system that are likely, while ignoring much of the irrelevant complexity. That way, we can focus our analysis on the parts we care about and at least get statistical guarantees on our system. Sampling analysis is therefore useful for checking the effectiveness of our behaviors.

In our case, our behaviors are deterministic given their inputs, so the randomness comes only from the environment. The environment, recall, consists of random variables that depend on each other and on the agent's actions. We have a description of their relations and distributions, which are given by the PRM portion of our model. We can use this description to sample the

variables of the environment. Having sampled the environment variables, of course, production of the behavior is very simple—just simulate it, letting it see the observed variables we sampled. As we move forward in time, we will need to sample future time steps in the environment; we then condition these variable choices on the existing variables and on the behavior, as needed. There are a variety of sampling methods, including Gibbs sampling, particle filtering, and others. See [24] for a detailed discussion of many of them.

For our analysis purposes, we only care about the possible outcomes once the behavior is complete. That is, we care most about whether a module is successful or not, not what particularly happens within the behavior. Therefore, the standard results for multinomial sampling apply. We have m outcomes, each occurring with probability p_i. The unbiased estimate of p_i is $\hat{p}_i = n_i/n$, where n_i is the number of times outcome i is observed, and n is the total number of samples. According to [72], with probability α, we can bound the true p_i by a function of $\hat{p}_i$:

$$\frac{\hat{p}_i + \frac{1}{2n}z^2_{1-\alpha/2} \pm z_{1-\alpha/2}\sqrt{\frac{\hat{p}_i(1-\hat{p}_i)}{n} + \frac{z^2_{1-\alpha/2}}{4n^2}}}{1 + \frac{1}{n}z^2_{1-\alpha/2}}$$

where z_α is the α percentile of a standard normal distribution. This bound is good even when p_i very close to 0 or 1, which is important, since very often we are interested in error cases that happen extremely rarely. There are also other bounds that might be useful in other circumstances—[3] give a comparison. So, to obtain a sufficiently tight bound on the probability of success, we need merely sample a sufficient number of times to drive down the variance.

3.3.1 Backward Sampling

Forward sampling, though simple, is at its worst when trying to calculate accurate probabilities of unlikely events—and if our behaviors are effective, failure outcomes should be extremely unlikely. That makes these sampling methods less than ideal for our problem. Instead, since we want to focus on the chances of low-probability events, we should condition on those unlikely events, by setting that as evidence, and sample for likelihood based on that.

That is, we will do sampling in reverse by treating an error condition of the target module as evidence. Then, conditional on that conclusion, we go backward in time, sampling earlier possible environment and agent execution traces. From those samples, we may then estimate how likely that error condition is. This will give a more accurate estimate for unlikely cases than will pure forward sampling. Such reverse sampling is not entirely a simple problem, but it has already been studied, especially as it applies to smoothing time series [59, 30, 6].

3.4 Hand-Wrought Analysis

The most general but labor-intensive analyses are those produced by humans. Such an analysis may be applicable over a wide range of environments and behaviors. However, each such analysis will likely be different, so I can only give a general framework and specific examples here. Only the modules that directly interact with the environment would require a proof like this, as higher-level modules can be analyzed as a composition of their components.

The general hand-analysis framework is based on the use of an error metric, much as is done in control theory [67]. First, we define an appropriate error metric that will be a proxy for the goal conditions. In particular, it will be below some threshold only when the goal conditions are met. As such, it may be a composition of different error metrics for each of the goal conditions. Then, we show that this metric is improved in every time step, by an amount sufficient to cause it to eventually go below the target threshold. Unfortunately, showing this reduction may be impossible in some circumstances. For instance, if the agent has momentum away from a target, more than it can reverse in a single time step, then in the next time step it will have a higher distance from the goal.

Therefore, it is often advantageous to define a set of bounds on state variables that our behavior maintains—bounds that guarantee an error reduction. Then, we show two things: that the error decreases, and that the bounds hold at the end of each time step. In particular, the bounds must satisfy the following conditions:

1. these bounds are maintained by the behavior in every time step
2. given these conditions, the error metric is reduced in every time step
3. the goal state satisfies these bounds

So, if we just derive appropriate bounds and show that these conditions are satisfied, then this will show that our behavior reaches its goal state, given that its initial conditions lie within the bounds.

To show how this hand-analysis is used to guarantee the success of a behavior, there is an example in Appendix A.

Chapter 4

Recovering from Surprises and Interruptions

One unfortunate fact of the real world is that things do not always go as we would like them to. Sometimes, things that we never could have imagined surprise us and send awry even our best-laid plans. For instance, our soccer-playing robots could be interrupted by a second ball getting loose on the field, by getting caught on another robot, by a hardware malfunction, or even by an earthquake. More commonly, they might encounter situations where their expectations of the future were simply wrong. Perhaps they encounter a team using a brand-new strategy, or accidentally kick the ball somewhere unusual. Some of the above issues are outside the scope of a soccer-playing behavior, but for those that permit the game to continue, our robots' behavior must not break down just because something unexpected happened.

For the simplest sort of agent, a purely reactive one, an interruption is only a problem if it gives the agent inputs it was not designed for. As soon as it again sees familiar inputs, it can take the appropriate action. However, because the environments we are considering are too complicated for such simple behaviors, and the agent instead keeps careful track of the things it has perceived, determining which observations are obsolete and which are relevant can be tricky. Fortunately, we can take advantage of the fact that our agent does not need to entirely understand its environment to make useful actions. Instead, it can often do something productive based only on its local information while its model is out of date. Such actions will not be optimal, but they will be better than simply giving up entirely or continuing as though there had been no disruption of the environment.

The things that go wrong, sadly, are far too many to try to handle each one separately—after all, there are many that are impossible to even anticipate. Therefore, instead of trying to

recover from every possible interruption differently, we use a set of **safe states** that are simple and locally defined. Such a safe state is a stable situation from which the agent can consider and then begin a new action. So, when it encounters a problem, it attempts to reach one of those safe states. We label those states **safe states**, and we use the word **recovery** to mean the process that is initiated upon **error detection** and ends in such a safe state.

When something untoward happens, the agent then attempts to use local information to retreat to one of these safe states. There, it may reconsider its plans and gather additional information from relative safety. Note that safe states need not be static; they can be quite dynamic. For instance, a safe state for a bicyclist will involve some forward motion, which is needed to provide balancing stability. This forward motion also necessitates some sort of steering, so the we see that the safe state is a process, not a fixed position.

Recovery involves two components. First of all, our agent must produce actions to try to move the outside environment to a safe condition. It must also set its internal state to one that now reflects the true state of the world. This may require letting go of some assumptions and rebuilding parts of its internal models. Most of all, it will require abandoning partially-completed behaviors that are no longer useful. Therefore, we need a way of cleaning up its internal representations of incomplete actions.

4.1 Machinery

Within a given behavior or module, when our agent encounters an error and must quickly reach a safe state, it must abandon possibly concurrent parts of its current behavior. In order to simplify this recovery process, we once more extend the standard Petri Net definition. In particular, we introduce **set arcs**. These arcs, which go from transitions to places, have the distinct property of *setting* their outputs to values specified by the arc weights, rather than modifying them by those amounts. This makes it very easy to reset large parts of the network at once, simultaneously abandoning multiple concurrent activities. For instance, in Figure 4.1, the recovery transition uses set arcs to set all the places in the module when an error is discovered.

Note that for any recovery transition with set arcs, this same setting-of-places effect could be achieved by the introduction of a particular collection of other places and transitions. That is, there is an equivalent Petri net without set arcs that would still cause the target places to end up with the right number of tokens. So, the set arcs do not change our representational power; any combination of effects they could describe are already feasible with the existing model. However, by allowing a simpler description, they make any sort of representation manipulation much more

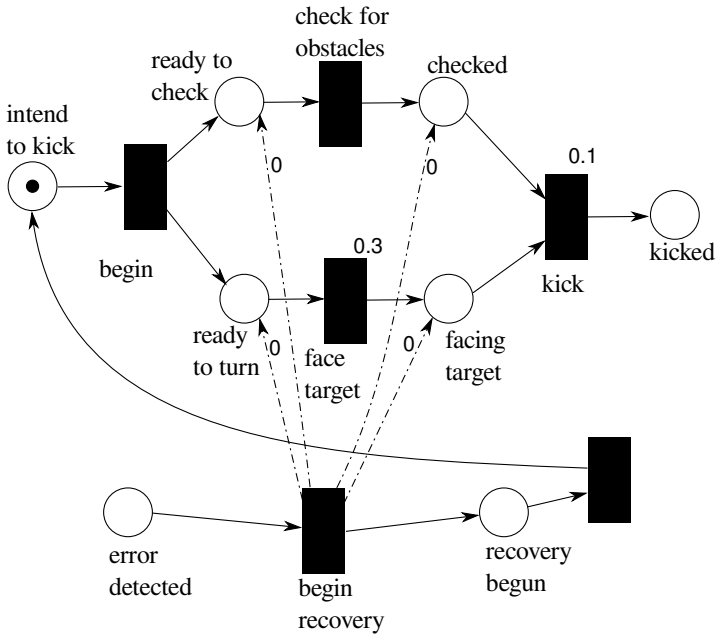


Figure 4.1: Here, set arcs from the *begin recovery* transition put the entire module into the desired state.

feasible. So, this simplification effect makes it easier to create networks, whether by design or by learning, and also to analyze them.

For example, in Figure 4.2, we can see how the previous example (Figure 4.1) could be created without any set arcs. It uses inhibitory arcs to stop the progress of the network, plus one transition for each place to reset that place. However, it is a much more complex structure, and in the event of any changes to the basic structure of the network, it will have to be heavily modified. Furthermore, that complexity also makes it more difficult to analyze.

4.2 Modules

Recall that the behaviors we use are deliberately designed to be hierarchical. For instance, the "pass" behavior may use the lower-level "kick" behavior without relying on any detailed knowledge about how kicking is done. Because of this compartmentalization of skill, errors will often be detected at the behavioral level they affect. If our agent loses control of the ball, that will first affect the low-level "kick" behavior. If an opponent covers the player our agent was going to pass to, that will instead affect the higher-level "pass" behavior.

So, error detection may be needed in any module. But, because the low-level behaviors

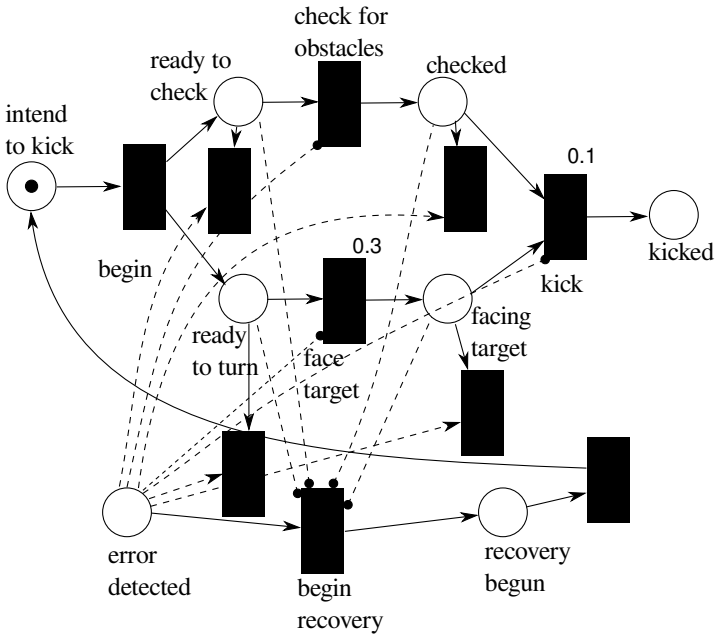


Figure 4.2: Here, we see how the recovery in Figure 4.1 could have been performed without set arcs; however, this is a much more unwieldy graph and will be harder to work with.

are only useful in the right context, and the high-level behaviors will be affected by the failure of low-level components, error and recovery information must propagate up and down through the behavioral hierarchy. Therefore, it is important that modules not only have "start" and "end" places in their interfaces, but also "abort" and "failed" places. These allow the needed sort of signals to be transmitted.

Of course, there may be more than one kind of failure, and the type of sub-behavior failure will affect how the higher-level behavior recovers. If the kick fails because an opponent stole the ball, then the pass behavior should give up. However, if it fails because the robot's leg simply missed the ball, then retrying the kick might be best.

These recovery mechanisms share a lot in common with the exception handling in modern computer languages, and that should not be surprising—after all, these two domains share common goals. In both cases, we want to identify errors where they are most relevant but handle them where appropriate action can be taken. Then, based on exactly what the situation is, the module either recovers immediately or informs higher-level components about the problem. The difference between these two error-handling schemes is that our action representation is inherently concurrent, while computer languages are made for describing serial processing schemes. That is why, in our action representation, error information may also flow down to low-level components. These simpler

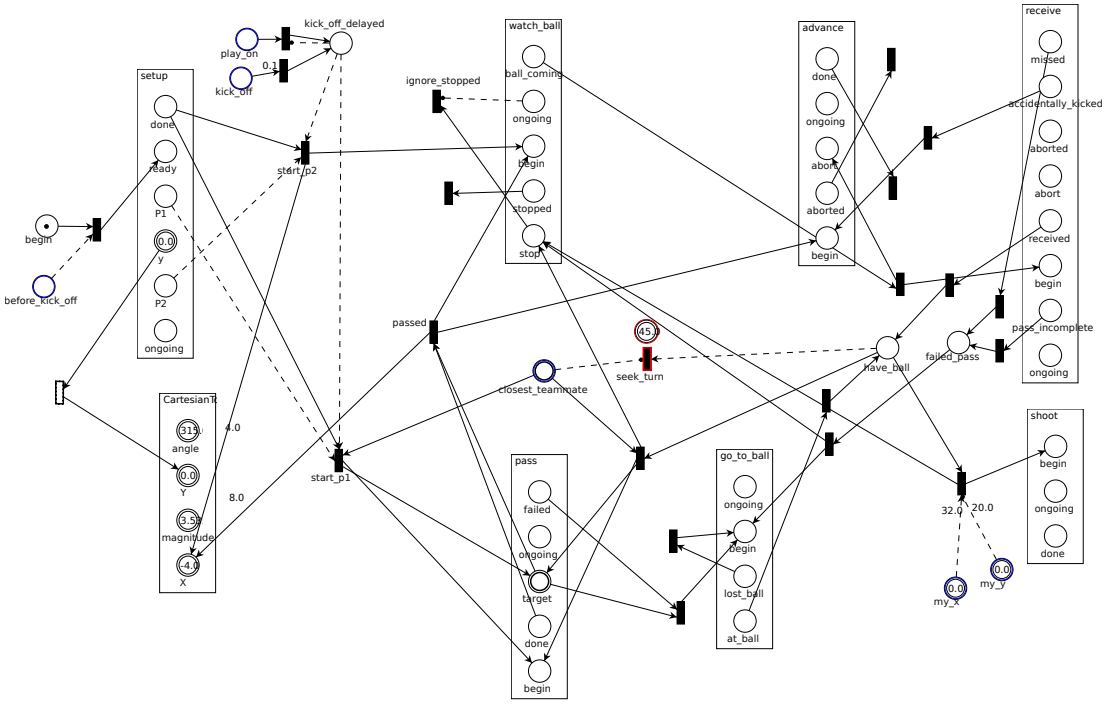


Figure 4.3: The high-level structure for a behavior which passes the ball back and forth, advancing on the opponent's goal, and eventually shooting once the goal is close enough.

behaviors may need to abort or change their operation based on errors that occurred beyond their own scope. Note, however, that this part of the error-handling scheme does not require any new representational capability. Instead, it simply requires a straightforward expansion of the existing hierarchical structure.

4.3 Examples

Here, I present the example of a soccer behavioral network with added recovery components. It is a behavior that passes the ball to its teammate, moves forward, and eventually shoots on the opponents' goal once close enough. The high-level behavior is given in Figure 4.3.

This high-level structure is built out of many sub-networks. They range from relatively simple ones, such as *setup*, to more complex ones, such as *receive*. The basic structure is simpler than the diagram makes it seem. The player repeatedly passes the ball to its teammate, then begins watching the ball as it runs forward. When it has gotten to the desired forward location, it waits to receive a pass. Once it has received the ball, it starts the cycle again and passes back to its teammate. The additional complexity of the network is largely due to its error recovery systems,

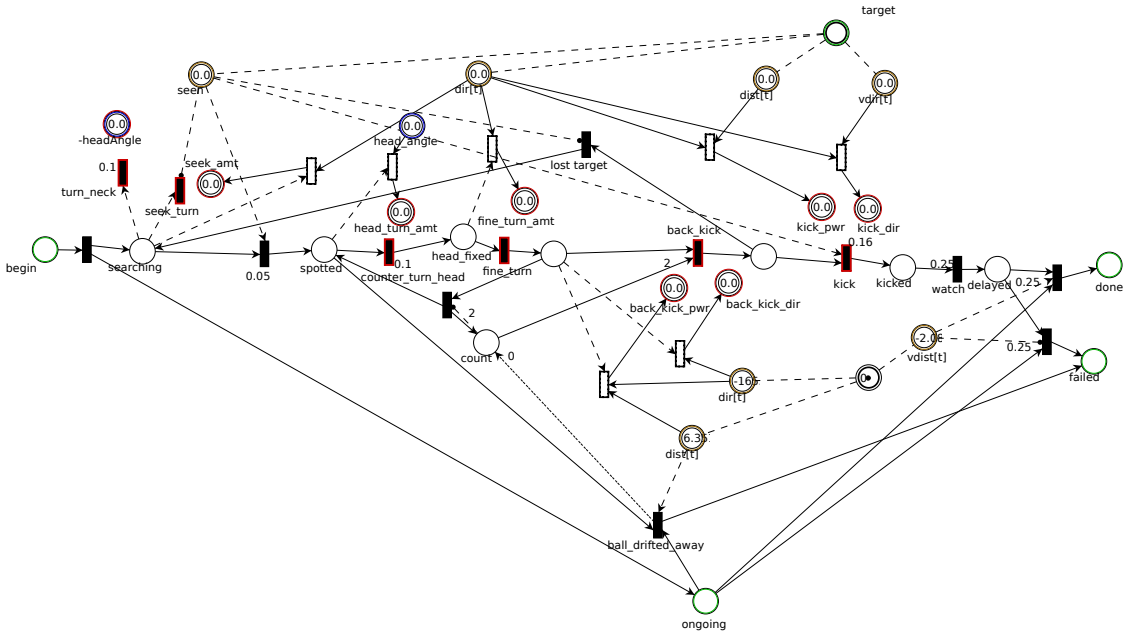


Figure 4.4: The low-level behavior to *pass* the ball to a teammate. If the distance to the ball grows too high while it orients toward the teammate, it recognizes this and goes to the *failed* state.

which are described in detail below.

4.3.1 Passing the Ball

As a simple example of error detection and recovery, consider the low-level *pass* network, shown in Figure 4.4. First, let us understand its basic structure. It has a main path where it repeatedly turns to see the target, if it does not yet, then turns directly toward the target, and finally kicks the ball. It loops on the turning to face the target, doing so twice because the soccer environment introduces error proportional to the turn size, so the second, smaller turn is much more precise.

However, while the player is getting ready to pass the ball, something could happen, such as the ball drifting away or an opponent stealing it. The pass network, however, is not designed to take corrective action in such cases—what should happen after such an event is beyond its scope. Instead, it simply checks for the error and, if it finds it, goes to the *failed* state. That way, the higher-level behavior can deal with it appropriately.

A different error could occur during passing. After kicking the ball, something could have gone wrong (such as the ball being stopped by an obstacle), and the ball would not be moving away. In that case, the kick would not be successful, but again the *pass* network itself would be

unable to take appropriate action. Therefore, in this case also, the network goes into the *failed* state.

Note that this network makes use of only a single set arc. It uses that to reset the internal counter for repeating the turn. If it did not reset this counter, then the next time the *pass* network was used, it would start in an incorrect state. However, it only needs one set arc because most of the state does not need to be reset.

So, now we know that upon detecting an error, the *pass* network passes it up to a higher level; what does that higher level do with that information? As we see in Figure 4.3, the higher-level behavior takes the token from the *pass*'s *failed* place and activates the *go_to_ball* low-level behavior. So, having lost the ball for one reason or another, the high-level network relies on an appropriate other low-level behavior to handle the problem. An equivalent way to view this recovery situation is that the agent attempts to reestablish a safe state: possessing the ball. When the error occurs in the low-level behavior, it relies on the high-level behavior to successfully navigate that path to safety.

4.3.2 Advancing down the Field

Another example problem recovery situation can be seen in Figure 4.5. This is a simple enough behavior that nothing can really go wrong. The player just keeps moving toward a given destination. If it gets off course, it turns itself to get back on course; if it remains on course, it emits *dash* commands to keep its velocity up. Therefore, it does not detect any error situations and does not initiate any recovery.

However, this low-level behavior may become inappropriate if something changes elsewhere. Suppose that the destination is no longer appropriate, or that the player needs to stop running in order to do something else. Therefore, this network has an *abort* input that immediately shuts down all activity here and sets the *aborted* output. It does this by using *set* arcs to reset the internal state of the network. Its safe state is simply not moving—when something happens, the player's first response must be to cease its previous motion. As we see in Figure 4.3, this *abort* signal is sent when the *watch_ball* network detects that the ball is coming toward the agent. Then, the agent's appropriate action is to ready itself to receive the ball, so it also sends the *begin* signal to the *receive* low-level action.

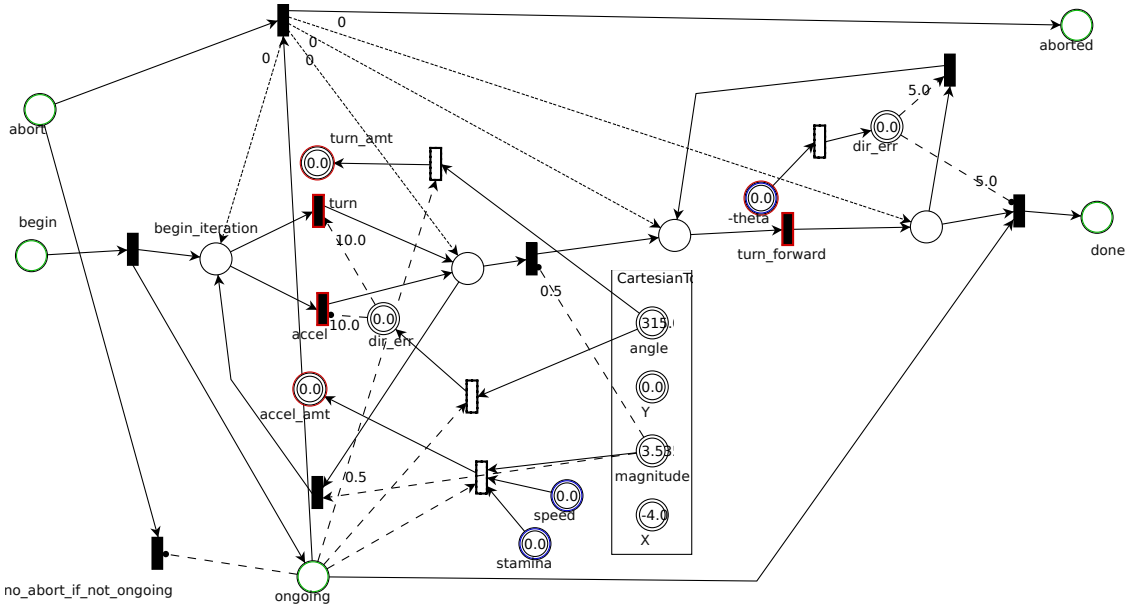


Figure 4.5: The low-level *advance* behavior, which moves forward to a new position. It does not directly detect errors, but it may be aborted if other conditions change elsewhere.

4.3.3 Keeping Watch on the Ball

It is relevant to inspect the error-detection component of the *watch_ball* network, seen in Figure 4.6. This network runs concurrently with active behaviors such as *advance* and *receive*, and its goal is to keep the ball in view and to alert the other modules when the ball is coming. Thus, whenever it sees the ball approaching the player with a sufficient velocity, it sets the *ball_coming* output place, so that the high-level network can use this information appropriately to control the active behaviors also underway.

Also, because the *watch_ball* behavior has no obvious internal stopping point, and it is hard to know exactly when to stop watching the ball and start watching something else, it leaves that control up to the high-level behavior. That high-level behavior stops watching the ball once it has received the ball. Then, it sets the *stop* input of the *watch_ball* network. This induces the *watch_ball* network to reset its internal state and mark the *stopped* output, thus letting the high-level behavior know that it stopped successfully.

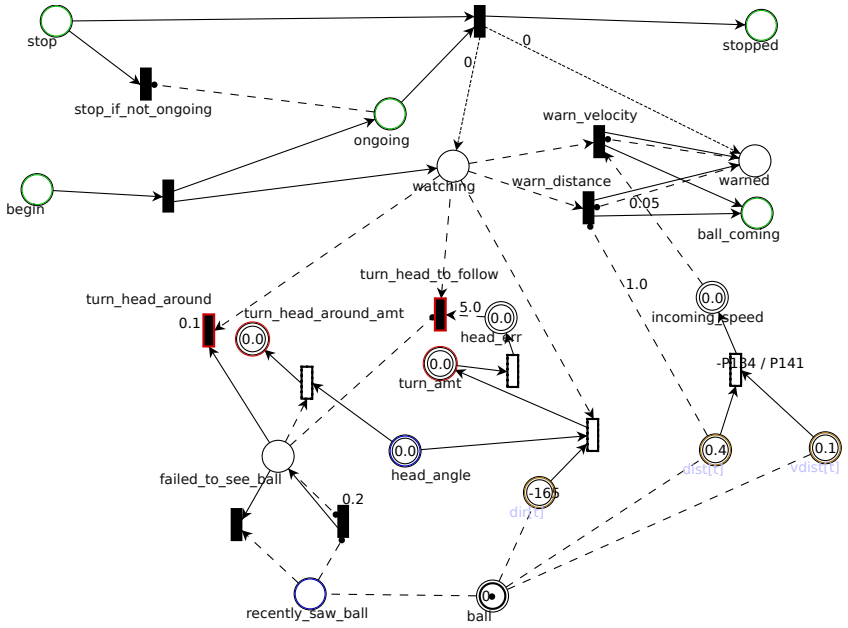


Figure 4.6: The low-level behavior for watching the ball. This behavior is run concurrently with more active behaviors such as *receive* and *advance*.

Chapter 5

Results

In the chapters above, I have described a new architecture for dealing with complex, real-world actions. Being designed to capture both the structural properties and the needed inferences of actions, it is built on two existing components, Petri nets and Probabilistic Relational Models. These components, especially the Petri nets, were modified to become a better match for the requirements of real-world, real-time, complex actions. In particular, the Petri nets received added modularity, mathematical mappings, and direct world interface mechanics. I fit the two pieces together in such a way that each could view the other in its own terms, meaning that the internal mechanisms for each were maintained without modification. This required additional extensions for using PRM objects as tokens in the Petri net, as well as controlling which relations are active in the PRM. Furthermore, I provided additional extensions to allow simple recovery from errors.

Also, by taking advantage of the existing analysis methods for each component, I developed combined analysis methods for the combined architecture. These methods can show the correctness and effectiveness of actions, embedded in their environments. They not only take advantage of the individual analysis tools for the two components, but they also draw on the hierarchical structure of actions to simplify the analysis. In order to make analysis as useful as possible, I include methods optimized for varying degrees of environmental complexity, including both direct and approximate analyses.

Here, I discuss the results of this architecture: its implementation, demonstrations, and comparisons to a set of baselines.

5.1 Implementation

The representation described here has been implemented and demonstrated. The implementation is based on existing open-source tools for manipulating both Bayes nets and Petri nets. It was written in Java in order to best take advantage of such tools. It is built upon a foundation of *Bayesian Network tools in Java* (BNJ) [37] and the *Platform Independent Petri net Editor* (PIPE) [8, 9]. These tools each include both a backend (simulator/inference engine) and a graphical user interface (GUI). The software is available at http://soccer.barrettnexus.com/. It uses ATAN [39] to communicate with the RoboCup 2D simulator.

However, each of these tools required modifications to match the requirements described above. That is why it was important to work with open-source tools like these: that way, they were exposed to the tinkering I required. BNJ was extended with support for inference in temporal Bayes nets, and it also received substantial work to allow it to handle objects, broadening its inference capabilities to cover Probabilistic Relational Models. Similarly, PIPE was heavily adapted with the extensions in §2.2.1; PIPE itself can handle only Petri nets with ordinary, test, and inhibitory arcs.

Of course, there was major implementation effort in connecting the components. Neither package was originally designed to work with the other, and they had to be very intimately connected to each other and also to ATAN as a bridge to the RoboCup world. The core of this interaction was provided by adding alternate types of Petri net places and Bayes net random variables that are actually links to the other representation. That way, whenever the Petri net needs the value of a place that is equivalent to a random variable, it automatically queries it, and similarly for the Bayes net's access of place-equivalent random variables.

In order to make the resulting software easy to use, the graphical user interfaces were also adapted. Because they were built using different user interface toolkits, they could not be completely integrated. Instead, the user interface consists of two separate windows. In one, the user may modify the Petri net structure; in the other, the user may affect the probabilistic relational model. This interface was used to construct all the example behaviors.

It should be noted that as a preliminary effort to the complete implementation, a simpler version with hybrid (discrete/continuous) Petri nets and Bayes nets was created work with Alberto Amengual [4]. This simpler tool was used to model the behavior, particularly attachment behavior, of children in a psychological experiment. This initial work was an important stepping-stone to the complete architecture.

5.2 Demonstration

In order to demonstrate both the ideas of this architecture and its implementation, I have produced real behaviors using it. In this dissertation, I have presented three examples in the RoboCup robotic soccer domain. They form a series increasing in complexity, with each showcasing a particular aspect of the architecture. Although it is difficult in a text like this to adequately describe the effectiveness of these demonstrations, recordings of the example behaviors' performance are available at http://soccer.barrettnexus.com/demo.

The first demonstration, in §2.2.7, showcases the use of Petri nets to structure action. It consists of an agent that passes and receives the ball, dribbles the ball, and shoots on the goal. It does not use the probabilistic relational model components of §2.3, nor does it use the recovery machinery of §4.1. Instead, it forms a structured behavior by using the Petri net's internal state to use past inputs to affect current actions. By splitting the network into modules, it demonstrates a hierarchy of actions, which keeps the states for various sub-activities compartmentalized into related chunks. It also uses the procedural structure to track what goals and actions it should pursue at any time.

The second demonstration, shown in §2.3.5, focuses on the added abilities of the probabilistic relational model. This example is the behavior for a goalie who watches the ball alert for danger and, when the ball is incoming, moves to block it. Again, it contains a modular procedural structure, but now it also relies on its inference mechanisms to combine its noisy sensor data into inference about hidden parts of the world. For instance, it tracks the ball and its opponents, identifies when an opponent possesses the ball, and uses that information to predict the future position of the ball.

The third and final demonstration, in §4.3, extends all of these ingredients by highlighting the recovery system of this architecture. This complex multi-agent behavior passes the ball back and forth, advancing down the field until in range to shoot on the goal. Despite its use of the probabilistic relational model, the noise in the world can make precisely passing and stopping the ball difficult, so there are many possible errors that could occur. This behavior uses the recovery machinery of §4.1 to detect these error conditions and move to safe states, from which it can continue its advance on the enemy goal.

5.3 Baseline Comparisons

In order to show the value of this architecture, we require comparisons against other systems, or if there are no other systems in quite the same domain, at least comparisons against baselines. By showing that our architecture is at least as capable as the competing systems on the test domain of RoboCup, we could establish its value. However, this architecture targets a new problem and thus falls in the latter category, so we must examine possible baselines. I argue that the reasonable baselines to compare against are either subsumed within this architecture or fail to possess some critical advantages of this architecture.

There are several baselines for useful behaviors, such as those activities needed in the RoboCup domain, that might be relevant. For instance, we might represent structured behavior with finite state machines [33], or we might use concurrently operating finite state machines, such as those of the subsumption architecture [11, 12]. On the other hand, to better exploit an understanding of the environment, an alternate baseline is to use a Bayesian network to infer world state and map that state to output actions [62]. Finally, we might imagine not a simplified baseline, but a complex one: a parallel computer language.

Each of these baselines is particularly relevant because of its relationship to the architecture presented here. However, those relationships also make it possible to see clearly the similarities and differences in abilities by inspection and argument alone. Here, I will discuss those relationships.

5.3.1 Finite State Machine Baseline

One main requirement of a useful behavior is to produce appropriately structured actions. The naive way to create the structure of an action would be to create a finite state machine [33] for it. One could specify outputs along arcs, and inputs would change which arcs are possible from each state. With this, it would not be too difficult to script out some simple RoboCup soccer agent behaviors. Thus, the first baseline to consider is that of a finite state machine.

However, such a simple finite state machine model is a vast oversimplification. In fact, it is completely subsumed by the architecture described in Chapter 2. In particular, a Petri net can depict a finite state machine [58]. All that is needed is to have only one input arc and one output arc (both with a weight of 1) per transition, and to initialize the state with a single token. Then, that Petri net is exactly equivalent to a finite state machine, with states of the FSM being identical to places in the PN. So, the architecture I present here completely contains the finite state machine baseline.

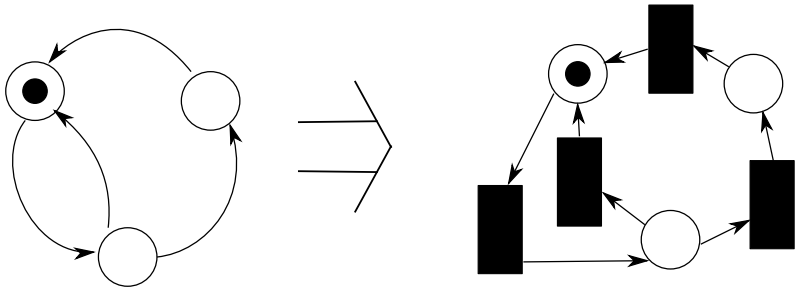


Figure 5.1: A finite state machine is equivalent to a Petri net where each transition has one input and one output arc.

Over-representing another tool does not imply superiority. After all, important properties may be lost with increased complexity. For instance, a Turing machine subsumes finite state machines, but cannot be analyzed as can PNs and FSMs. Considering that, the finite state machine is still inferior for our task—it cannot be analyzed significantly more effectively than a Petri net.

Also, the FSM baseline can only describe concurrent activities by giving the full cross-product state space of both together. Now, that particular disadvantage of lacking concurrency has been addressed before. Specifically, the subsumption architecture has used multiple interacting concurrent finite state machines [11, 12]. However, even this does not fix all the flaws. First of all, that is a noticeably weak form of concurrency—after all, this is essentially a less-principled sort of Petri net, and is again subsumed by, and less analyzable than, the Petri net.

Furthermore, this purely procedural representation lacks any understanding of its environmental. In order to use previous observations, which may be important clues to unobserved components of the environment, to affect future actions, it is necessary to store that information by keeping the FSM in an entirely different state until that information is needed. The result is again an explosion of state space, just as occurred when trying to cram concurrency into a single FSM. Only this quite crude method of shoehorning all the various considerations of past actions, current actions, and hidden state into the single state value can allow finite state machines to produce anything close to the required complexity of actions. As a result, a concurrent finite state machine baseline is also inferior to our method for the demands of a complex environment.

5.3.2 Bayesian Network Baseline

If ignoring the inference required to handle a real environment is a fatal flaw of the finite state machine, perhaps an inference-centric baseline could be more useful. After all, knowing what

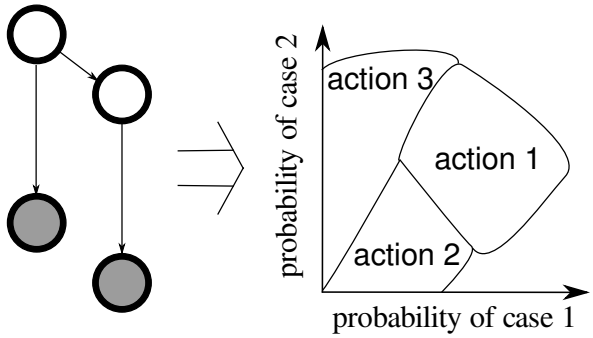


Figure 5.2: A Bayesian network baseline maps its inferred distributions into actions.

is going on in the world should be sufficient to choose actions. We could just use a Bayes net to infer the hidden world state, and then we can map that state into output actions [62]. That is, divide the possible inferred distributions over world states, which form a multidimensional continuous state space, into regions. Each region corresponds to the emission of an atomic action, as shown in Figure 5.2, and other measures over distributions could give parameters for those actions. This is rather similar to the solution descriptions for partially observable Markov decision problems (POMDPs) [42, 34]. Note further that if the action regions overlap, then multiple actions could be output at once, allowing for concurrency.

This baseline has a number of disadvantages, again beginning with the fact that it is subsumed by my architecture. You could produce exactly the same thing by taking the representation from Chapter 2 and simplifying the Petri net. Let the regions of distribution space be given by combinations of places corresponding to random variables; their numbers of tokens are given by functions of the variables' distributions, and they may be combined via math transitions to precisely define arbitrary regions of distribution space. Thus, we can make an exact equivalent of the baseline in my architecture, as seen in Figure 5.3.

This baseline is still missing crucial elements, though. That is clearly suggested by the fact that it uses only a few of the components described in this dissertation, but let us examine its flaws in more detail. First, it may not be able to take swift action. In order to determine what action to take, it must perform inference to establish what part of distribution space it is in. In a complex environment, inference may be slow, and many real-world situations require near-instantaneous actions. After all, humans use reflexes to make important protective reactions when our cognitive processes may be too slow.

Also, by neglecting internal state, a baseline Bayes net agent will not be able to track its own intentions and goals. For instance, on an outing into the world, we might find ourselves

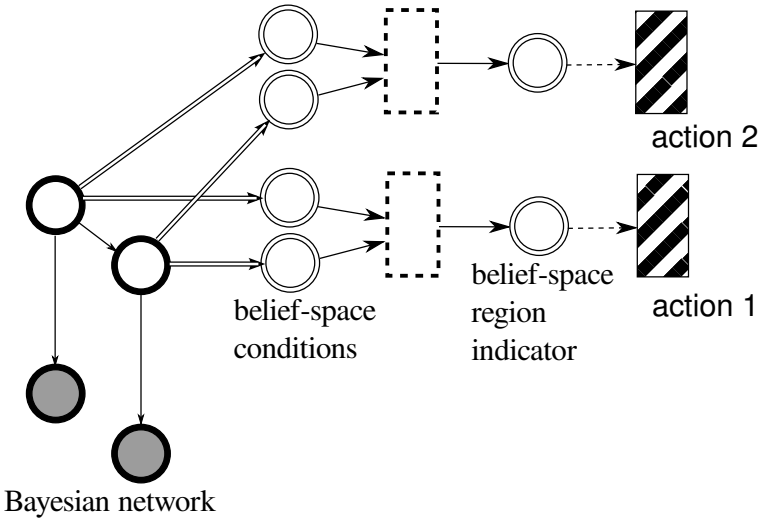


Figure 5.3: The Bayesian network baseline that maps belief-space regions to actions can be written in my architecture.

in identical world states more than once. We might hike out to see a view, and on our return retrace our steps. When returning, the world state is essentially identical, but we want to move in a diametrically opposite direction than we did previously. So, we cannot simply base our activity choices on the world alone.

Note that this second objection might be answered by including random variables to encode the agent's internal state. These internal random variables would exist only to reflect the agent's plans and goals. However, instead of letting the plans and goals relate and coordinate only when relevant, they must be related constantly and inferred at all times. Furthermore, this approach actually worsens the issue of reaction speed; now, the agent must use slow inference to figure out even what its own state is.

5.3.3 Concurrent Turing-Complete Language Baseline

Finally, though the above baselines are somewhat simple, we might also compare to a quite complicated option. If our own architecture can do fancy things, we might want to compare against a fancy baseline. The sensible such baseline would be arbitrary concurrent computer code. After all, it is certainly capable of describing complex coordinated actions. Indeed, there are no activities this architecture can represent that code cannot; it subsumes this architecture so fully that the architecture itself was completely implemented in computer code. Thus, it is not hard to see that such code must be able to perform actions at least as complex and coordinated as my

own representation. From this, we might conclude that my architecture, being subsumed, is not worthwhile.

However, this neglects one significant fact: representation alone is not enough. Although arbitrary computer code is capable of representing many things and producing many actions, it is very difficult to analyze. Therefore, the set of analysis methods for a Turing-complete language will be much smaller than those for my architecture. Because we lack methods for guaranteeing reachability and an absence of deadlocks, we cannot use the Markov chain analysis methods described in §3.2. We can still use forward sampling methods, as in §3.3, but our selection of possible starting states will be more difficult, as it may be hard to say what states the concurrent code could be in at the beginning of an activity. Furthermore, since we cannot determine which previous internal states might have led to the current state, we also cannot use the backward analysis of §3.3.1 to determine likely causes of errors. The result is that although the representation ability of computer code is unrivalled, the lack of useful analysis methods is a serious handicap when it comes to making the guarantees needed for artificial agents to operate in certain critical environments. Therefore, even this extremely capable baseline representation is not an appropriate comparison.

Having reviewed a set of the sensible baselines, we can see that a side-by-side comparison of performance metrics is not likely to be useful. Although performance on certain tasks may be higher with some representations than others, the abilities provided by this architecture fill an important domain that is not otherwise covered. By carefully matching my architecture to the demands of real-world actions, I have obtained a result that can perform complex activities in complex environmental conditions while also permitting a great deal of analysis and proof methods. This middle ground is the essential one for coordinated artificial agents.

Chapter 6

Conclusion

This work is an attempt to fix one of the largest outstanding gaps in the field of Artificial Intelligence. There is a huge class of problems that are easy for humans and difficult for computers or robots, and many of these problems revolve around interactions with the real world. Specifically, any kind of complex task that requires an agent to react to changing conditions rather than merely replaying recorded actions, that requires simultaneous control of multiple effectors, or that has a state space too large to search through, is often intractable or nearly intractable to computerized agents.

Unfortunately for our robots, the real world is replete with these tricky problems. They range from the mundane, such as cooking an egg or hulling a strawberry, to the exotic and critical, like performing emergency surgery or fleeing a burning building. Fortunately for us, humans are pretty good at accomplishing such complex tasks. Why, many humans are even capable of walking and chewing gum at the same time. However, we have not been able to transfer these abilities to our mechanical contrivances.

Tackling this problem, like most problems in computer science, requires a new way of representing our data. Just as some data structures are highly useful for some tasks and terrible for others (hint: avoid binary search in linked lists), this problem may only be tractable when matched with an appropriate representation. Such a representation would need to capture commonalities about complex actions in its very structure. Therefore, my work here has been to develop, describe, and demonstrate just such a specialized representation, plus a set of algorithms enabled by the representation, together forming a new architecture for actions. Demonstrations were performed using the RoboCup [43] robotic soccer simulator.

In this work, as described in Chapter 2, I tackled that problem by combining two separate representations, each optimized for a different task. In order to describe the structure of actions,

using internal state and producing quick reactions, I used a procedural model. To interpret noisy and tricky inputs into guesses about the state of the world, I included a model for probabilistic inference. These two components were then combined and extended, so that together they provide a great deal of leverage to move us forward on complex actions.

In choosing a procedural model, it was necessary to walk a fine line. On the one hand, a simple model like the finite state machine [33] is not able to handle the complex demands of the real world. On the other hand, a very complex model, such as a parallel computer language, can represent what we need but cannot be analyzed, making it very difficult to give useful guarantees about the effectiveness of actions. The ideal middle ground is the Petri net [63], described in §2.2.1. This is a standard tool for analyzing concurrent processes, modified here with additions to make it deal better with the interactive, timed, and continuous aspects of the world. It uses markings moving around a bipartite graph to describe the progress of an activity, with links and transitions to describe the concurrent or coordinated steps that may be taken. The result is a procedural model that can describe quite complex actions and is demonstrated via a RoboCup soccer behavior in §2.2.7.

The second component chosen, the inferential model, was the Probabilistic Relational Model [46], an object-centered extension of the well-known Bayesian network [62]. It is superb at describing the evolution of random processes in the world, usign probabilistic information about objects with possibly changing relationships. It describes the world as a set of random variables attached to objects; if a relationship connects a set of objects, then their variables may be interrelated. The dependencies between variables, combined with evidence for the values of some variables, describe a probability distribution over all the variables. We can then use probabilistic inference [38, 47, 75, 21] to query estimated distributions over the values of some unobserved variables. Thus, combined with the Petri net, the result is a combined representation capable of describing both internal progress through an activity and external estimates of the world and the agent's affect on the world. This addition to the Petri net was demonstrated in §2.3.5.

Unfortunately, as excellent as these two components are, they do not naturally fit together extremely well. Thus, some work was required to get them to cooperate. This was done by having each component view the other in its own terms. To the probabilistic relational model, a Petri net place is viewed as an observed random variable, which can then have interdependencies on other variables and thus affect inferred distributions. In this way, an agent can use its knowledge of its own state and actions to predict effects upon the world. Similarly, the distribution of a PRM random variable can be mapped into a marking of a place, allowing the Petri net to control its actions based on its estimate of the hidden state of the world. Thus, the two components help

each other accomplish their tasks.

In Chapter 3, I described algorithms that take advantage of the structure of the behavioral representation to analyze behaviors for effectiveness and safety. Because both of the representational components are designed for analysis of just this sort, the combined representation is also amenable to analysis. There are the usual limitations, of course—if the environment contains a terribly complex process, then analyzing an agent that interacts with it may be as difficult as analyzing the terribly complex process or at least require some approximation and simplification. After all, analyzing an agent in a Turing-complete environment could require solving the halting problem. However, given those limitations, I presented a broad spectrum of analysis methods.

These methods cover a range of precision and complexity, making them appropriate in different conditions. On the one hand, the behavior can be treated as a Markov model. This is an exact method, but it may be useless in the face of complex behaviors in difficult environments. At the other extreme lie sampling methods, which can be used even in the face of the worst complexity, but which also provide weaker guarantees. I also described tricks to mitigate these methods' issues. For instance, there is much to gain by analyzing individual modules, as then these analyses can be used as summaries to simplify higher-level analysis. Similarly, sampling methods can be used backwards in time to identify the sources of error conditions, rather than simply simulating forward until an error occurs. These techniques help make guarantees of behavior quality feasible.

In Chapter 4, I explained how this framework allows recovery from unexpected interruptions. When an agent detects an error, it passes that information up or down the action hierarchy to an appropriate position to handle it. Then, the agent can retreat to a safe state, which, depending on the environment, may be dynamic (for instance, a bicycle's stability issues may not permit static safe states when cycling). From there, it can resume progress toward its goals. The representation of such recovery is simplified by further extension that allows the direct setting of place markings in the Petri net. The utility of this recovery mechanism was demonstrated by another RoboCup behavior in §4.3.

Finally, in Chapter 5, I summarized the implementation and final results of this work. This architecture is more capable than the relevant baselines, besting each in representational ability or in analyzability. Also, it has been demonstrated with a set of example behaviors in the RoboCup domain. All this serves to display the value of my architecture.

6.1 Future Work

Unsurprisingly, this thesis has not taken this work to all possible logical conclusions. Here are some interesting future directions.

6.1.1 Reasoning and Planning

Reasoning about the world is a difficult problem, and many attempts have been made at it. Logical reasoning [64] and probabilistic inference [59] are among the most successful of these, and they can explain much of the reasoning humans do. However, they do not completely explain how humans reason. In particular, they are not very good at explaining how we understand complex events and activities. Because we encounter such complexities so often, it is critical to be able to reason about them. Since the action representation discussed here is designed for describing complex activities, it should have good application to this.

One method is particularly interesting. It is based on an idea that human understanding and reasoning work via mental simulation [25, 18]. Somehow, we imagine the situation playing out in front of us, or we "put ourselves in someone's shoes," and by picturing how the world evolves, we can draw inferences and make predictions. It is easy to see how that could be applied to this model. After all, we can imagine a situation (by setting up the state of the model to match that of the situation) and then simulate that model forward. We can move the Petri net forward in the usual way, and we can move the Bayes net forward by either sampling or by keeping a more complex distribution model, such as a set of the most likely states.

Similarly, planning can be done by considering the effects of forward simulation. We can see what outcomes are likely if we try various actions, and base our choice of those actions on the predicted results. Also, experience might teach us which actions are worth investigating, allowing us to spend our planning time more effectively.

6.1.2 Learning

Throughout this dissertation, I have almost entirely presumed that behaviors in this representation are either human-designed or are given to us fully complete from some mysterious source. However, it is an intriguing problem to consider how they might be learned. There are already learning methods for many of the pieces. For instance, a dynamic Bayes net model of the environment may be learned [59, 29]. Furthermore, it is possible to learn finite state machines to navigate well through a partially observable Markov decision process [55], though it is unclear how to extend this work to learning Petri nets. There are techniques for learning Petri nets from state

graphs [22, 13], so it might be possible to tighten a learned FSM into a Petri net, but it is unclear how to directly learn a Petri net to solve a POMDP.

Hierarchical planning [52, 73, 74, 16] and reinforcement learning [49, 71, 50, 35, 36], as discussed in §1.2, are also good candidates for learning useful behaviors. After all, a behavior in the representation described here may be seen as a compiled policy or set of plans. Based on relevant observations, the agent uses its policy or chooses which plan to use. It can do so at every level of the hierarchy, so if a behavioral module corresponds to a plan, then having a hierarchy of behaviors is similar to having a planning hierarchy. It does not matter precisely how the hierarchical policy or plans are found, just that the plans at each level are broken down into successively finer plans below. I suspect this sort of behavioral learning would be a very interesting possible research direction.

Another direction for tackling this learning problem would be from the angle of observing other actions. Just as reinforcement learning can accomplish amazing things by observing an already-trained agent [1], these behaviors might be usefully learned by seeing the structure of another agent's actions. Here, techniques on learning Petri nets from observed states might be highly applicable [22, 13]. Or, we could view this problem in the other direction. Supposing another agent is learning to perform a task by observing our actions; how, then, can we make it easy for them to determine our behavior's structure?

6.1.3 Improved Inference Flexibility

As mentioned in §2.3.2, there are other inference methods beyond the exact ones like variable elimination and the Junction Tree algorithm. For instance, particle filtering is one such method that uses sampling to allow a more detailed tradeoff between inference accuracy and time. There are other such methods, such as "anytime algorithms," which can stop inference early and still get a useful approximate answer [48, 53]. Adding such inference tools to this architecture would make it more flexible.

6.1.4 Language

This work was motivated in part by a desire to ground language and thought in real actions in a complex environment. However, despite its success in handling actions in the world, I have not connected it to any sort of linguistic ability. It might be highly enlightening to relate this architecture to linguistics, perhaps by connecting it to the semantics of FrameNet [27]. It might be similarly interesting to use these properly grounded actions for metaphorical reasoning, as was

done in [60].

This architecture could also be useful for more complex linguistic problems. It is not uncommon for linguistic communication to be interrupted; indeed, speakers will often interrupt themselves with side issues and temporary concerns. The methods described here may provide leverage at dealing with concurrency and interruption even in linguistic acts.

6.1.5 Modeling Internal State of Another Agent

One common problem in multi-agent settings is that in order to make useful predictions or plans, it is necessary to consider one's partners and opponents. As this architecture is designed for describing behaviors, it could also describe the behaviors of an agent other than oneself. This gives rise to a set of other questions. How can we determine the structure of someone else's actions? Given a structure, how might we estimate the current internal state of another agent? How can we manipulate another agent's state to obtain a benefit, mutual or individual? How might we conceal our own internal state while still accomplishing our goals? These problems become trickier also because we have to guess at the other agent's inputs.

6.2 Conclusion

My thesis describes a full framework for complex actions, including both an action-production-centric representation and a set of algorithms to analyze the effects such behaviors. Rather than focusing on a single behavior, I have attempted to capture commonalities from across the space of human activity. Giving computers access to these fundamental aspects of the way humans deal with the world is, I believe, a crucial step toward creating agents as capable as humans at interacting with, and understanding, the world.

Bibliography

[1] Pieter Abbeel. *Apprenticeship Learning and Reinforcement Learning with Application to Robotic Control*. PhD thesis, Stanford University, August 2008.

[2] Philip E. Agre and David Chapman. Pengi: An implementation of a theory of activity. In *Proceedings of AAAI-87*, pages 268–272, 1987.

[3] Alan Agresti and Brent A. Coull. Approximate is better than "exact" for interval estimation of binomial proportions. *The American Statistician*, 52, 1998.

[4] Alberto Amengual. A computational model of attachment secure responses in the strange situation. Technical Report TR-09-002, International Computer Science Institute, March 2009.

[5] Corin R. Anderson, Pedro Domingos, and Daniel Weld. Relational Markov models and their application to adaptive web navigation. In *Proceedings KDD-2002*, 2002.

[6] Adam M. Johansen Arnaud Doucet. A tutorial on particle filtering and smoothing: Fifteen years later. In D. Crisan and B. Rozovsky, editors, *Handbook of Nonlinear Filtering*. Oxford University Press, 2009.

[7] Leon Barrett and Srini Narayanan. Learning all optimal policies with multiple criteria. In *ICML*, July 2008.

[8] Pere Bonet, Catalina M. Lladó, Ramon Puijaner, and William J. Knottenbelt. Platform independent Petri net editor. http://pipe2.sourceforge.net/.

[9] Pere Bonet, Catalina M. Lladó, Ramon Puijaner, and William J. Knottenbelt. PIPE v2.5: A Petri net tool for performance modelling. In *Proceedings of the 23rd Latin American Conference on Informatics (CLEI 2007)*, San Jose, Costa Rica, October 2007.

[10] Taylor L. Booth. *Sequential Machines and Automata Theory*. John Wiley and Sons, Inc., New York, first edition, 1967.

[11] R. A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal Of Robotics And Automation*, RA-2:14–23, April 1986.

[12] Rodney Brooks. Intelligence without representation. *Artificial Intelligence*, 47:139–159, 1991.

[13] Nadia Busi and G. Michele Pinna. Synthesis of nets with inhibitor arcs. In *International Conference on Concurrency Theory*, pages 151–165, 1997.

[14] Mao Chen, Ehsan Foroughi, Fredrik Heintz, ZhanXiang Huang, Spiros Kapetanakis, Kostas Kostiadis, Johan Kummeneje, Itsuki Noda, Oliver Obst, Pat Riley, Timo Steffens, Yi Wang, and Xiang Yin. RoboCup soccer server users manual, August 2002.

[15] Wai-Ki Ching and Michael K. Ng. *Markov Chains: Models, Algorithms and Applications*. Springer, New York, 2006.

[16] Bradley J. Clement, Edmund H. Durfee, and Anthony C. Barrett. Abstract reasoning for planning and coordination. *Journal of Artificial Intelligence Research*, 28:453–515, 2007.

[17] René David and Hassane Alla. On hybrid Petri nets. *Discrete Event Dynamic Systems*, 11(1-2):9–40, 2001.

[18] Martin Davies and Tony Stone. *Folk Psychology and Mental Simulation*. Number 43 in Royal Institute of Philosophy Supplements. Cambridge University Press, 1998.

[19] Rodrigo de Salvo Braz, Eyal Amir, and Dan Roth. Lifted first-order probabilistic inference. In Lise Getoor and Ben Taskar, editors, *Introduction to Statistical Relational Learning*. The MIT Press, 2007.

[20] Rodrigo de Salvo Braz, Sriraam Natarajan, Hung Bui, Jude Shavlik, and Stuart Russell. Anytime lifted belief propagation. In *Workshop on Statistical Relational Learning*, 2009.

[21] Rina Dechter. Bucket elimination: A unifying framework for probabilistic inference. *Uncertainty in Artificial Intelligence*, pages 211–219, 1996.

[22] Jorg Desel and Wolfgang Reisig. The synthesis problem of Petri nets. In *STACS 93*, pages 120–129, 1993.

[23] Jörg Desel and Wolfgang Reisig. *Place/transition Petri Nets*, volume 1491/1998 of *Lecture Notes in Computer Science*, pages 122–173. Springer Berlin / Heidelberg, 1998.

[24] Arnaud Doucet, Nando de Freitas, and Neil Gordon, editors. *Sequential Monte Carlo Methods in Practice*. Springer-Verlag, 2001.

[25] Jerome A. Feldman. *From Molecule to Metaphor: A Neural Theory of Language*. The MIT Press, Cambridge, Massachusetts, 2006.

[26] Richard Fikes and Nils Nilsson. STRIPS: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[27] Charles J. Fillmore and Collin F. Baker. Frame semantics for text understanding. In *Proceedings of WordNet and Other Lexical Resources Workshop, NAACL*, Pittsburgh, June 2001.

[28] Nir Friedman, Lise Getoor, Daphne Koller, and Avi Pfeffer. Learning probabilistic relational models. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1300–1307, Stockholm, Sweden, August 1999.

[29] Song Gao, Qinkun Xiao, Quan Pan, and Qingguo Li. *Learning Dynamic Bayesian Networks Structure Based on Bayesian Optimization Algorithm*, volume 4492/2007. Springer Berlin / Heidelberg, 2007.

[30] Simon J. Godsill, Arnaud Doucet, and Mike West. Monte Carlo smoothing for nonlinear time series. *Journal of the American Statistical Association*, 99(465):156–168, March 2004.

[31] Peter J. Haas. *Stochastic Petri Nets*. Springer Series in Operations Research and Financial Engineering. Springer, 2002.

[32] Eric A. Hansen. Solving POMDPs by searching in policy space. In *Proceedings of the Fourteenth International Conference on Uncertainty In Artificial Intelligence (UAI-98)*, pages 211–219, 1998.

[33] Michael A. Harrison. *Introduction to Switching and Automata Theory*. McGraw-Hill, New York, 1965.

[34] M. Hauskrecht. *Planning and Control in Stochastic Domains with Imperfect Information*. PhD thesis, MIT, Cambridge, MA, 1997.

[35] Bernhard Hengst. Discovering hierarchy in reinforcement learning with HEXQ. In *ICML*, pages 243–250, 2002.

[36] Bernhard Hengst. *Partial Order Hierarchical Reinforcement Learning*, volume 5360/2008 of *Lecture Notes in Computer Science*, pages 138–149. Springer Berlin / Heidelberg, 2008.

[37] William H. Hsu. Bayesian network tools in java. http://bnj.sourceforge.net/.

[38] Cecil Huang and Adnan Darwiche. Inference in belief networks: A procedural guide. *International Journal of Approximate Reasoning*, 15(3):225–263, October 1996.

[39] Nick James and Wolfgang Wagner. ATAN. http://atan1.sourceforge.net/.

[40] Michael I. Jordan, Zoubin Ghahramani, Tommi S. Jaakkola, and Lawrence K. Saul. An introduction to variational methods for graphical models. *Machine Learning*, 37(2):183–233, November 1999.

[41] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[42] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101:99–134, 1998.

[43] Hiroaki Kitano, Milind Tambe, Peter Stone, Manuela Veloso, Silvia Coradeschi, Eiichi Osawa, Hitoshi Matsubara, Itsuki Noda, and Minoru Asada. The RoboCup synthetic agent challenge. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI97)*, San Francisco, CA, 1997. Morgan Kaufmann.

[44] Hiroaki Kitano, Milind Tambe, Peter Stone, Manuela Veloso, Silvia Coradeschi, Eiichi Osawa, Hitoshi Matsubara, Itsuki Noda, and Minoru Asada. The RoboCup synthetic agent challenge, 1997. In *International Joint Conference on Artificial Intelligence (IJCAI97)*, 1997.

[45] Daphne Koller and Avi Pfeffer. Object-oriented Bayesian networks. In *Proceedings of the 13th Annual Conference on Uncertainty in AI (UAI)*, pages 302–313, Providence, Rhode Island, August 1997.

[46] Daphne Koller and Avi Pfeffer. Probabilistic frame-based systems. In *Proceedings of AAAI*, 1998.

[47] Steffen L. Lauritzen and David David J. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society*, 50(2):157–224, 1988.

[48] Chao-Lin Liu and Michael P. Wellman. On state-space abstraction for anytime evaluation of Bayesian networks. *SIGART Bulletin*, 7(2):50–57, 1996.

[49] Bhaskara Marthi. *Concurrent Hierarchical Reinforcement Learning*. PhD thesis, UC Berkeley, 2006.

[50] Bhaskara Marthi, Lelsie Kaelbling, and Tomas Lozano-Perez. Learning hierarchical structure in policies. In *NIPS 2007 Workshop on Hierarchical Organization of Behavior*, 2007.

[51] Bhaskara Marthi, Stuart Russell, and Jason Wolfe. Angelic Semantics for High-Level Actions. In *ICAPS*, 2007.

[52] Bhaskara Marthi, Stuart Russell, and Jason Wolfe. Angelic hierarchical planning: Optimal and online algorithms. In *Proceedings of ICAPS*, 2008.

[53] Robert Mateescu, Rina Dechter, and Kalev Kask. Tree approximation for belief updating. In *Eighteenth national conference on Artificial intelligence (AAAI-02)*, pages 553–559, 2002.

[54] Nicolas Meuleau, Kee-Eung Kim, Leslie Kaelbling, and Anthony Cassandra. Solving POMDPs by searching the space of finite policies. In *Proceedings of the Conf. on Uncertainty in AI*, 1999.

[55] Nicolas Meuleau, Leonid Peshkin, Kee-Eung Kim, and Leslie Kaelbling. Learning finite-state controllers for partially observable environments. In *Proceedings of the Conf. on Uncertainty in AI*, 1999.

[56] Brian Milch, Bhaskara Marthi, Stuart Russell, David Sontag, Daniel L. Ong, and Andrey Kolobov. BLOG: Probabilistic models with unknown objects. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1352–1359, 2005.

[57] Brian Milch, Luke S. Zettlemoyer, Kristian Kersting, Michael Haimes, and Leslie Pack Kaelbling. Lifted probabilistic inference with counting formulas. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence*, pages 1062–1068, 2008.

[58] Tadao Murata. Petri nets: Properties, analysis, and applications. *Proceedings of the IEEE*, 77(4), April 1989.

[59] Kevin Murphy. *Dynamic Bayesian Networks: Representation, Inference and Learning*. PhD thesis, UC Berkeley, Computer Science Division, July 2002.

[60] Srini Narayanan. *KARMA: Knowledge-based Action Representations for Metaphor and Aspect*. PhD thesis, UC Berkeley, Berkeley, CA, 1997.

[61] Srini Narayanan. Reasoning about actions in narrative understanding. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99)*. Morgan Kaufmann Press, 1999.

[62] Judea Pearl. Bayesian networks: A model of self-activated memory for evidential reasoning. In *Proceedings of the 7th Conference of the Cognitive Science Society*, pages 329–334. University of California, Irvine, CA, 1985.

[63] Carl A. Petri. *Kommunikation mit Automaten*. PhD thesis, Bonn: Institut fur Instrumentelle Mathematik, 1962.

[64] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Englewood Cliffs, New Jersey, 2 edition, 2003.

[65] Sumit Sanghai, Pedro Domingos, and Daniel Weld. Dynamic probabilistic relational models. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, pages 992–997, Acapulco, Mexico, 2003.

[66] Shankar Sastry. *Nonlinear Systems: Analysis, Stability, and Control*. Springer Science+Business Media, New York, NY, 1999.

[67] Shankar Sastry. *Nonlinear Systems: Analysis, Stability, and Control*. Springer Verlag, 1999.

[68] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, Cambridge, Massachusetts, 1998.

[69] Fengzhan Tian and Yuchang Lu. A DBN inference algorithm using junction tree. In *Proceedings of the 5th World Congress on Intelligent Control and Automation*, Hangzhou, China, June 2004.

[70] Jiacun Wang. *Timed Petri Nets: Theory and Application*. Kluwer Academic Publishers, Norwell, Massachusetts, 1998.

[71] Shimon Whiteson and Peter Stone. Concurrent layered learning. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multi-Agent Systems*, Melbourne, Australia, July 2003.

[72] Edwin Bidwell Wilson. Probable inference, the law of succession, and statistical inference. *Journal of the American Statistical Association*, 22:209–212, 1927.

[73] Jason Wolfe, Bhaskara Marthi, and Stuart Russell. Combined task and motion planning for mobile manipulation. In *Proceedings of ICAPS*, 2010.

[74] Qiang Yang. Formalizing planning knowledge for hierarchical planning. *Computational Intelligence*, 6(1):12–24, 1990.

[75] Nevin Lianwen Zhang and David Poole. Exploiting causal independence in Bayesian network inference. *Journal of Artificial Intelligence Research*, 5:301–328, 1996.

Appendix A

Hand-Wrought Analysis

To show how hand-analysis is used to guarantee the success of a behavior, we will now see its use on an example behavior, designed to operate on the RoboCup soccer environment. Consider a robotic soccer task, where a robot must go to a stationary soccer ball. In particular, the environment, precisely as described by the RoboCup Soccer Simulator [44], is as follows:

1. The environment operates in discrete time steps.
2. The environment is described by six variables:
   - x and y, the agent's position
   - s, the agent's speed
   - ρ, the direction in which the agent is facing and moving
   - x^* and y^*, the position of the ball.
3. These also permit us to compute two auxiliary variables:
   - d, the Euclidean distance to the ball.
   - θ, the direction to the ball relative to the agent's orientation.
4. The goal is to reduce the distance to the ball below a threshold. Therefore, we also define the error metric to be d, and the goal will be reached only when $d < \epsilon_d$.
5. The agent may influence the environment with two inputs:
   - a, an acceleration that changes only the agent's speed s, bounded in the range $[a_{\min}, a_{\max}]$ with $a_{\max} > 0 > a_{\min}$.

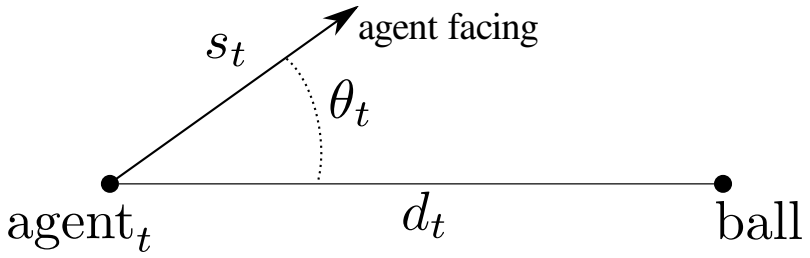


Figure A.1: A simple version of the soccer environment

- ω, a rotation that changes only the agent's orientation θ, in the range $[-\theta_{\max}, \theta_{\max}]$, with $\theta_{\max} > 0$.

Only one of these inputs may be applied in any time step. That is, $\forall_t (a_t = 0) \vee (\omega_t = 0)$.

6. The environment is updated according to:

$$\begin{aligned} \rho_{t+1} &= \rho_t + \omega_{t+1}(1 + \xi_\omega) \\ s_{t+1} &= \gamma s_t + a_{t+1}(1 + \xi_a) \\ x_{t+1} &= x_t + s_{t+1}(\cos(\rho_{t+1}) + \xi_x) \\ y_{t+1} &= y_t + s_{t+1}(\sin(\rho_{t+1}) + \xi_y) \end{aligned}$$

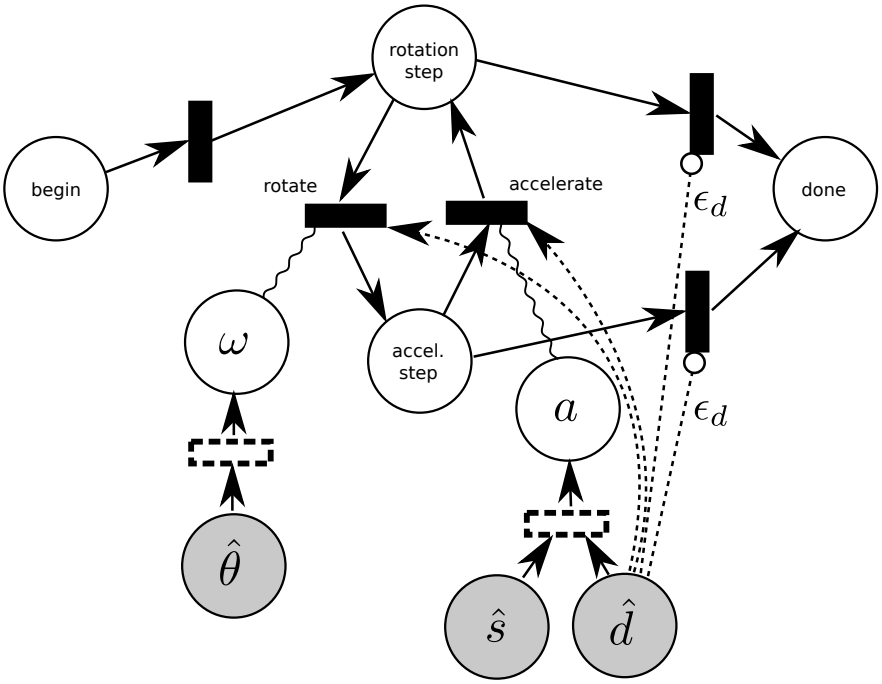
Here, the environment injects random errors, labeled ξ_* (where $*$ is any of the above variables), that are distributed uniformly in the ranges $[-\delta_*, \delta_*]$, except that ξ_x and ξ_y are in the range $[-\frac{1}{\sqrt{2}}\delta_s, \frac{1}{\sqrt{2}}\delta_s]$.

7. The agent receives the following observations:

$$\begin{aligned} \hat{\theta}_t &= \theta_t + \hat{\xi}_\theta \\ \hat{s}_t &= s_t(1 + \hat{\xi}_s) \\ \hat{d}_t &= d_t(1 + \hat{\xi}_d) \end{aligned}$$

Again, the environment injects random errors into the observations, labeled $\hat{\xi}_*$ (where $*$ could be any of the above variables), which are distributed uniformly within the ranges $[-\hat{\delta}_*, \hat{\delta}_*]$.

We may summarize this environment by ignoring x, y, x^*, and y^*, resulting in a 3-variable description consisting of d, s, and θ, as shown in Figure A.1. (After all, these are the only variables about which the agent receives input.) It is difficult to write equations for the updates of these variables, but it is possible to find bounds on these variables, and that is what we require.



$$\begin{aligned} \omega_{t+1} &= -\hat{\theta}_t \\ a_{t+1} &= \frac{1-\gamma}{(1+\delta_s)(1+\delta_a)}\left(\frac{1}{1+\hat{\delta}_d}\hat{d}_t - \frac{1}{1-\hat{\delta}_s}\frac{\gamma}{1-\gamma}(1+\delta_s)\hat{s}_t\right) \end{aligned}$$

Figure A.2: "Slow": A behavior to close with a stationary ball.

A.1 Proving Bounds: Behavior

In order to analyze a behavior, we must have a behavior to analyze. Consider the behavior shown in Figure 3.1. We will analyze the sub-component "Slow" shown in Figure A.2.

A.2 Proving Bounds: Bounds

The goal, again, is to reduce d below ϵ_d, so the natural error metric is simply d. However, because there is a momentum factor γ and limited ranges of inputs, there are times when it is inevitable that d increases. For instance, if the agent is moving quickly away ($\theta_t = \pi$, $\omega_{\min} > -\frac{\pi}{2}$, $\omega_{\max} < \frac{\pi}{2}$, and $\gamma s_t > -a_{\min}$), then it cannot turn or slow sufficiently to reverse course, so $d_{t+1} > d_t$. To avoid this sort of increase in error, we will first define a set of bounds on environmental state that we can maintain and that guarantee an improvement in d.

Now, because our agent can only affect one of its angle or its speed, we must actually

$$\begin{aligned} |\theta_t| &\leq \epsilon_{a,\theta} \\ d_t - (1+\delta_s)\frac{\gamma}{1-\gamma}s_t &\geq 0 \\ s_t &\geq \epsilon_{a,s} \end{aligned}$$

Figure A.3: Bounds in the acceleration case, $\omega_{t+1} = 0$

$$\begin{aligned} |\theta_t| &\leq \epsilon_{\omega,\theta} \\ d_t - (1+\delta_s)\frac{\gamma}{1-\gamma}s_t &\geq 0 \\ s_t &\geq \epsilon_{\omega,s} \end{aligned}$$

Figure A.4: Bounds in the rotation case, $a_{t+1} = 0$

define two sets of bounds. In one case, the agent will accelerate, so we must guarantee that it will not need to adjust the angle; that is, $\omega_{t+1} = 0$. Then, we need tight bounds on θ, but we can tolerate looser bounds on s. Specifically, we will have the bounds described in Figure A.3.

In the other case, the turning case, the agent will turn but not accelerate, so $a_{t+1} = 0$. Then, we need tight bounds on s, but we need not be so restrictive of θ. (That is, $\epsilon_{a,\theta} \leq \epsilon_{\omega,\theta}$, and $\epsilon_{a,s} \leq \epsilon_{\omega,s}$.) All the required bounds are given in Figure A.4.

So, given that the agent starts in one of these cases in any time step, we must show that it ends within the bounds of one of these cases, too. The easiest way to do that is to show alternation: that if at time step t the agent is in the acceleration case, then in time $t+1$ it meets the bounds for the turning case; and vice versa. Another way to say that is that the postconditions for the acceleration case match the preconditions for the rotation case (and vice versa). In that way, we can guarantee that the agent never violates these bounds. (Of course, given this guarantee, when the agent is actually working in the environment, it will often find that it meets both sets of bounds. In that case, it can choose whichever action most reduces the error; since it satisfies the current preconditions of both cases, the action will leave it within the postconditions of one of the cases.)

During the detailed analysis below, we will find certain restrictions on the environment and bounds required to make the analysis hold true. (As with all proofs, there may be looser bounds that also allow this analysis and would be revealed by a more clever inspection of the problem.) The conditions used in this analysis are shown in Figure A.5.

A.3 Proving Bounds: Lemmas

First, let us show some useful facts that will make later analysis easier. The easiest are bounds on the agent's outputs. For instance, we know that

$$
\begin{aligned}
1-(1-\gamma)\frac{1+\delta_s}{1+\delta_s} &\geq 0 \\
\frac{\pi}{2}\left(1-(1-\gamma)\frac{1+\delta_s}{1+\delta_s}\right)^{-1}(\epsilon_{a,\theta}+\delta_s) &\leq \epsilon_{\omega,\theta} \\
\frac{\pi}{2}\left(1-(1-\gamma)\frac{1+\delta_s}{1+\delta_s}\right)^{-1}(\epsilon_{\omega,\theta}\delta_\omega+\hat{\delta}_\theta(1+\delta_\omega)+\delta_s) &\leq \epsilon_{a,\theta} \\
\gamma\epsilon_{\omega,s} &\geq \epsilon_{a,s} \\
1-\frac{1-\delta_a}{1+\delta_a}\frac{1+\hat{\delta}_s}{1-\hat{\delta}_s} &\geq 0 \\
\gamma\left(1-\frac{1-\delta_a}{1+\delta_a}\frac{1+\hat{\delta}_s}{1-\hat{\delta}_s}\right)\epsilon_{a,s}+(1-\delta_a)\frac{1-\gamma}{(1+\delta_s)(1+\delta_a)}\frac{1-\hat{\delta}_d}{1+\hat{\delta}_d}\epsilon_d &\geq \epsilon_{\omega,s} \\
1-\gamma-\cos(\epsilon_{\omega,\theta}+\delta_s) &\leq 0
\end{aligned}
$$

Figure A.5: Restrictions on the environment for this analysis to hold

Also,

$$
\begin{aligned}
a_{t+1} &= \frac{1-\gamma}{(1+\delta_s)(1+\delta_a)}\left(\frac{1}{1+\hat{\delta}_d}\hat{d}_t-\frac{1}{1-\hat{\delta}_s}\frac{\gamma}{1-\gamma}(1+\delta_s)\hat{s}_t\right) \\
&\leq \frac{1-\gamma}{(1+\delta_s)(1+\delta_a)}\left(\frac{1+\hat{\delta}_d}{1+\hat{\delta}_d}d_t-\frac{1-\hat{\delta}_s}{1-\hat{\delta}_s}\frac{\gamma}{1-\gamma}(1+\delta_s)s_t\right) \\
&= \frac{1-\gamma}{(1+\delta_s)(1+\delta_a)}\left(d_t-\frac{\gamma}{1-\gamma}(1+\delta_s)s_t\right)
\end{aligned}
$$

Similarly,

$$
a_{t+1} \geq \frac{1-\gamma}{(1+\delta_s)(1+\delta_a)}\left(\frac{1-\hat{\delta}_d}{1+\hat{\delta}_d}d_t-\frac{1+\hat{\delta}_s}{1-\hat{\delta}_s}\frac{\gamma}{1-\gamma}(1+\delta_s)s_t\right)
$$

To help define some variables, a diagram of the relevant angles and distances used in this analysis is given in Figure A.6. Notably, ω_{t+1} is the agent's rotation, η_{t+1} is the amount of divergence caused by random errors, and ϕ_{t+1} is the full resulting angle between the ball and the agent's direction of movement. Similarly, z_{t+1} is the distance the agent actually moves in the $(t+1)^{\text{st}}$ step.

Let us begin by bounding some of these quantities at time $t+1$. We know that the agent has a speed of $s_{t+1}=\gamma s_t+a_{t+1}(1+\xi_a)$, and it moves that amount plus some error in both the x and y directions. Well, if we have error in both x and y of at most $\frac{1}{\sqrt{2}}s_{t+1}\delta_s$, then the maximum error in distance we could have is if the agent went as far as possible in both, so

$$
(1-\delta_s)s_{t+1} \leq z_{t+1} \leq (1+\delta_s)s_{t+1}.
$$

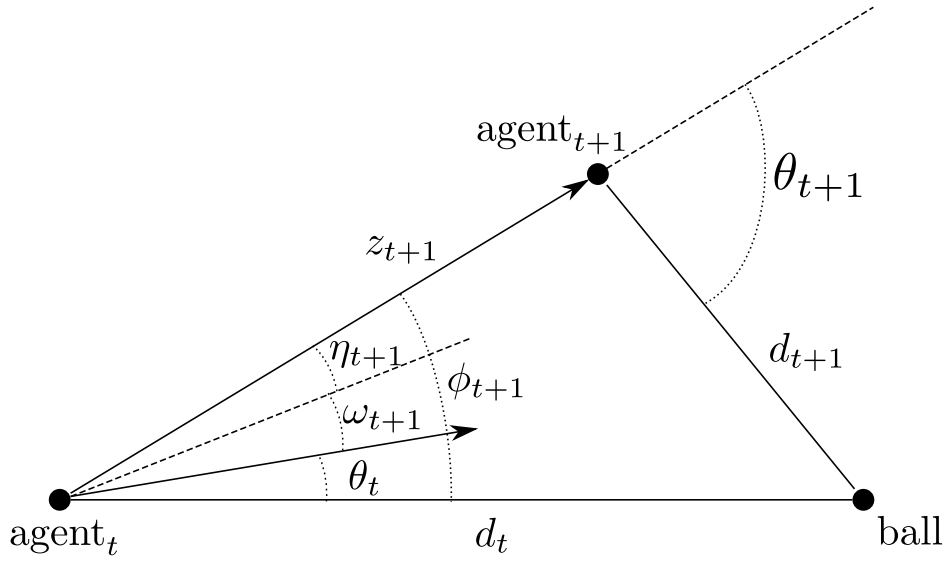


Figure A.6: The angles and distances used in the example analysis.

More precisely,

$$\begin{aligned} z_{t+1} &\leq (1+\delta_s)s_{t+1} \\ &\leq (1+\delta_s)(\gamma s_t + (1+\delta_a)a_{t+1}) \\ &\leq (1+\delta_s)\left(\gamma s_t + (1+\delta_a)\left(\frac{1-\gamma}{(1+\delta_s)(1+\delta_a)}\left(d_t - \frac{\gamma}{1-\gamma}(1+\delta_s)s_t\right)\right)\right) \end{aligned}$$

The s_t terms cancel, as do most of the multipliers of d_t, so

$$z_{t+1} \leq (1-\gamma)d_t$$

Similarly, $\phi_{t+1} = \theta_t + \omega_{t+1} + \eta_{t+1}$. But η_{t+1} comes from two sources: first, the agent's angle might be adjusted by the error in executing its turn, giving a term of $\xi_\omega \omega_{t+1}$. Second, the agent might move in a different direction because some error is added to both x and y when it moves; it could go as far as $\delta_s s_{t+1}$ in a different direction, but the agent would still go at least $(1-\delta_s)s_{t+1}$ in the right direction. This gives an angular error term of δ_s. In that case, recalling the bounds on ξ_ω,

$$\theta_t + (1-\delta_\omega)\omega_{t+1} - \frac{\delta_s}{1-\delta_s} \leq \phi_{t+1} \leq \theta_t + (1+\delta_\omega)\omega_{t+1} + \frac{\delta_s}{1-\delta_s}$$

Although it is not immediately clear why we would want to bound $\frac{d_t}{d_{t+1}}$, it will become useful later, so we do it now. By the triangle rule, $d_t \leq d_{t+1} + z_{t+1}$. But we have upper-bounded

$$z_{t+1} \leq (1+\delta_s)(\gamma s_t + a_{t+1}(1+\delta_a)),$$

so

$$d_{t+1} \geq d_t - z_{t+1} \geq d_t - (1+\delta_s)(\gamma s_t + a_{t+1}(1+\delta_a)).$$

But we have an upper bound on a_{t+1}, so

$$\begin{aligned} d_{t+1} &\geq d_t - (1+\delta_s)(\gamma s_t + \frac{1-\gamma}{(1+\delta_s)(1+\delta_a)}\left(d_t - \frac{\gamma}{1-\gamma}(1+\delta_s)s_t\right)(1+\delta_a)) \\ &= \left(1 - (1+\delta_s)\frac{(1-\gamma)(1+\delta_a)}{(1+\delta_s)(1+\delta_a)}\right)d_t \\ &\quad -(1+\delta_s)(\gamma + \frac{1-\gamma}{(1+\delta_s)(1+\delta_a)}\left(-\frac{\gamma}{1-\gamma}(1+\delta_s)\right)(1+\delta_a))s_t \\ &= \left(1 - (1+\delta_s)\frac{1-\gamma}{1+\delta_s}\right)d_t - (1+\delta_s)(0)s_t \\ &= \left(1 - (1-\gamma)\frac{1+\delta_s}{1+\delta_s}\right)d_t \end{aligned}$$

If this multiplier is at least 0 (we add this condition to our restrictions on the environment), then

$$\frac{d_t}{d_{t+1}} \leq \left(1 - (1-\gamma)\frac{1+\delta_s}{1+\delta_s}\right)^{-1}$$

A.4 Proving Bounds: θ

Having derived some inequalities that will later be helpful, let us now show that $|\theta_{t+1}| \leq \epsilon_{\omega,\theta}$ given that the agent is turning (i.e. $a = 0$, $\omega \neq 0$) and the starting conditions for this case.

First, we show that θ_{t+1} is acute. By the cosine rule,

$$\begin{aligned} d_t^2 &= d_{t+1}^2 + z_{t+1}^2 - 2d_{t+1}z_{t+1}\cos(\pi - \theta_{t+1}) \\ d_{t+1}^2 &= z_{t+1}^2 + d_t^2 - 2z_{t+1}d_t\cos(\phi_{t+1}) \end{aligned}$$

so

$$2d_{t+1}z_{t+1}\cos(\pi - \theta_{t+1}) = 2z_{t+1}^2 - 2z_{t+1}d_t\cos(\phi_{t+1})$$

$$\begin{aligned} d_{t+1}\cos(\pi - \theta_{t+1}) &= z_{t+1} - d_t\cos(\phi_{t+1}) \\ &\leq (1-\gamma)d_t - d_t\cos(\phi_{t+1}) \\ &= (1 - \gamma - \cos(\phi_{t+1}))d_t \end{aligned}$$

And since d_t is non-negative, if that parenthesized term is less than 0, then $\pi - \theta_{t+1}$ is obtuse and θ_{t+1} is acute. We can also upper-bound $\phi \leq \theta_t + (1-\delta_\omega)\omega_{t+1} + \delta_s$, which we combine with $\theta_t \leq \epsilon_{\omega,\theta}$ (the looser of the two bounds on θ_{t+1}) and $\omega_{t+1} \leq 0$, so that term will certainly be positive if

$$1 - \gamma - \cos(\epsilon_{\omega,\theta} + \delta_s) \leq 0$$

Therefore, we include this term in the requirements on the environment.

Now that we know that θ_{t+1} is acute, we can use that to establish tighter bounds. Well, by using the sine rule on the triangle shown in Figure A.6,

$$\sin(\pi - \theta_{t+1}) = \sin(\theta_{t+1}) = \frac{\sin(\phi_{t+1})d_t}{d_{t+1}}$$

But since $0 \leq \theta_{t+1} \leq \frac{\pi}{2}$, then we can lower-bound $\sin(\theta_{t+1})$ by $\sin(\theta_{t+1}) \geq \frac{2}{\pi}\theta_{t+1}$. Similarly, if $\phi \geq 0$, we can upper-bound $\sin(\phi_{t+1}) \leq \phi_{t+1}$. Then,

$$\frac{2}{\pi}\theta_{t+1} \leq \frac{\phi_{t+1}d_t}{d_{t+1}}$$

But we can upper-bound $\frac{d_t}{d_{t+1}}$, so

$$\theta_{t+1} \leq \frac{\pi}{2}\left(1 - (1-\gamma)\frac{1+\delta_s}{1+\delta_s}\right)^{-1}\phi_{t+1}$$

and recall our bound of $\phi_{t+1} \leq \theta_t + \omega_{t+1}(1+\delta_\omega) + \delta_s$ to get

$$\theta_{t+1} \leq \frac{\pi}{2}\left(1 - (1-\gamma)\frac{1+\delta_s}{1+\delta_s}\right)^{-1}(\theta_t + \omega_{t+1}(1+\delta_\omega) + \delta_s)$$

And ω_{t+1} can take one of two values. If the agent is accelerating in this time step, then $\omega_{t+1} = 0$, and we know a bound on θ_t and have already obtained an upper bound on θ_{t+1}, so

$$\theta_{t+1} \leq \frac{\pi}{2}\left(1 - (1-\gamma)\frac{1+\delta_s}{1+\delta_s}\right)^{-1}(\epsilon_{a,\theta} + \delta_s)$$

Then our bound is satisfied if the following condition holds, so we include it in our list of environmental restrictions.

$$\frac{\pi}{2}\left(1 - (1-\gamma)\frac{1+\delta_s}{1+\delta_s}\right)^{-1}(\epsilon_{a,\theta} + \delta_s) \leq \epsilon_{\omega,\theta}$$

Now, consider the rotational case. Recall that our agent rotates with $\omega_{t+1} = -\hat{\theta}_t$, correcting the angular divergence it detects. Then the right-hand side of the inequality becomes

$$\begin{aligned} & \frac{\pi}{2}\left(1 - (1-\gamma)\frac{1+\delta_s}{1+\delta_s}\right)^{-1}(\theta_t - (\theta_t + \hat{\xi}_\theta)(1+\delta_\omega) + \delta_s) \\ = \; & \frac{\pi}{2}\left(1 - (1-\gamma)\frac{1+\delta_s}{1+\delta_s}\right)^{-1}(-\theta_t\delta_\omega - \hat{\xi}_\theta(1+\delta_\omega) + \delta_s) \end{aligned}$$

But of course we really do not care about the sign of the error, so we can simply upper-bound the error with the absolute value of the terms. Furthermore, the observation error can be upper-bounded, so

$$\theta_{t+1} \leq \frac{\pi}{2}\left(1-(1-\gamma)\frac{1+\delta_s}{1+\delta_s}\right)^{-1}(\theta_t\delta_\omega + \hat{\delta}_\theta(1+\delta_\omega)+\delta_s)$$

And of course we started out with an upper bound on θ_t, so

$$\theta_{t+1} \leq \frac{\pi}{2}\left(1-(1-\gamma)\frac{1+\delta_s}{1+\delta_s}\right)^{-1}(\epsilon_{\omega,\theta}\delta_\omega + \hat{\delta}_\theta(1+\delta_\omega)+\delta_s)$$

So in order for our final conditions $\theta_{t+1} \leq \epsilon_{a,\theta}$ to hold, we only need that this right-hand quantity be less than $\epsilon_{a,\theta}$:

$$\frac{\pi}{2}\left(1-(1-\gamma)\frac{1+\delta_s}{1+\delta_s}\right)^{-1}(\epsilon_{\omega,\theta}\delta_\omega + \hat{\delta}_\theta(1+\delta_\omega)+\delta_s) \leq \epsilon_{a,\theta}$$

Note that this equation, though messy, includes no environment state variables, only parameters describing the environment. Thus, it describes a set of environments for which this behavior will maintain this particular bound. As long as our agent operates in one of those environments, its behavior will work appropriately.

A.5 Proving Bounds: d and s

We want to show that at the end of each time step, the agent does not overshoot. That is, we require

$$d_t - (1+\delta_s)\frac{\gamma}{1-\gamma}s_t \equiv q \geq 0$$

Here, I label the left-hand quantity q for simplicity in the rest of the analysis. But of course the agent cannot have traveled more than $(1+\delta_s)s_{t+1}$ in this time step, so we can lower-bound $d_{t+1} \geq d_t - (1+\delta_s)s_{t+1}$, giving us

$$\begin{aligned} q &\geq d_t - (1+\delta_s)s_{t+1} - \frac{\gamma}{1-\gamma}(1+\delta_s)s_{t+1} \\ &= d_t - \left(1+\frac{\gamma}{1-\gamma}\right)(1+\delta_s)s_{t+1} \\ &= d_t - \frac{1}{1-\gamma}(1+\delta_s)s_{t+1} \end{aligned}$$

Also, we can upper-bound $s_{t+1} \leq \gamma s_t + (1+\delta_a)a_{t+1}$, so

$$\begin{aligned} q &\geq d_t - \frac{1}{1-\gamma}(1+\delta_s)(\gamma s_t + (1+\delta_a)a_{t+1}) \\ &= \left(d_t - \frac{\gamma}{1-\gamma}(1+\delta_s)s_t\right) - \frac{(1+\delta_s)(1+\delta_a)}{1-\gamma}a_{t+1} \end{aligned}$$

We started this time step with a bound that the left parenthesized term is at least 0, and in the rotational case we also have that $a_{t+1} = 0$, so

$$q \geq 0.$$

It is very easy to apply similar logic to arrive at a similar conclusion for the acceleration case, where

$$a_{t+1} \leq \frac{1-\gamma}{(1+\delta_s)(1+\delta_a)}\left(d_t - \frac{\gamma}{1-\gamma}(1+\delta_s)s_t\right)$$

Then, recalling that

$$\begin{aligned} q &\geq \left(d_t - \frac{\gamma}{1-\gamma}(1+\delta_s)s_t\right) - \frac{(1+\delta_s)(1+\delta_a)}{1-\gamma}a_{t+1} \\ &\geq \left(d_t - \frac{\gamma}{1-\gamma}(1+\delta_s)s_t\right) - \frac{(1+\delta_s)(1+\delta_a)}{1-\gamma}\frac{1-\gamma}{(1+\delta_s)(1+\delta_a)}\left(d_t - \frac{\gamma}{1-\gamma}(1+\delta_s)s_t\right) \\ &= 0 \end{aligned}$$

so the desired bound is satisfied for both the rotational and the acceleration cases.

A.6 Proving Bounds: s

Now that we have proven that the agent stays correctly-oriented and never overshoots its target, we need only show that it maintains a sufficient speed to converge to its goal. Fortunately, this is not difficult.

In the rotational case, we have that $a_{t+1} = 0$, so $s_{t+1} = \gamma s_t$. We want $s_{t+1} \geq \epsilon_{a,s}$ but know that $s_t \geq \epsilon_{\omega,s}$, so all we need is that

$$\gamma\epsilon_{\omega,s} \geq \epsilon_{a,s}$$

In the acceleration case, if $a_{t+1} \neq 0$, we know that

$$\begin{aligned}
s_{t+1} &\geq \gamma s_t + (1-\delta_a) a_{t+1} \\
&= \gamma s_t + (1-\delta_a)\frac{1-\gamma}{(1+\delta_s)(1+\delta_a)}\left(\frac{1}{1+\hat{\delta}_d}\hat{d}_t - \frac{1}{1-\hat{\delta}_s}\frac{\gamma}{1-\gamma}(1+\delta_s)\hat{s}_t\right) \\
&\geq \gamma s_t + (1-\delta_a)\frac{1-\gamma}{(1+\delta_s)(1+\delta_a)}\left(\frac{1-\hat{\delta}_d}{1+\hat{\delta}_d}d_t - \frac{1+\hat{\delta}_s}{1-\hat{\delta}_s}\frac{\gamma}{1-\gamma}(1+\delta_s)s_t\right) \\
&= \left(\gamma - (1-\delta_a)\frac{1-\gamma}{(1+\delta_s)(1+\delta_a)}\frac{1+\hat{\delta}_s}{1-\hat{\delta}_s}\frac{\gamma}{1-\gamma}(1+\delta_s)\right)s_t \\
&\quad + (1-\delta_a)\frac{1-\gamma}{(1+\delta_s)(1+\delta_a)}\frac{1-\hat{\delta}_d}{1+\hat{\delta}_d}d_t \\
&= \gamma\left(1 - \frac{1-\delta_a}{1+\delta_a}\frac{1+\hat{\delta}_s}{1-\hat{\delta}_s}\right)s_t \\
&\quad + (1-\delta_a)\frac{1-\gamma}{(1+\delta_s)(1+\delta_a)}\frac{1-\hat{\delta}_d}{1+\hat{\delta}_d}d_t
\end{aligned}$$

We know that $s_t \geq \epsilon_{a,s}$ and $d_t \geq \epsilon_d$ (because otherwise the agent would already have reached its goal). In that case, the two following conditions guarantee that the agent will satisfy our bounds. (The first condition is to make sure we can use our lower bound on s_t.)

$$\begin{aligned}
\left(1 - \frac{1-\delta_a}{1+\delta_a}\frac{1+\hat{\delta}_s}{1-\hat{\delta}_s}\right) &\geq 0 \\
\gamma\left(1 - \frac{1-\delta_a}{1+\delta_a}\frac{1+\hat{\delta}_s}{1-\hat{\delta}_s}\right)\epsilon_{a,s} + (1-\delta_a)\frac{1-\gamma}{(1+\delta_s)(1+\delta_a)}\frac{1-\hat{\delta}_d}{1+\hat{\delta}_d}\epsilon_d &\geq \epsilon_{\omega,s}
\end{aligned}$$

A.7 Proving Bounds: Convergence

Now, with all those bounds, let us show that this behavior actually leads to reaching the goal. We will do that by showing that d is reduced in each time step.

By the cosine rule,

$$\begin{aligned}
d_t^2 &= d_{t+1}^2 + z_{t+1}^2 - 2d_{t+1}z_{t+1}\cos(\pi - \theta_{t+1}) \\
z_{t+1}^2 &= d_t^2 + d_{t+1}^2 - 2d_t d_{t+1}\cos(\theta_{t+1} - \phi_{t+1})
\end{aligned}$$

We combine those to get

$$d_{t+1} = d_t\cos(\theta_{t+1} - \phi_{t+1}) + z_{t+1}\cos(\pi - \theta_{t+1})$$

But we can upper-bound the left cosine by 1, and rewrite the right one, so

$$\begin{aligned} d_{t+1} &\leq d_t - \cos(\theta_{t+1}) z_{t+1} \\ &\leq d_t - \cos(\theta_{t+1})(1+\delta_s) s_{t+1} \\ &\leq d_t - \cos(\epsilon_{\omega,\theta})(1+\delta_s)\epsilon_{a,s} \end{aligned}$$

That is, in each time step, the agent improves its error d by at least $\cos(\epsilon_{\omega,\theta})(1+\delta_s)\epsilon_{a,s}$, so it will converge in a finite amount of time. Therefore, this behavior is guaranteed to be successful.

A.8 Proving Bounds: Conclusion

This entire derivation has been more hassle than anyone could have hoped. All the same, I would like to underline the fact that this proves behavioral correctness for a whole range of environmental conditions. In particular, with the RoboCup Soccer Simulator default parameters, these bounds hold.