PLAN RECOGNITION IN DYNAMIC 2-D ENVIRONMENTS

Joshua Gross

BSc, Ryerson University, Toronto, Canada, 2008

A thesis

presented to Ryerson University

in partial fulfilment of the

requirements for the degree of

Master of Science

In the program of

Computer Science

Toronto, Ontario, Canada, 2011

I hereby declare that I am the sole author of this thesis or dissertation.

I authorize Ryerson University to lend this thesis or dissertation to other institutions or individuals for the purpose of scholarly research.

_____

Place, Date

_____

Signature

I further authorize Ryerson University to reproduce this thesis or dissertation by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

_____

Place, Date

_____

Signature

PLAN RECOGNITION IN DYNAMIC 2-D ENVIRONMENTS

Joshua Gross

MSc, Computer Science, Ryerson University, 2011

Abstract

We look at the relatively unexplored problem of plan recognition applied to motion in 2-D environments where all moving objects are modelled as circles. Golog is a well-known high level logical language for solving planning problems and specifying agent controllers. Few studies have applied Golog to plan recognition. We use some of the features of this language, but its standard interpreter is adapted to solving plan recognition problems. This thesis makes several other contributions. First, plan recognition procedures are formulated as finite automata and expressed as Golog programs. Second, we elaborate a logical formalism for reasoning about depth and motion from an observer's viewpoint. We not only expand on this situation calculus based formalism, but also apply it to tackle plan recognition problems in the traffic domain. The proposed approach is implemented and thoroughly tested on recognizing simple behaviours such as left turns, right turns, and overtaking.

Contents

List of Tables

List of Figures

List of Appendices

Important Definitions

This sections contains a collection of definitions of important terms and symbols used throughout the thesis.

Table 1: Definitions of Important Terms and Symbols

Term/Symbol	Definition
Depth Peak	A depth peak is a data structure that contains information about an object in the domain. It consists of depth, size, and left boundary distance.
Depth Profile	A depth profile is a collection of depth peaks. It encodes all the information about objects in the domain at a given time.
Depth	Depth is the distance from the center of the observer to the nearest point on an object.
Size	Size is the angular size that an object occupies in an observer's field of view. It is calculated by the angular distance between the two tangent points on the object.
Left Boundary Distance (lbd)	The left boundary distance is the angular distance from the left limit of the observer's field of view to the center of the object.
S_0	The symbol representing the initial situation before any actions have occurred in The Situation Calculus
Left Border	The left limit of the field of view of the robot.
Body Angle	The angle a body makes with the x-axis.
$\stackrel{\circ}{=}$	The symbol, $\stackrel{\circ}{=}$ is defined as a macro, where the left hand side of the equations are defined by the right hand side.

Chapter 1

Introduction

In this thesis we are solving the problem of recognizing behaviours of objects from the view point of an observer as they move in an environment. The domain consists of an observer and other objects that are in motion. We use a logic-based formalism for representing knowledge about objects in space and their movement. This representations of moving objects helps to define the relations between the observer and an object moving in the domain. Using these relations we are able to define behaviours that encapsulate the changes in relations over time between the observer and other objects. A direct application is to the traffic domain to recognize what behaviours other vehicles are doing around you as you are driving.

Behaviours can be recognized in real time and incrementally. Sensor data is gathered from the point of view of the robot and processed in real time. This data is then used to define relations using a logic-based formalism. Behaviours are defined based on these formalisms. Furthermore behaviours are described as being a series of steps that need to be completed. Since behaviours are built as a series of steps using logic-based relations we are able to determine before completion of a behaviour which behaviour is being performed in the domain. If the changes we see in relations between observer and an object match the behaviour defined we can know how much of that behaviour definition has been matched so far.

Much research in robotics is based around search to solve planning problems. In this thesis we look at the more unexplored problem of plan recognition. Specifically in this thesis we move away from the traditional application of Golog to planning problems and apply Golog to solving

plan recognition problems instead. Also, we build on the initial work by [**?**] on defining a high level logic based formalism for constructing symbolic representations of moving objects called depth profiles. We apply this domain representation to simulated real world examples based in the traffic domain. We explore the ability of depth profiles to successfully define relations in the domain and furthermore use these formalisms in a plan recognition system based in Golog, traditionally used as a planing system.

1.1 Methodology

Our approach to plan recognition in this thesis can be described as being a qualitatively based logic formalism used in an adapted Golog system. We support these methodological decisions based on research of work as discussed in Chapter 2. Choosing to qualitatively define our domain is advantageous over a quantitative approach. When dealing with numbers, qualitative descriptions allow us to define a single descriptor for a range consisting of relatively similar values. This in turn leads to less computation time and more expressiveness in the system. Also, it increases readability of the system. Using a qualitative approach allows more users to understand the data that is being dealt with. Providing a qualitative English language description of ranges and relations between data better describes problems than a quantitative approach for the average user. Finally, we found in this thesis that a qualitative approach allows us to better describe behaviours of objects and furthermore combine these into more complex behaviours. This leads us to an ability to better reason about behaviours of objects even when they are temporally occluded while in motion.

In this thesis we choose to adapt Golog to be used as the basis for our plan recognition system. Golog relies on the situation calculus which allows us to reason about the effects of actions and the results of sensing. Also, Golog has already been shown to be adaptable from planning to plan recognition [**?**]. Other formalisms, such as hierarchical task networks(HTN), have been shown to be functionally equivalent to Golog, [**?**]. HTN's can be encoded in Golog using only a few of their constructs. As well, many techniques have been developed for the many problems that have arisen in cognitive robotics research. These techniques are relevant to modelling dynamic worlds. Golog also naturally leads to designing plan recognition programs in a modular fashion. This allows us to define behaviours in steps, more similar to how they are executed in the real world. Finally,

the interpreters built around Golog allow for incremental recognition. This is important as it gives the system the ability to recognize behaviours as they are happening, rather than once they are completed. Because of being based in the situation calculus and previously being used to model dynamic worlds, Golog was a more natural choice for developing a robust plan recognition system for motion.

Also, in this thesis, we chose a modular approach to designing plan recognition programs as well as an incremental plan recognition system.

1.2 Outline

In Chapter 2 of this thesis we review all background materials that are needed for later discussions in the thesis. We define the language of Situation Calculus as well as the high level programming languages of Golog and ConGolog. A few basic axioms used throughout the paper are also defined in this chapter. We also explore related work in defining and recognizing motion in dynamic environments. Specifically, we look at cognitive vision systems, [**?**, **?**], qualitative mapping, [**?**, **?**, **?**, **?**], and automated learning models, [**?**, **?**, **?**].

In Chapter 3, we formally define the most important axioms we will use in our plan recognition system. Also we show how the system is designed to take input and in turn recognize plans. We explore the similarities between an informal definition of a Finite State Automaton(FSA) and a Golog plan recognition system as the basis of an automaton to recognize plans. As well we do a brief complexity analysis of the system.

In Chapter 4, we use ConGolog as a plan recognition system. We modify the traditional Trans/-Final interpreter to turn it into a plan recognition system as well as adding the ability to recognize multiple bodies and plans at the same time. On top of this we discuss a number of examples showing the capabilities and limitations of the system.

Finally, in Chapter 5, we discuss future directions for this work.

The main objective of this thesis was to expand and apply the depth profile theory developed by Santos, [**?**]. We use the relations defined by Santos to define more complex behaviours and in turn we use these as the basis for a plan recognition system. We create a formal definition of a plan recognition system based in the Situation Calculus by comparing it to an informal FSA. From there

we look at behaviours in the traffic domain for showing the effectiveness of the plan recognition system and the relations we have defined.

1.3 Contributions

Below is a list of contributions made in this thesis

1. 8 additional successor state axioms added as an extension to [**?**] that describe relative direction from the view of an observer. These are first explained in Section **??**

2. A software simulator for simulating traffic scenarios as well as mimicking the sense and movement actions of an observer consisting of 9000 lines of C++ code. The simulator is built using QT and CGAL frameworks. As well as a plan recognition system based in ConGolog that connects to the simulator consisting of 6800 lines of Prolog code. The software can be found here: `http://www.scs.ryerson.ca/~j2gross`

3. As discussed later in Chapter 3, an informal FSA for plan recognition that receives as input fluents. A design structure for FSA to be built that can recognize different plan recognition programs. As well as a clearly defined structure for input to the FSA.

4. A complexity analysis that supports the design choices when building an FSA as defined in Chapter 3. The design choices were made to limit the number of transitions needed to reach a Final state in the FSA as well as eliminate false positives in the FSA.

5. A modification of the $Trans/Final$ ConGolog interpreter to change it from planning system to a plan recognition system. This includes changes that allow it to reason about multiple bodies and multiple plan recognition programs in real time as discussed in Chapter 4.

6. Finally a series of examples showing the capabilities of the plan recognition system developed throughout the thesis as seen in Chapter 4.

Chapter 2

Background

This chapter provides background material that we use in all subsequent chapters of the thesis.

2.1 Dynamic Depth Profiles

We use the idea of depth profiles to encode information in our domain. The description of the static representation of depth profiles in the situation calculus will be discussed in the next section. In this section we will discuss how a depth profile can encode all relevant information in a single time slice, independent of the situation calculus, as put forward by Santos and others in [**?**, **?**, **?**, **?**, **?**]. We leave all other formal situation calculus definitions to chapter **??**.

The logical theory presented in this paper is for representing knowledge about objects in space and their movement from the viewpoint of a mobile robot. The general framework described here falls into the area of cognitive robotics. The aim of cognitive robotics is to endow robots with high-level facilities by using techniques from the field of knowledge representation, especially those that use formal logic as their theoretical foundation.

The idea of a depth profile was proposed to reason about perception of depth and spatial relations between moving physical objects. For simplicity, it is assumed that all objects have cylinder-like shapes. Subsequently, we rely on this assumption and on a simple computational geometry algorithm that handles circles (cross-sections of a cylinder). A depth profile is a unique way to represent depth and motion, whereby each object in the domain is represented by a single peak.

A depth profile can be constructed using the data obtained from a robot's range finding sensor in the form of a horizontal slice across the centre of the visual field. Each horizontal slice is representative of a two-dimensional slice which preserves the horizontal shape and depth of objects in the scene. We use Euclidean geometry to compute the depth profiles from horizontal slices gained from the robots sensors. Depth profiles can change due to movement actions. (See the example in Section **??** for details).

For now, let us consider a static scene populated with cylindrical objects, see Figure **??**. Calculating a depth profile from sensor data is focused around rotational sweep line algorithm that can run in $O(n * log(n))$ from [**?**, **?**, **?**]. Here we rely on the simplification that objects are approximated as circles. Using the sensor with an adapted sweep line algorithm we are able to create a depth profile for a single horizontal slice. We assume that our sensor provides a left and right tangent point on the circle, as well the closest point on the circle. With this information we are able to move in a circular motion from the robot's left boundary and upon encountering every object we can calculate the properties of each depth peak [1] from the algorithms that are subsequently discussed in axioms { **??**,**??**,**??** }. A depth peak is a data structure that contains information about an object in the domain. It consists of depth, size, and left boundary distance It is important to note that some objects may appear to be invisible due to occlusions.

Using the information that a sensor is able to gather, Figure **??**, we are able to ascertain a few important facts. Using the tangent points, p' and p'' the angular distance between them is equal to the $size$ of a depth peak. The distance d to the closest point on the object is equal to the $depth$ of a depth peak. Finally, the left boundary distance, (subsequently abbreviated as lbd), of a depth peak can be calculated by the difference from the left limit of the robots field of view to the center of the object. The left boundary distance is measured in degrees, as it is an angular distance. In Figure **??**, the nearest tangent point would be p''. Using the information from the sensors we are able to create a depth profile of the current situation, see Figure **??**.

[1] This is accomplished using axioms

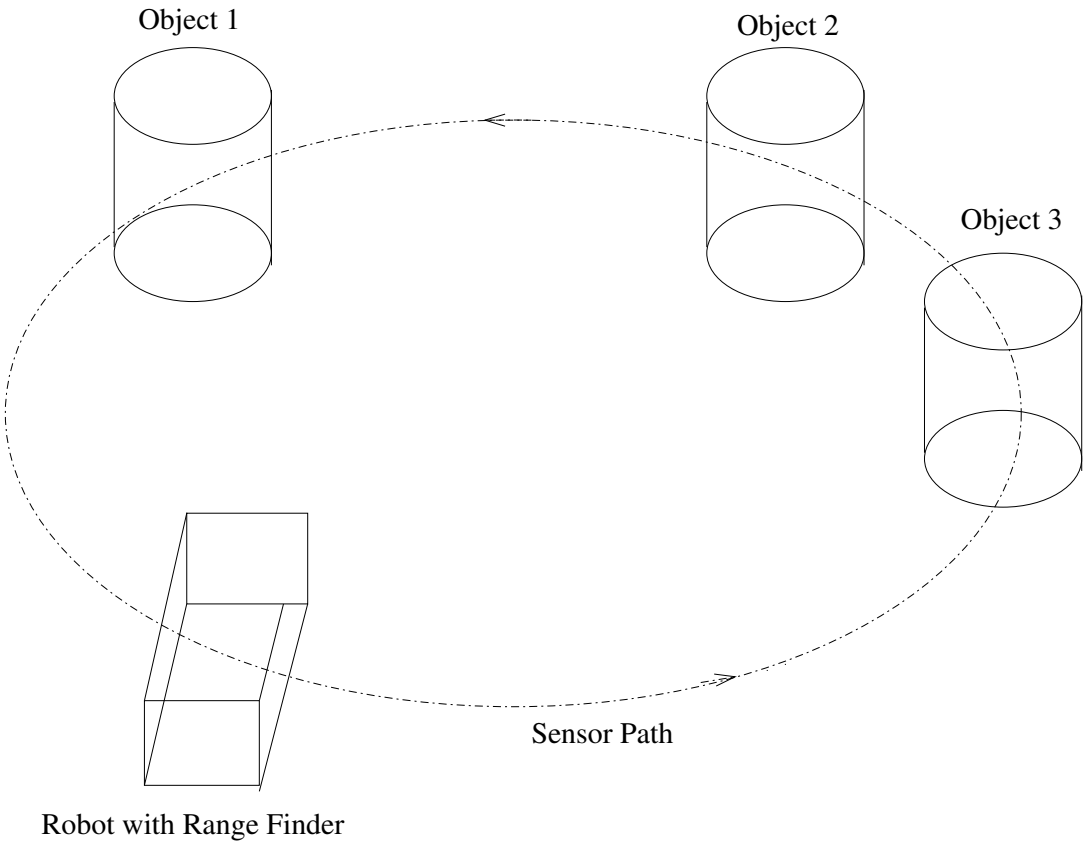

Figure 2.1: An example of a 3-D scene with the robots range finding sensor path approximated

2.2 The Situation Calculus

In this section we define the language of the situation calculus and show how we can represent actions and their effects in this language. The situation calculus is a dialect of first order logic used as a logical approach to modelling dynamical systems first proposed by McCarthy in 1963 [**?**]. We use the definitions further described by Reiter in [**?**].

The situation calculus is composed of three basic concepts, *actions*, *situations* and *fluents*. *Actions* describe all possible changes to the world. A history of the world is a sequence of actions represented by a first order term called a *situation*. The initial situation is represented by the constant S_0 and is the empty sequence of actions. A special binary function symbol do is used to represent the successor situation,$do(\alpha, s)$, to the situation, s. resulting from the action α. The values of relations or functions may change from situation to situation. These are called *fluents*

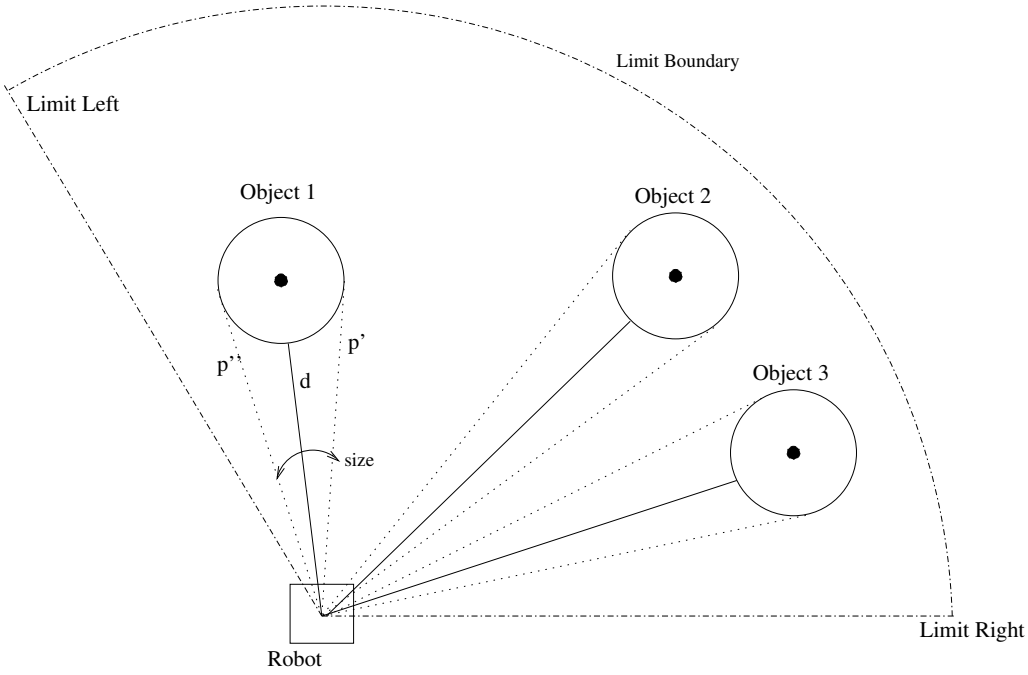

Figure 2.2: Horizontal slice obtained from sensors in Figure **??**

and are denoted by a predicate or function symbol, each symbol takes among its arguments exactly one of type *situation*. We will now go into further detail of each of these elements of the situation calculus.

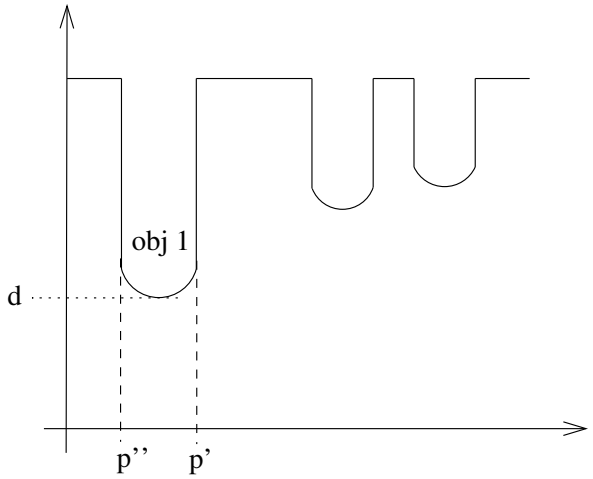

Figure 2.3: Depth graph from horizontal slice in Figure **??**

2.2.1 Definition of The Situation Calculus

Actions

There are a finitely many different (primitive) actions that can be executed. Actions are defined using function symbols. Every argument of an action function symbol must be of type object (objects can be agents, locations, and physical entities). We name these function symbols primitive actions. There are two different types of primitive actions, physical and sensing. They differ in that physical actions represent changes to the world around them, while sensing actions only make observations about external changes. Variables of type action will be denoted by $\alpha, \alpha', \alpha_1, \alpha_2, \ldots$. Terms of type action will be explicitly declared with primes, subscripts and arguments when the differences between variables and terms needs to be shown. Subsequently, we consider instantaneous actions whose last argument is the time when an action is executed. Note that the process extended in time (e.g. moving from one location to another) can always be represented by two instantaneous actions: an action that initiates the process and another action that terminates the process.

Situations

There are two function symbols of type situation, the constant S_0, representing the initial situation, and the binary function symbol do which takes arguments of type action and situation. Ground situation terms are represented by uppercase S with primes and subscripts. The constant S_0 represents the initial situation or the empty action sequence. It is similar to LISP's () or Prolog's []. The binary function symbol do is used with arguments α and s to represent the situation resulting from action α in situation s, $do(\alpha, s)$. Using this notation the situation term

$$do(endMove(robot, location1, location2, t3),$$
$$do(startMove(robot, location1, location2, t2),$$
$$do(sense(profile, location1, t1), S_0)))$$

represents the sequence of actions $sense(profile, location1, t1)$, $startMove(robot, location1, location2, t2)$, $endMove(robot, location1, location2, t3)$. It is important to notice that the actions sequence obtained from the binary function symbol do is read from right to left.

Fluents

Fluents are relations or functions whose values may vary from one situation to the next. Fluents are represented as relation and function symbols that have a situation term as their last argument. Fluents are domain specific predicate or function symbols and there are finitely many fluents. For example, we could have the relational fluent $on(x, y, s)$ which would mean that after the action sequence s, x is on y.

All of the above function symbols and relations are part of the language of situation calculus denoted by ℓ.

Basic Action Theories

A basic action theory (BAT) is a collection of axioms in the situation calculus with the following five classes of axioms to model actions and their effects: action precondition axioms, successor state axioms, the initial theory, unique name axioms for actions, and domain independent foundational axioms for situations. Once a BAT has been defined we are able to do logical reasoning about an agents' actions. These axioms are formulated using uniform formulas. These are situation calculus formulas whose only situation term is s, which have no quantifications over s, have no occurrences of special predicates $Poss(\alpha, s)$ and $\sqsubset$, that are defined below.

Precondition Axioms D_{ap}

For every primitive action $A(\vec{x})$ there is an axiom of the form, where $\vec{x}$ is representative of any number of variables for the action A.

$$Poss(A(\vec{x}), s) \equiv \Pi_A(\vec{x}, s)$$

where $\Pi_A(\vec{x}, s)$ is an uniform formula with free variables among $\vec{x}, s$. Each axiom defines the conditions, that must be met in order for the action $A(\overrightarrow{x})$ to be executed in situation s. The action $A(\vec{x})$ can be executed in situation s if $\Pi_A(\vec{x}, s)$ holds.

Successor State Axioms D_{ss} (SSA)

For each relational fluent $F(\vec{x}, s)$ there is one axiom of the form

$$F(\vec{x}, do(\alpha, s)) \equiv P_F(\vec{x}, \alpha, s) \vee [F(\vec{x}, s) \wedge \neg N_F(\vec{x}, \alpha, s)]$$

where F is the fluent symbol, P_F is a uniform formula representing the positive effect condition for fluent F becoming true and N_F is a uniform formula representing the negative effect condition for F. A part of this axiom, $F(\vec{x}, s) \wedge \neg N_F(\vec{x}, \alpha, s) \supset F(\vec{x}, do(\alpha, s))$ sometimes is called the frame axiom, since it characterizes when fluent $F(\vec{x}, s)$ does not change its logical value from s to $do(\alpha, s)$. This axiomatization,proposed by Reiter in [**?**], provides a solution to the frame problem first. Here, $P_F(\vec{x}, \alpha, s)$ includes all the conditions where the fluent F can be made true by the action α and $N_F(\vec{x}, \alpha, s)$ includes all the conditions where the fluent F can be made false by the action α. Subsequently, when we talk about the frame axiom for fluent F, we refer to a part of the SSA for F that involves $N_F(\vec{x}, \alpha)$. The examples of precondition and successor state axioms are provided in the next Section **??**.

Initial Theory D_{S_0} and Unique Name Axioms for Actions D_{una}

The initial theory is a set of first order formulas whose only situation term is S_0. It specifies the values of all fluents in the initial state before any actions have been executed. It also describes all the facts that are not changeable by any actions in the domain. In particular, it includes unique name axioms for object constants.

Unique name axioms specify that all actions in a domain are different. That is, if their names are different then they cannot be equal and identical actions will have identical arguments and names.

Foundational Axioms for situations, FA

[**?**] Below are the foundational axioms for situations which characterize the basic properties of situations. These axioms are domain independent. They are included in the axiomatization of any dynamical system in the situation calculus. The set of foundational axioms is as follows:

$$do(\alpha_1, s_1) = do(\alpha_2, s_2) \supset \alpha_1 = \alpha_2 \wedge s_1 = s_2, \tag{2.1}$$

$$(\forall P).P(S_0) \wedge (\forall \alpha, s)[P(s) \supset P(do(\alpha, s))] \supset (\forall s)P(s), \tag{2.2}$$

$$\neg s \sqsubset S_0, \tag{2.3}$$

$$s \sqsubseteq do(\alpha, s') \equiv s \sqsubseteq s' \vee s = s'. \tag{2.4}$$

The special predicate $\sqsubseteq$, (can be read as, subsequence or equal to) defines partial orders between situations. The axiom (2.2) states that the type situation is the smallest set containing S_0 that is closed under the application of do to an action and situation. Axiom (2.1) states that if two situations are identical, then they are reached using identical actions and identical situation terms. This is similar to the unique name axioms for actions, but for situations. Axiom (2.3) states that S_0 has no predecessors and axiom (2.4) asserts that if a sequence s is included in the sequence represented by $do(\alpha, s')$, then either $s = s'$ or s is a subsequence of s'. This axiom (2.4) guarantees that the predicate $\sqsubseteq$ represents partial order.

2.2.2 Example Domain in The Situation Calculus

Logical Precondition Axioms for Motion

The precondition axioms describe what actions can occur in a domain, and under what conditions these actions can occur. In the motion domain we are describing here, all actions are considered instantaneous. Processes that have duration over time (such as moving between 2 locations) are represented with the corresponding instantaneous initiating and terminating actions (e.g. $startMove$ and $endMove$. There are 5 actions that are possible in this domain and we will discuss, in detail, all of them in this section. Once an action has been executed, it becomes part of the situation term.

Movement

There are two precondition axioms for movement. We use the notation (x^2, y^2), where we use superscripts to indicate meaning and not the traditional meaning of powers on (x, y) in the following sections. The first axiom for the $startMove$ action states that if an object is in a location then it can start moving to a new location iff the two locations are not the same and the object is not already moving. The second axiom for the end move action states that an object can stop

moving and reach the second location iff it was already moving in the previous situation and a change in location did not occur. Having both a start and end action further show that actions are instantaneous. After an object has started moving, the fluent for moving will be true, until it has reached its destination location, $loc(x^2, y^2)$, and the action, $endMove$, has occurred. The SSA for moving is discussed below.

$$poss(startMove(object, loc(x^1, y^1), loc(x^2, y^2), t), s) \equiv \tag{2.5}$$
$$location(object, loc(x^1, y^1), s) \wedge loc(x^1, y^1) \neq loc(x^2, y^2) \wedge$$
$$\neg\exists(loc(x', y'), loc(x'', y'')) \, moving(object, loc(x', y'), loc(x'', y''), s).$$

$$poss(endMove(object, loc(x^1, y^1), loc(x^2, y^2), t), s) \equiv \tag{2.6}$$
$$moving(object, loc(x^1, y^1), loc(x^2, y^2), s) \wedge (loc(x^1, y^1) \neq loc(x^2, y^2)).$$

Rotating

Similarly to movement, there are start and end actions for rotation of the robot. The robot or observer is representative of the point where sensing and reasoning is performed from. The actions for rotation allow us to know which angle the observer is facing. The first precondition axiom for the start pan action states that an object can start rotating, iff it is not already rotating. The second precondition axiom states that an object can end rotating iff it is already rotating. The rotation angle, ω is positive if the rotation is clockwise and negative if the rotation is counterclockwise.

$$poss(startPan(\omega, t), s) \equiv \neg\exists \, \omega' rotating(\omega', s). \tag{2.7}$$
$$poss(endPan(\omega, t), s) \equiv rotating(\omega, s). \tag{2.8}$$

Sensing

The robot can also use its sensors to get a profile, p, of all the depth peaks visible at a given moment in time. The precondition axiom for sensing states that the action of sensing can occur if the robot is in the location, $loc(x^r, y^r)$, and the data it gets from its sensors are all within the acceptable ranges of size, depth and left boundary distances. This profile can be used for more

complex reasoning tasks relying on the successor state axioms discussed below. Also, from this sensing data we are able to update the profile based on any other actions that have occurred, such as movement and rotation. Note, lbd stands for an angle representing the angular distance from the left boundary of the robot's field of view to the centre of the object being reasoned about.

$$
\begin{aligned}
&poss(sense(p, loc(x^r, y^r), t, s) \equiv \\
&\quad getSensingData(p) \wedge location(robot, loc(x^r, y^r), s) \wedge \\
&\quad \{\neg\exists(body, depth, size, lbd, l)\, peakOf(body, depth, size, lbd) \in p \wedge \\
&\quad (depth =< 0 \vee size =< 0 \vee lbd < 0 \vee lbd > 360 \vee \\
&\quad limitBoundary(l) \wedge depth > l)\}.
\end{aligned} \tag{2.9}
$$

Action and Peak Attribute Successor State Axioms

In this section we discuss some of the successor state axioms that were explained in [**?**]. The first set of SSA represent what an object may be physically doing in the domain. The second set represent specific peak attributes that are affected by actions.

The successor state axiom for $facing$ (**??**) returns the direction that the robot is facing in the current situation $do(a, s)$ based on the actions it has performed in the past. The SSA states that the robot will be facing direction, θ' iff the action $endPan$ occurred and the robot was facing a previous direction, θ'' in the previous situation, s. The current direction θ' will be equal to the previous direction faced plus the angle of rotation, $\theta'' + \omega$. Otherwise the robot will still be facing the direction it faced in the previous situation, s, if the action that occurred was not a rotation action. The angle, θ is measured from north in a clockwise direction. Therefore if the angle of rotation was clockwise then it will be a positive number, if it was counterclockwise it will be a negative number.

$$
\begin{aligned}
&facing(robot, \theta', do(a, s)) \equiv \exists(\omega, t, \theta'') \\
&\quad a = endPan(\omega, t) \wedge facing(robot, \theta'', s) \wedge \theta' = \theta'' + \omega \\
&\quad \vee facing(robot, \theta', s) \wedge \neg\exists(\omega, t) a = endPan(\omega, t).
\end{aligned} \tag{2.10}
$$

The successor state axiom for location (**??**) determines the current location, $loc(x, y)$ of an object or the robot based on the actions the associated entity has performed in the past. The SSA for location states that if the action $endMove(object, loc(x', y'), loc(x, y), t)$ has occurred then the object is at location $loc(x, y)$. If the robot has performed a sensing action in the current situation, $do(a, s)$, then it can use Euclidean geometry to determine the location of all objects in the depth profile, $profile$, based on its current location, $loc(x^r, y^r)$. The location of the object is calculated from the current location of the robot, $loc(x^r, y^r)$, the angle the object makes with the positive x-axis and the depth of the object. Otherwise, the object will be in the same location it was in the previous situation, s, if an action $endMove$ that moved it from $loc(x', y')$ to a different location $loc(x^2, y^2)$ has not occurred. Note, we assume that the sensing action always updates the locations of all objects. We introduce two new variables in the axiom below. First, $leftBorder$ represents the angle of the observers left boundary for its field of view. Second, $bodyangle$ represents the angle the $body$ makes with the east axis in the domain. We use this for the formula, $calcLocation$, to accurately calculate the location of any object in the domain from a static axis, the x-axis. The formula, $calcLocation$ is explained later in Section **??**, see Formulas **??** and **??**. In Figure **??**, the figure shows how $bodyangle$ relates to $leftBorder$ and lbd.

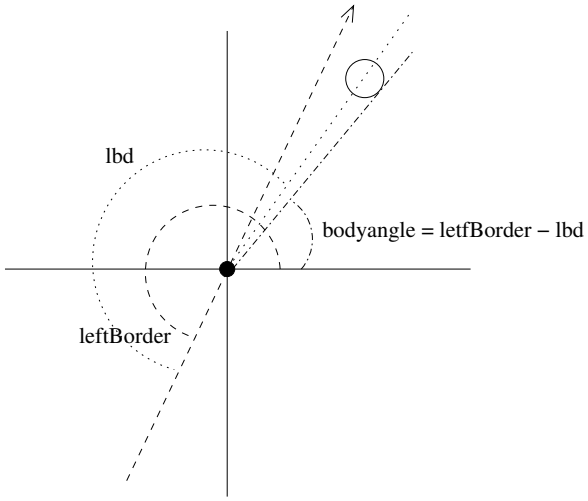

Figure 2.4: Relation Between $leftBorder$, lbd, and $bodyangle$

$$location(object, loc(x, y), do(a, s)) \equiv \tag{2.11}$$

$\exists(x', y', t)(a = endMove(object, loc(x', y'), loc(x, y), t)) \vee$

$\exists(body, profile, x^r, y^r, t, \alpha, \theta, leftBorder, bodyangle, depth, size, lbd, r)$

$body = object \wedge a = sense(profile, loc(x^r, y^r), t) \wedge$

$peakOf(body, depth, size, lbd) \in profile) \wedge$

$location(robot, loc(x^r, y^r), s) \wedge facing(robot, \theta, s) \wedge fieldView(alpha) \wedge$

$leftBorder = \theta + \alpha/2 \wedge (bodyangle = leftBorder - lbd) \wedge radius(body, r) \wedge$

$calcLocation(loc(x, y), loc(x^r, y^r)), bodyangle, depth + r) \vee$

$location(object, loc(x, y), s) \wedge \neg\exists(t, x^2, y^2)\, a = endMove(ob, loc(x, y), loc(x^2, y^2), t).$

The successor state axiom for moving (**??**) shows whether or not an object is currently moving in the domain. An object is considered moving from $loc(x^1, y^1)$ to $loc(x^2, y^2)$ iff an action, $startMove$ has occurred in the current situation $do(a, s)$. Or if the object was moving in the previous situation, s, and no action, $endMove$, has occurred that has already moved the object to its destination $loc(x^2, y^2)$.

$$
\begin{aligned}
&moving(object, loc(x^1, y^1), loc(x^2, y^2), do(a, s)) \equiv \\
&\quad \exists(t)\, a = startMove(object, loc(x^1, y^1), loc(x^2, y^2), t) \vee \\
&\quad moving(object, loc(x^1, y^1), loc(x^2, y^2), s) \wedge \\
&\qquad \neg\exists(t)\, a = endMove(object, loc(x^1, y^1), loc(x^2, y^2), t).
\end{aligned}
\tag{2.12}
$$

The successor state axiom for rotating (**??**) describes if the robot is rotating around its axis of symmetry. The SSA states that the robot is rotating around its axis of symmetry iff an action, $startPan$, has occurred in the current situation. Or the robot is rotating iff it was rotating in the previous situation, s, and the action, $endPan$, has not occurred.

$$
\begin{aligned}
&rotating(\omega, do(a, s)) \equiv \\
&\quad \exists(t)\, a = startPan(\omega, t) \vee \\
&\quad rotating(\omega, s) \wedge \neg\exists(t) a = endPan(\omega, t).
\end{aligned}
\tag{2.13}
$$

The previously explained SSA all describe specifically what an object is doing or where an object is in the domain. We are able to tell the location of an object based on its movements or sensing data using the SSA for location, (**??**). We can tell if the robot is rotating, or which direction it is facing using the SSA for facing, (**??**), or the SSA for rotating, (**??**). Finally we can tell if an object is moving using the SSA for moving, (**??**). The next group of SSA axioms are used to describe attributes of depth peaks in the current situation that are changed due to actions. Using these axioms we can ascertain facts about object's peak attributes, depth, size and left boundary distance.

The SSA for the fluent $depth$ is used to determine the attribute, depth, for the peak of an object. The fluent holds if a sensing action occurred in the current situation. The depth, $depth$, of the associated object, $body$, will be the depth obtained from the peak of that body in the depth profile, $profile$. Otherwise, if the robot, or object moved to a new position, then the new depth is calculated based on the current location of the robot and the object. This is calculated using the Euclidean distance from the location of the robot to the nearest point on the object. The formula for Euclidean distance, $euclDist$, can be found in Section **??**, see Formula **??**. Otherwise the axiom states that the depth will remain the same as it was in the previous situation, s, if no action occurred that would change the depth.

$$
\begin{aligned}
&depth(peakOf(body, depth, size, lbd), depth, loc(x^r, y^r), do(a, s)) \equiv \\
&\quad \exists(t, profile) a = sense(profile, loc(x^r, y^r), t) \wedge \\
&\qquad peakOf(body, depth, size, lbd) \in profile \vee \\
&\quad \exists(x^1, y^1, t, x, y, r, euclDistance) \\
&\qquad a = endMove(robot, loc(x^1, y^1), loc(x^r, y^r), t) \wedge location(body, loc(x, y), s) \wedge \\
&\qquad location(robot, loc(x^1, y^1), s) \wedge radius(body, r) \wedge \\
&\qquad euclDist(loc(x, y), loc(x^r, y^r), r, euclDistance) \wedge depth = euclDistance \vee \\
&\quad \exists(x^1, y^1, x^2, y^2, t, r, euclDistance) \\
&\qquad a = endMove(body, loc(x^1, y^1), loc(x^2, y^2), t) \wedge location(body, loc(x^1, y^1), s) \wedge \\
&\qquad location(robot, loc(x^r, y^r), s) \wedge radius(body, r) \wedge
\end{aligned}
\tag{2.14}
$$

$$
\begin{aligned}
&euclDist(loc(x^2, y^2), loc(x^r, y^r), r, euclDistance) \wedge depth = euclDistance \vee \\
&depth(peakOf(body, depth, size, lbd), depth, loc(x^r, y^r), s) \wedge \\
&\quad \exists(x, y) location(body, loc(x, y), s) \wedge location(robot, loc(x^r, y^r), s) \wedge \\
&\quad \{(\neg\exists(x^n, y^n, t, x, y, x^1, y^1, profile, depth', size', lbd') \\
&\qquad a = endMove(robot, loc(x^r, y^r), loc(x^n, y^n), t) \vee \\
&\qquad a = endMove(body, loc(x, y), loc(x^1, y^1), t) \vee \\
&\qquad (a = sense(profile, loc(x^r, y^r), t) \wedge \\
&\qquad peakOf(body, depth', size', lbd') \in profile \wedge depth \neq depth')\}.
\end{aligned}
$$

The SSA for size is used to determine the attribute, $size$, for the peak of an object. The fluent holds if a sensing action occurred in the current situation, then the size, $size$, of the associated object, $body$, is the size obtained from the peak of that body in the depth profile, $profile$. Otherwise, if the robot or object moved to a new position, then the new size is calculated based on the current location of the robot and the object. The new size is calculated using the Euclidean distance from the location of the robot to the two tangent points on the object. Subsequently, the angle between the location of the observer and the two tangent points are used to calculate the size of the object. The frame axiom states that the size will remain the same as it was in the previous situation, s, if no action occurred that would change the size. Figure **??** exemplifies this equation, $size = 2 * asin(r/(euclDistance + r)) * 180/\pi$ from the SSA for $size$. We use the distance to the center for the body, $euclDistance + r$, and the distance to a tangent point, r, to create a right triangle. With a right triangle we can use trigonometry, in this case the relationship sin to calculate half of the angular distance. Since we are using circles to represent objects, the calculation can be mirrored for the second half of the angular size. Therefore, in this case, $size = 2 * asin(r/(euclDistance + r)) * 180/\pi$, will provide the angular size of an object in relation to its distance from the observer.

$$
\begin{aligned}
&size(peakOf(body, depth, size, lbd), size, loc(x^r, y^r), do(a, s)) \equiv \\
&\quad \exists(t, profile)\, a = sense(profile, loc(x^r, y^r), t) \wedge \\
&\qquad peakOf(body, depth, size, lbd) \in profile \vee
\end{aligned}
\tag{2.15}
$$

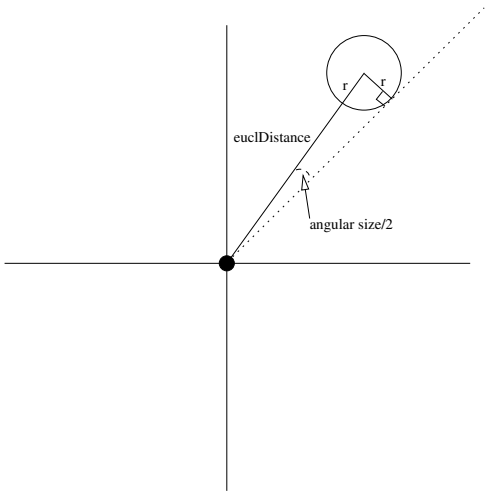

Figure 2.5: Using Euclidean Distance and Radius of a Body to Calculate Angular Size

$$\exists(x^1, y^1, t, x, y, r, euclDistance)$$
$$a = endMove(robot, loc(x^1, y^1), loc(x^r, y^r), t) \wedge radius(body, r) \wedge$$
$$location(body, loc(x, y), s) \wedge location(robot, loc(x^1, y^1), s) \wedge$$
$$euclDist(loc(x, y), loc(x^r, y^r), r, euclDistance) \wedge depth = euclDistance \wedge$$
$$size = 2 * asin(r/(euclDistance + r)) * 180/\pi \ \vee$$
$$\exists(x^1, y^1, x^2, y^2, t, r, euclDistance)$$
$$a = endMove(body, loc(x^1, y^1), loc(x^2, y^2), t) \wedge radius(body, r) \wedge$$
$$location(body, loc(x^1, y^1), s) \wedge location(robot, loc(x^r, y^r), s) \wedge$$
$$euclDist(loc(x^2, y^2), loc(x^r, y^r), r, euclDistance) \wedge depth = euclDistance \wedge$$
$$size = 2 * asin(r/(euclDistance + r)) * 180/\pi \ \ \vee$$
$$size(peakOf(body, depth, size, lbd), size, loc(x^r, y^r), s) \wedge$$
$$\exists(x, y) location(body, loc(x, y), s), location(robot, loc(x^r, y^r), s) \wedge$$
$$\{(\neg\exists(x^n, y^n, t, x^1, y^1, profile, depth', size', lbd')$$
$$a = endMove(robot, loc(x^r, y^r), loc(x^n, y^n), t) \vee$$
$$a = endMove(body, loc(x, y), loc(x^1, y^1), t) \vee$$
$$(a = sense(profile, loc(x^r, y^r), t) \wedge$$

$$peakOf(body, depth', size', lbd') \in profile \wedge size \neq size')\}.$$

The SSA for left boundary distance, lbd, is used to determine the attribute, distance from the left boundary, for the peak of an object. The fluent holds if a sensing action occurred in the current situation, then the left boundary distance, lbd, of the associated object, $body$, is the left boundary distance obtained from the peak of that body in the depth profile, $profile$. Otherwise, if the robot or object moved to a new position, then the new size is calculated based on the current location of the robot and the object. This is calculated relative to the direction the observer is facing. The left boundary distance is the angle from the left limit of the observer's field of view to the closest tangent point on the object from the observer. Otherwise, if the action observed was the robot rotating, then the new left boundary distance is calculated from the left boundary distance in the previous situation, s, and the angle of rotation that has been completed. Otherwise, the axiom states that the left boundary distance will remain the same as it was in the previous situation, s, if no action occurred that would change the left boundary distance. The formula $calcAngle$ is further explained in Section **??**.

$$\begin{aligned}
&lbd(peakOf(body, depth, size, lbd), lbd, loc(x^r, y^r), do(a, s)) \equiv \\
&\quad \exists(t, profile) a = sense(profile, loc(x^r, y^r), t) \wedge \\
&\qquad peakOf(body, depth, size, lbd) \in profile \vee \\
&\quad \exists(x^1, y^1, t, x, y, \theta, alpha, bodyangle) \\
&\qquad a = endMove(robot, loc(x^1, y^1), loc(x^r, y^r), t) \wedge \\
&\qquad location(body, loc(x, y), s) \wedge location(robot, loc(x^1, y^1), s) \wedge \\
&\qquad facing(robot, \theta, s) \wedge fieldView(alpha) \wedge \\
&\qquad calcAngle(bodyangle, x, y, x^r, y^r) \wedge lbd = (\theta + alpha/2 - bodyangle) \wedge \\
&\qquad lbd >= 0 \wedge lbd =< alpha \vee \\
&\quad \exists(x^1, y^1, x^2, y^2, t, \theta, alpha, bodyangle) \\
&\qquad a = endMove(body, loc(x^1, y^1), loc(x^2, y^2), t) \wedge \\
&\qquad location(body, loc(x^1, y^1), s) \wedge location(robot, loc(x^r, y^r), s) \wedge \\
&\qquad facing(robot, \theta, s) \wedge fieldView(alpha) \wedge
\end{aligned} \tag{2.16}$$

$$
\begin{aligned}
&calcAngle(bodyangle, x^2, y^2, x^r, y^r) \wedge lbd = (\theta + alpha/2 - bodyangle) \wedge \\
&lbd >= 0 \wedge lbd =< alpha \vee \\
&\exists(x, y, \theta, alpha, depth', size', lbd', \omega, t) a = endPan(\omega, t) \wedge \\
&location(robot, loc(x^r, y^r), s) \wedge location(body, loc(x, y), s) \wedge \\
&facing(robot, \theta, s) \wedge fieldView(alpha) \wedge \\
&lbd(peakOf(body, depth', size', lbd'), lbd', loc(x^r, y^r), s) \wedge lbd = lbd' + \omega \wedge \\
&lbd >= 0 \wedge lbd =< alpha \vee \\
&lbd(peakOf(body, depth, size, lbd), lbd, loc(x^r, y^r), s) \wedge \\
&\exists(x, y) location(body, loc(x, y), s), location(robot, loc(x^r, y^r), s) \wedge \\
&\{(\neg\exists(x^n, y^n, t, x^1, y^1, profile, depth', size', lbd', \omega) \\
&a = endMove(robot, loc(x^r, y^r), loc(x^n, y^n), t) \vee \\
&a = endMove(body, loc(x, y), loc(x^1, y^1), t) \vee \\
&a = endPan(\omega, t) \vee \\
&(a = sense(profile, loc(x^r, y^r), t) \wedge \\
&peakOf(body, depth', size', lbd') \in profile \wedge lbd \neq lbd')\}.
\end{aligned}
$$

This covers all the SSA axioms that describe the peaks of objects and the physical actions the objects and robot may perform in the domain. Using these axioms we are able to define a set of fluents that describe how the associated peaks may transition over time.

2.3 Golog

Golog is a high level language for solving planning problems and for specifying controllers. Behaviours are not specified completely, there are non-deterministic constructs that leave Golog programs partially unspecified. A Golog program is similar to an incomplete plan. Executing a non-deterministic Golog program is completing this plan with a sequence of actions. Golog programs are controllers designed to control actions of an agent. Due to non-determinism in Golog programs they do not prescribe every action that the agent is supposed to execute. Therefore, Golog programs can be considered as specifications for a set of plans and an agent chooses one

of the plans to execute that best helps them to reach predetermined goals. Although, designing controllers is the traditional use for Golog, we use it for different purposes (plan recognition) in Chapter 3 and 4.

Golog is a logic programming language for modelling complex behaviours. The program and complex action expressions defined in this section can be viewed as a programming language whose semantics are defined via macro-expansion into sentences of the situation calculus. This section is broken into three parts. First, we will discuss Golog and its evaluation semantics [**?**]. Next, we will explore ConGolog and its transition semantics[**?**]. Finally, we will explain, on a simple domain, how this logic programming language has been used before for incremental plan recognition[**?**].

2.3.1 Golog and Evaluation Semantics

Golog is a high level programming language based in the situation calculus for defining complex actions in terms of a set of primitive actions, [**?**]. All primitive actions, α are part of the background domain theory D. Primitive actions used in Golog are axiomatized in the situation calculus as described in the previous section. In Golog, a program δ can be a single primitive action, α, or it can be another construct. Golog has the following programming constructs:

1. Primitve Action α :

   This construct is used for executing action α in a Golog program δ.

2. Test Condition: $\phi?$

   Test if the condition ϕ holds in the current situation.

3. Sequence: $(\delta_1; \delta_2)$

   Execute δ_1 first then execute δ_2 second.

4. Nondeterministic choice between actions: $(\delta_1 | \delta_2)$

   Execute δ_1 or execute δ_2.

5. Nondeterministic choice of arguments: $\pi v.\delta$

   Nondeterministically pick a value for v and execute δ for that value. More specifically, v is

a variable in Golog, and δ is a construct that uses the variable v.

6. Nondeterministic iteration: δ^*

   Execute δ zero or more times.

7. Conditionals: $\mathbf{if}\ \phi\ \mathbf{then}\ \delta_1\ \mathbf{else}\ \delta_2$

   Execute δ_1 if ϕ is true or if ϕ is false execute δ_2.

8. While loops: $\mathbf{while}\ \phi\ \mathbf{do}\ \delta$

   Execute δ while ϕ is true

9. Procedures: $\{\mathbf{proc}\, P_1(\vec{v}_1)\delta_1\, \mathbf{end}\, ; \ldots \mathbf{proc} P_n(\vec{v}_n)\delta_n\, \mathbf{end}\, ; \delta\}$, (includes recursion)

Evaluation Semantics

Expressions in the Golog language can be thought of as macros that expand into genuine formulas of the situation calculus. We define the abbreviation $Do(\delta, s, s')$ where δ is a complex action expression. $Do(\delta, s, s')$ can be read as: it is possible to reach situation s' from situation s by executing a sequence of actions specified by δ. Since Golog includes nondeterministic constructs, there may be several different executions terminating in different situations. Subsequently, we define the expressions below that use the notation, $\stackrel{\circ}{=}$ to say that expressions on the left hand side expands into a formula on the right hand side. The symbol, $\stackrel{\circ}{=}$ in the following expressions is defined as a macro, where the left hand side of the equations are defined by the right hand side. Do is defined inductively on the structure of its first argument as follows;

1. **Primitve actions:**

   $Do(a, s, s') \stackrel{\circ}{=} Poss(a[s], s) \wedge start(s) \leq time(a) \wedge s' = do(a[s], s)$.

   The notation a[s] means the results of restoring the situation argument s to all functional fluents mentioned by the action term a. This is required because the situation s is suppressed in Golog programs. We have also added a temporal condition, where primitive actions have an additional time argument. The notation reads that the times of primitive actions leading from s to s' form a non-decreasing sequence.

2. **Test Conditions:**

   $Do(\phi?, s, s') \stackrel{\circ}{=} \phi[s] \wedge s = s'$

Here ϕ is a situation suppressed expression and $\phi[s]$ is the situation calculus formula obtained by restoring the situation variable s to all fluent names in ϕ. Restoring the situation variable means to evaluate all the fluents in ϕ using the current situation, s.

3. **Sequence**:
   $Do(\delta_1; \delta_2, s, s') \doteq (\exists s'')Do(\delta_1, s, s'') \wedge Do(\delta_2, s'', s')$.

4. **Nondeterministic choice between actions:**
   $Do(\delta_1|\delta_2, s, s') \doteq Do(\delta_1, s, s') \vee Do(\delta_2, s, s')$.

5. **Nondeterministic choice of arguments:**
   $Do((\pi v)\delta(v), s, s') = (\exists v)Do(\delta(v), s, s')$.

6. **Nondeterministic iteration:**
   $Do(\delta^*, s, s') \doteq$
   $(\forall P).\{(\forall s_1)P(s_1, s_1) \wedge (\forall s_1, s_2, s_3)[Do(\delta, s_1, s_2) \wedge P(s_2, s_3) \supset P(s_1, s_3)]\} \supset P(s, s')$.

7. **Conditionals:**
   $\mathbf{if}\ \phi\ \mathbf{then}\ \delta_1\ \mathbf{else}\ \delta_2\ \mathbf{endIf} \doteq [\phi?; \delta_1]|[\neg\phi?; \delta_2]$

8. **While Loops:**
   $\mathbf{while}\ \phi\ \mathbf{do}\ \delta\ \mathbf{endWhile} \doteq [\phi?; \delta]^*\neg\phi?$

9. **Procedures:** First we define an auxiliary macro definition, For any predicate variable P of arity $n + 2$, that has situation variables as the last two arguments
   $Do(P(t_1, \ldots, t_n), s, s') \doteq P(t_1[s], \ldots, t_n[s], s, s')$.
   Expressions of the structure $P(t_1, \ldots, t_n)$ occurring in programs will serve as procedure calls. $Do(P(t_1, \ldots, t_n), s, s')$ will mean executing the procedure P on actual parameters $t_1, \ldots, t_n$ causes a transition from situation s to s'. Next, suppose a program consists of a sequence of declarations of procedures $P_1, \ldots P_n$ with formal parameters $\vec{v_1}, \ldots, \vec{v_n}$ and procedure bodies $\delta_1, \ldots \delta_n$ respectively, followed by a main program body δ_0. Here $\delta_1, \ldots, \delta_n, \delta_0$ are complex actions, extended by actions for procedures calls as described above. Therefore, a program will have the form
   $\mathbf{proc}P_1(\vec{v_1})\delta_1\mathbf{endProc}\,; \ldots \mathbf{proc}P_n(\vec{v_n})\delta_n\mathbf{endProc}; \delta_0$

and is defined as follows:

$Do(\{\mathbf{proc}\, P_1(\vec{v_1})\delta_1\, \mathbf{endProc}\,; \ldots \mathbf{proc}\, P_n(\vec{v_n})\delta_n\, \mathbf{endProc}; \delta_0\}, s, s') =$

$(\forall P_1, \ldots, P_n).[\bigwedge_{i=1}^{n}(\forall s_1, s_2, \vec{v_i}).Do(\delta_i, s_1, s_2) \supset P_i(\vec{v_i}, s_1, s_2)] \supset Do(\delta_0, s, s')$

Its situation calculus macro expansion states the following, when $P_1, \ldots, P_n$ are the smallest binary relations on situations that are closed under the evaluation of their procedure bodies $\delta_1, \ldots, \delta_n$ then any transition (s, s') obtained by evaluating the main program δ_0 is a transition for the evaluation of the program.

To evaluate a program we must prove that using the situation calculus axioms D is a logical consequence of the definition of the domain, i.e. $D \models \exists s Do(program, S_0, s)$. Any binding for s obtained by a constructive proof of this sentence is an execution trace, in terms of the primitive actions, of a program.

Example The following is a recursive Golog program for the domain proposed in this thesis. The program, $snuggleUp(obj)$ will move a robot to within a certain range of an object moving smaller amounts as it gets closer. Note, the expression 'nil' changes nothing and is introduced as a convenience in explanation.

$\mathbf{proc}\, snuggleUp(obj)$

$\pi(d, size, lbd, loc(x^r, y^r), loc(x^b, y^b))(depth(peakOf(obj, d, size, lbd), d, loc(x^r, y^r)) \wedge$

$location(obj, loc(x^b, y^b)))?;$

$(\pi t, t')[\mathbf{if} d > 5\, \mathbf{then}$

$startMove(robot, loc(x^r, y^r), loc((x^b - x^r)/2, (y^b - y^r)/2), t);$

$endMove(robot, loc(x^r, y^r), loc((x^b - x^r)/2, (y^b - y^r)/2), t'); snuggleUp(obj)$

$\mathbf{else}\, nil$

$\mathbf{endIf}\,]$

$\mathbf{endProc}$

This program is executed by proving $(\exists s) Do(snuggleUp(obj), S_0, s)$, using background axioms defined earlier. The symbol π is a unique Golog construct that represents variable initializa-

tion, such that the variables initialized by π can be defined later in the program. The combination, $\pi(arguments)\ predicate(arguments)?$, is a common Golog programming idiom convenient in many Golog programs. The intention is to bind arguments to those values that will be determined from evaluating predicate as above (or a logical formula in more complex cases). When a test statement, $predicate(arguments)?$, is evaluated, the values of arguments that make this query true will be returned back to the Golog program for subsequent execution. In this particular example, the predicates, $depth$ and $location$, will be evaluated by connecting to the simulator, and values of their arguments obtained from this evaluation will be used subsequently in the provided Golog code snippet. Note, that in Golog test expressions, the fluents, $depth$ and $location$, do not have any situational argument, but the current situation is provided by the Golog interpreter when tests are evaluated. If the robot manages to get within 5 units of the object then the program will terminate. Otherwise it nondeterministically chooses a time t and t' for the start and end move actions as long as $t < t'$, and calls its program again, $snuggleUp(obj)$. After one iteration the new situation term would be the initial situation, S_0 with the two performed actions added to the term:

$$do(endMove(robot, loc(x^r, y^r), loc((x^b - x^r)/2, (y^b - y^r)/2), t'),$$
$$do(startMove(robot, loc(x^r, y^r), loc((x^b - x^r)/2, (y^b - y^r)/2), t), S_0))$$

Then, the program will call itself recursively until the appropriate distance from the object is reached. It should be noted that Golog and the Do interpreter is an offline execution environment and the execution trace of a Golog program should be passed to an execution module for the online execution.

Although previously we have defined a Golog syntax for logical formulas, throughout this paper we use the Prolog notation used to render Golog programs. We chose a Prolog implementation of Golog because all previous Golog interpreters are implemented in Prolog. We will define below the equivalent Prolog syntax for the Golog constructs that will be used throughout this thesis, especially in Chapter 4 when defining our plan recognition behaviours.

Construct Name	Golog Construct	Equivalent Prolog Notation
Test Conditions	$\phi?$	$?(\phi)$
Sequence	$\delta_1 : \delta_2$	$\delta_1 : \delta_2$
Nondeterministic Choice	$\delta_1 \mid \delta_2$	$\delta_1 \# \delta_2$
Nondeterministic Argument Choice	$(\pi v)\delta(v)$	$pi(v, \delta)$
Conditionals	$\mathbf{if}\ \phi\ \mathbf{then}\ \delta_1\ \mathbf{else}\ \delta_2\ \mathbf{endIf}$	$if(\phi, \delta_1, \delta_2)$
Procedures	$proc(p, \delta)$	$proc(p, \delta)$

2.3.2 ConGolog and Transition Semantics

ConGolog and Transition Semantics are an extension of Golog and evaluation Semantics, [**?**]. They allow single step computation in contrast to Golog defining complete computations of programs. We are going to define a relation, $Trans(\delta, s, \delta', s')$ that associates a program δ and situation s, with a new situation s' that results from executing a primitive action or test action and a new program δ' that represents what remains of the original program δ after having performed such an action. This was first put forward in [**?**] The following set of axioms characterizes the predicate $Trans$ by induction on the structure of the program term δ_1.

1. Empty program: $Trans(Nil, s, \delta', s') \triangleq False$
2. Primitive Actions:
   $Trans(a, s, \delta', s') \triangleq Poss(a, s) \wedge \delta' = Nil \wedge s' = do(a, s) \wedge start(s) \leq time(a)$,
3. Test Conditions: $Trans(\phi?, s, \delta', s') \triangleq \phi[s] \wedge \delta' = Nil \wedge s' = s$
4. Sequence : $Trans(\delta_1; \delta_2, s, \delta', s') \triangleq \exists\gamma.Trans(\delta_1, s, \gamma, s') \wedge \delta' = \gamma; \delta_2$
   $\vee Final(\delta_1, s) \wedge Trans(\delta_2, s, \delta', s')$
5. Nondeterministic choice between actions: $Trans(\delta_1 | \delta_2, s, \delta', s') \triangleq Trans(\delta_1, s, \delta', s') \vee$
   $Trans(\delta_2, s, \delta', s')$
6. Nondeterministic choice of arguments: $Trans(\pi v.\delta(v), s, \delta', s') \triangleq \exists x.Trans(\delta_x^v, s, \delta', s')$
7. Nondeterministic iteration: $Trans(\delta^*, s, \delta', s') \triangleq \exists\gamma.Trans(\delta, s, \gamma, s') \wedge \delta' = \gamma; \delta^*$

8. Conditionals: $Trans(\mathbf{if}\ \phi\ \mathbf{then}\ \delta_1\ \mathbf{else}\ \delta_2, s, \delta', s') \triangleq$
   $\phi[s] \wedge Trans(\delta_1, s, \delta', s') \vee \neg\phi[s] \wedge Trans(\delta_2, s, \delta', s')$

9. While Loops: $Trans(\mathbf{while}\ \phi\ \mathbf{do}(\ \delta, s, \delta', s')) \triangleq$
   $\exists\gamma.Trans(\delta, s, \gamma, s') \wedge \phi[s] \wedge (\delta' = \gamma; \mathbf{while}\ \phi\ \mathbf{do}\ \delta)$

$Trans$ denotes a transition relation between configurations. It lets us know which is a possible next step in computation, returning the remaining program δ' and next situation, s'. We omit in the above definitions constructs included only in ConGolog as they are not used in this work. Also omitted are axioms for procedures as they are too complex, but see [**?**] for details. We now introduce the predicate $Final(\delta, s)$, which states that the configuration (δ, s) is final or there remains no more program to be executed. The final situations reached after a finite number of transitions from a starting situation coincide with those satisfying the Do relation. Complete computations are thus defined by repeatedly composing single transitions until a final configuration is reached.

1. Empty Program: $Final(Nil, s) \triangleq True$

2. Primitive action: $Final(a, s) \triangleq False$

3. Test Conditions: $Final(\phi?, s) \triangleq False$

4. Sequence: $Final(\delta_1; \delta_2, s) \triangleq Final(\delta_1, s) \wedge Final(\delta_2, s)$

5. Nondeterministic choice between actions: $Final(\delta_1|\delta_2, s) \triangleq Final(\delta_1, s) \vee Final(\delta_2, s)$

6. Nondeterministic choice of arguments: $Final(\pi v.\delta, s) \triangleq \exists x.Final(\delta_x^v, s)$

7. Nondeterministic iteration: $Final(\delta^s, s) \triangleq True$

8. Conditionals: $Final(\mathbf{if}\ \phi\ \mathbf{then}\ \delta_1\ \mathbf{else}\ \delta_2, s) \triangleq \phi[s] \wedge Final(\delta_1, s) \vee \neg\phi[s] \wedge Final(\delta_2, s)$

9. While Loops: $Final(\mathbf{while}\ \phi\ \mathbf{do}\ \delta, s) \triangleq Final(\delta, s) \vee \neg\phi[s]$

This set of assertions defines when a configuration (δ, s) can be final, when no more program remains to be executed. There are also axioms for procedures, see [**?**] for details.

Incremental Plan Recognition

In the paper [**?**], Goultiaeva modifies the transition semantics to match observed actions to a program library. Therefore instead of incremental execution of actions from a program by a robot, the observed actions are matched to a plan library allowing a user to match plans based on observed actions in the domain. It is important to note that this works makes a strong assumption that a plan recognizing program can observe each action. In contrast, we do not make this assumption that all actions are observable.

Recognizing a plan means that given a sequence of observed actions, the system must be able to determine which plan(s) the user may be following. The framework used in the paper, [**?**] relies on a plan library, which details the possible plans as procedures in ConGolog. Given a sequence of actions the system should be able to provide the following information: the plan the user is currently following, what stage of the plan the user is in, what has been done and what remains to be done.

The additional details to the transition based semantics are not included here due to complexity but can be found in the paper, [**?**]. The important contributions adopted in our plan recognition system are the following: actions are performed independently, and plans are recognized incrementally.

First, actions are performed independently. Unlike previous descriptions of ConGolog and Golog, the procedures used do not cause an agent to perform actions. Agents in the domain are independent and are carrying out their own goals. The purpose of the plan recognition is to figure out the goal by matching what actions are observed to one of the procedures described in the plan library. Plan recognition using ConGolog is about observation and matching of actions to a plan library.

Second, plans are recognized incrementally. The ability to recognize plans incrementally allows us to recognize plans online and in real-time. It is much simpler to perform plan recognition offline. Recognizing plans online requires partial matching of observed actions to the plan library. At any given time, more than one possible plan could be in the process of execution as there is not complete information about the goal state of the agent. These two major contributions are used to build a plan recognition system for a motion domain. Our system is built to observe changes in

fluents that characterize objects in the domain and matches these observations to Golog programs in a plan library. The main conceptual departure from [**?**] is that our system does not know what actions other objects (agents) are doing, but can only sense changes in their spatial attributes.

2.4 Regression

Regression is one of the central reasoning algorithms of the situation calculus. Regression forms the basis for a planning procedure and for the automated reasoning about effects of actions in the situation calculus. The logic underlying regression is explored in the following section.

Supposed we want to prove that a sentence W is entailed by some BAT(basic action theory), D, and furthermore we suppose that W mentions a relational fluent atom $F(\vec{t}, do(\alpha, \sigma))$, where we have the successor state axiom for F as $F(\vec{x}, do(\alpha, \sigma)) \equiv \Phi_F(\vec{x}, \alpha, \sigma)$. Using this we can easily determine a logically equivalent sentence W' by the substitution of $\Phi_F(\vec{t}, \alpha, \sigma)$ for $F(\vec{t}, do(\alpha, \sigma))$ in W.

After this substitution, the fluent atom $F(\vec{t}, do(\alpha, \sigma))$, involving the complex situation term $do(\alpha, \sigma)$ has been replaced in W in favour of $\Phi_F(\vec{t}, \alpha, \sigma)$ which involves the simpler situation term σ. In this regressive sense W' is "closer" to the initial situation S_0 than was W.

More so, this operation can be repeated until the resulting goal formula mentions only the situation term S_0, after which it should be sufficient to check the resulting goal using only sentences from the initial database, D_{S_0} (see Regression Theorem in Section **??**).

Regression is the mechanisms we will use to repeatedly perform the above operations. Starting with a goal W we repeatedly perform the reduction until we have obtained the logically equivalent W_0, whose only situation term is S_0. An analogous approach can be used for reducing functional fluent terms. But we only consider the methodology for relational fluent terms, as we only use relational fluents in our work.

2.4.1 Regression Theorem

In the following section we fully explore the idea of regression as repeatedly performing simplifying operations. We move on to establish that a proof of the resulting regressed sentence needs to only appeal to the initial database together with unique name axioms for actions. We define the

convenient abbreviations below in preparation for writing long situation terms.

Abbreviation: $do([a_1, \ldots, a_n], s)$.

$$do([\,], s)] = s \; if \; n = 0$$
$$do([a_1, \ldots, a_n], s) = do(a_n, do(a_{n-1}, \ldots do(a_1, s) \ldots)) \; , n \in 1, 2, \ldots$$

$do([a_1, \ldots, a_n], s)$ is the compact notation for the situation term $do(a_n, do(a_{n-1}, \ldots do(a_1, s) \ldots))$ denoting the situation resulting from action a_1, followed by $a_2, \ldots$, followed by a_n beginning in the situation s, being performed, [**?**].

The Regressable Formulas: A formula W of $L_{sitcalc}$ is regressable iff

1. Each term of sort $situation$ mentioned by W has the syntactic form $do([\alpha_1, \ldots, \alpha_n], S_0)$ for some $n \geq 0$, where $\alpha_1, \ldots, \alpha_n$ are of sort action.

2. For each atom of the form $Poss(\alpha, \sigma)$ mentioned by W, α has the form $A(t_1, \ldots, t_n)$ for some n-ary action function symbol A of the $L_{sitcalc}$.

3. W does not quantify over situations.

4. W does not mention the predicate symbol $\sqsubset$, nor does it mention any equality atom $\sigma = \sigma'$ for terms σ,σ' of sort situation.

A regressable formula is characterized by the fact that each of its $situation$ terms is rooted at S_0 and through inspection of the situation term how many actions were performed. Also, when the regressable formula contains a $Poss$ atom, we can tell, through inspection of the $Poss$ atom what is the action function symbol of the atom (its first argument). As our work only contains relational fluents we define a simple version of regression next, but it is possible to expand this to include functional fluents, see [**?**].

2.4.2 The Regression Operator

In defining the regression operator we assume that W is a regressable formula of the $L_{sitcalc}$ and that W does not contain any function fluents. The $regression\ operator\ R$ applied to W is governed by a BAT of the $L_{sitcalc}$ that serves as the background axiomatization. In the following definitions:

- I is a tuple of terms
- α is a term of sort $action$
- σ is a term of sort $situation$ and
- W is a regressable formula of the $L_{sitcalc}$ with no functional fluents

1. We assume W is an atom. Because W is regressable there are four possibilities
   (a) W is a situation independent axiom such as an equality between terms of the same sort, $object$ or $action$. Or W is atom whose predicate symbol is not a fluent, then:
   $R[W] = W$.
   (b) W is a relational fluent of the form $F(\vec{t}, S_0)$, then:
   $R[W] = W$.
   (c) W is a regressable $Poss$ atom with the form $Poss(A(\vec{t}), \sigma)$ where $A(\vec{t})$ is of sort $action$ and σ is of sort $situation$. Here A is an action function symbol of the $L_{sitcalc}$. Therefore there must be an action precondition axiom for A with the form
   $Poss(A(\vec{x}), s) \equiv \Pi_A(\vec{x}, s)$
   with the assumption that all, if any, quantifiers of $\Pi_A(\vec{x}, s)$ have had their quantified variables renamed to be distinct from the free variables of $Poss(A(\vec{x}), s)$, then:
   $R[W] = R[\Pi_A(\vec{t}, \sigma)]$
   (d) W is a relational fluent atom of the form $F(\vec{t}, do(\alpha, \sigma))$, then let F's SSA in D_{ssa} be
   $F(\vec{x}, do(a, s)) \equiv \Phi(\vec{x}, a, s)$
   We assume that all quantified variables of Φ have had their quantified variables distinctly renamed from the free variables of F, then:
   $R[W] = R[\Phi_F(\vec{t}, \alpha, \sigma)]$
2. for non-atomic arguments regression is defined inductively
   $R[\neg W] = \neg R[W]$
   $R[W_1 \wedge W_2] = R[W_1] \wedge R[W_2]$
   $R[\exists v W] = \exists v R[W]$

In general the regression operator eliminates $Poss$ atoms in favour of their action precondition axiom definitions and replaces fluent atoms about $do(\alpha, \sigma)$ by logically equivalent expressions given by their SSA. It will repeatedly do this until it can't make any more replacements. The resulting formula will only mention the situation term S_0. Also, because regression only substitutes logically equivalent formulas for atoms, after the operations are complete, the final formula will be equivalent to the beginning.

2.4.3 Regression Theorem

Regression Theorem: We suppose W is a regressable sentence of the $L_{sitcalc}$ with no functional fluents, and D is a BAT, then:

1. $R_D[W]$ is a sentence uniform in S_0
2. $D \models R_D[W]$ iff $D_{una} \cup D_{S_0} \models R_D[W]$

2.4.4 Regression Applied

Regression in AI is useful in two standard settings. First, proving that a given action sequence is executable and second, solving the $projection\ problem$. We consider each of these in turn. We provide below a regression-based method for computing whether a grounded situation is executable (a grounded term has no free variables).

Given the basic action theory D, for each $n \geq 0$, it is convenient to introduce the following abbreviation.

$$(\forall a_1, \ldots, a_n).executable(do[a_1, \ldots, a_n], S_0) \stackrel{\circ}{=} \bigwedge_{i=1}^{n} Poss(a_i, do(a_1, \ldots, a_{i-1}], S_0)).$$

i.e. a sequence of actions $[a_1, \ldots, a_n]$ is executable if each action consecutively is possible to execute in the situation that results from executing previous actions. From this and our regression theorem we obtain,

Corollary: Suppose $\alpha_1, \ldots, \alpha_n$ is a sequence of ground action terms of $L_{sitcalc}$ then:

$$D \models executable(do[\alpha_1, \ldots, \alpha_n], S_0)$$

iff

$$D_{S_0} \cup D_{una} \models \bigwedge_{i=1}^{n} R[Poss(\alpha_i, do([\alpha_1, \ldots, \alpha_{i-1}], S_0))].$$

This corollary provides a systematic regression-based method for determining whether a ground situation term is executable. It is impressive that this method reduces the problem to a theorem-proving task in the initial database, D_{S_0} together with unique name axioms for actions. Furthermore, we know that this regression based method to a simple theorem proving task in D_{S_0} is faster than resolution, even with the best theorem provers (e.g.: Vampire) reducing executability on types of problems that can be expressed in the situation calculus.

2.4.5 Projection

The projection problem is, given a sequence of ground action terms and a formula G determine whether G is true in the situation resulting from the performance of these actions. We let D be a BAT (Basic Action Theory). Let $\alpha_1, \ldots, \alpha_n$ be a sequence of ground action terms and $G(s)$ be a regressable formula with only s as a free variable and whose only situation term is s. We want to determine whether

$$D \models G(do([\alpha_1, \ldots, \alpha_n], S_0)). \tag{2.17}$$

To solve the projection problem it is both necessary and sufficient to regress $G(do([\alpha_1, \ldots, \alpha_n], S_0)$ and ask whether the resulting sequence is entailed by the initial database and the unique names for actions. More specifically we want to determine the truth of

$$D_{una} \cup D_{S_0} \models R_D[G(do([\alpha_1, \ldots, \alpha_n], S_0))]. \tag{2.18}$$

Planning and database query are two applications where the projection problem is encountered naturally and where this regression based approach is useful. In planning we try to verify that a proposed plan $do([\alpha_1, \ldots, \alpha_n], S_0)$ satisfies a goal G, showing that the axioms entail the projection query $G(do([\alpha_1, \ldots, \alpha_n], S_0))$. In database query evaluation we try to answer a query G against a transaction sequence, answering the projection query G in the resulting situation.

2.5 Plan Recognition

In previous parts of this chapter we have covered the techniques we will use to approach plan recognition. We will now look at some other approaches to plan recognition systems and cognitive visions systems mostly related to the traffic domain. Two major differences can be spotted in previous works First is a central focus on vision systems. A lot of work has focused on developing robust visions systems that can translate real world data into computer data. In this thesis we focus on creating a high level description of how to understand this computer data if it is available.

Also, previous work has focused on dynamic scenes from a static view point. We focus on a dynamic scene from the standpoint of a dynamic observer. This creates a lot more complexity as the view point is constantly moving as well as the agents being reasoned about in the domain.

2.5.1 Cognitive Vision Systems

In this section, I will discuss the work done by H.H. Nagel, [**?**, **?**] in his research towards a cognitive vision system. Nagel spent over thirty years on producing a cognitive vision system. While a lot of his early work was limited by hardware he has produced many concepts that are still relevant today.

The first step in producing an intelligent vehicle tracking system is how to represent your scene. It is very complex to understand how a human can view a scene and efficiently transfer this understanding into a computationally efficient and usable definition. Nagel proposes natural language descriptions to define simple manoeuvres. These can be defined by geometric movements in the domain. Nagel introduces a fuzzy logic concept to define individual actions. This can be more easily understood as qualitative descriptors for actions. Using qualitative descriptors you are able to encode more data with less memory. The idea of qualitative descriptions is a very important concept that is seen in most related work.

Representing individual actions is one part of the understanding vehicle movement. The next step is to combine these individual actions to define more complex behaviours. Nagel introduces the idea of incremental recognition of complex behaviours. The system has a priori knowledge about vehicle manoeuvres, and these are defined as trees. Every time an action is observed it can

be matched to a node in a tree. A set of actions that matches a behaviour will match a branch of the situation tree.

Assuming we have a vision system the next step toward a cognitive vision system is to recognize movement primitives. Movement primitives can be considered as elementary manoeuvres which can be performed by a vehicle and can be described by simple verb phrases. These simple verb phrases can be combined to create noun phrases referring to an agent. These noun phrases can begin to represent vehicle behaviours, such as turning and passing.

The system has to be able to incorporate a priori knowledge about which elementary manoeuvres can be concatenated and under which conditions. Such knowledge about vehicular behaviour is represented internally as a situation graph formed by situation nodes. A situation node combines a state representation scheme and an action scheme.

2.5.2 Qualitative Mapping

Lattner et al. [**?**, **?**, **?**, **?**] have developed a rich vocabulary for describing motion of bodies using both space and time. They have a heavy focus on qualitative descriptors for this. They have descriptors for describing how a single agent moves over time, and also for how a single agent moves in relation to other agents over time. The qualitative motion vocabulary includes terms like velocity, acceleration, relative speed, and driving direction. These are temporally and spatially affected as almost any of these concepts are affected both by location and time.

Some of their more interesting work is how they handled plan recognition. Traffic situation plans are realized as logical formulae of road scene and object descriptors in connection with abstract validity intervals of qualitative motion predicates and their temporal relations. In their work plans can be formulas made up of a combination of logic symbols found in interval temporal logic (including negation, conjunction, disjunction, equivalence, and existential quantification). This work is especially interesting as similar connections can be made between recognizing plans in this work and how plans can be similarly developed in situation calculus. The following formula is an example of two mutually approaching vehicles in collision course:

$$HOLDS(collision\_course(v1, v2), i) =$$

$$
\begin{aligned}
&\exists dist : \\
&HOLDS(distance\_trend(v1, v2, decreasing\_distance), i) \\
&HOLDS(relative\_direction(v1, v2, opposite\_direction, i) \\
&HOLDS(relative\_lane(v1, v2, same\_lane), i) \\
&HOLDS(time\_distance(v1, v2, dist), i) \\
&dist < medium\_distance.
\end{aligned}
\tag{2.19}
$$

The logic being used is interval logic because of its ability to express temporal relations. These logic relations can easily be translated to the situation calculus using the available constructs in the situation calculus as well as replacing the holds predicate with fluents directly.

Another interesting concept is a mapper. The purpose of a mapper is to continually update a single qualitative descriptor over time. It does this by using a fuzzy interpreter from the geometrical representation to a qualitative description. A mapper has to tackle three challenges:

1. Acquisition of the needed sensory data from the environment
2. Definition of adequate qualitative equivalence classes for the respective motion descriptor
3. Online mapping from the sensory data to the motion descriptor.

For example, the Lateral Position Mapper examines the lateral position of a vehicle in its lane and its development as monotony segmentation. If a vehicle's center is located less than 30 cm to the lane's center then it is considered central in lane. Otherwise it is considered left or right in the lane respectively.

This work brings together a lot of good ideas, but uses them in an over simplified problem. It still remains to be seen if such techniques scale to dynamic environments. The idea of fuzzy logic mappers to convert sensory data is useful in describing ideas, such as distance and direction. Also, the concept of a mapper is similar to using depth profiles as an abstraction from numerical data as a way to represent the interaction of bodies. The work done here is important and we attempt to apply some of these techniques to more complex scenarios using the Situation Calculus.

2.5.3 Automated Learning Models

Cohn et al. [**?**, **?**, **?**] have worked on methods to automatically recognize and define semantically relevant spatio-temporal regions. They demonstrate the effectiveness of a spatio-temporal model in a qualitative event learning system. There are two major contributions, how to break down a scene into different spatial regions and also the automatic learning of events.

A scene is analyzed over time to describe areas of behavioural significance, such as a lane way. A tracking application is used to collect training data that provides the position and shape descriptions of moving objects as well as associating labels with object that are maintained throughout the object's lifetime in the scene. Object paths are constructed from the area covered by an object travelling in the domain. These paths are then merged into a database before statistical analysis indicates which entries are too infrequent to be included in the spatial model. Small regions for the spatial representations are obtained from the combination of the remaining paths stored in the database. This model is extended to include temporal information. When the database of paths is constructed, at regular time intervals points are recorded that can be used later to form regions which subdivide the paths into temporal regions. The velocity of objects as well as their distance from the camera affects the temporal regions. A main feature is that it takes the same amount of time for an object to traverse each temporal region. How an object reacts over space and time can be encoded in this manner automatically. Over a training sequence a number of spatial temporal paths are recorded and stored in the database. The idea is that if objects move in the same spaces over similar time intervals then they will be performing similar tasks.

Unlike most previous work event models are not provided as a priori information. Events models are created by observing a scene and applying an event learning model. The event learning model has five stages.

1. The same tracking process used above to track objects obtains shape description of moving objects.

2. A classification stage allows the identification of qualitative position and direction from information provided by the tracking application. This provides the spatial relationships of objects.

3. An object history generation uses an attention control mechanism to identify objects that are close and the qualitative relationships of those objects are added to the object history.

4. Object history verification is used to eliminate meaningless object histories. For example, if a history sequence refers to a relatively short interaction between two objects, then that interaction could be between elements of tracked nosie and not actual objects. Because of the large number of object relations, even the accidental pruning of some object relations does not affect the overall history. In some cases it strengthens it as it removes accidental noise affected relations that actually represent the relation between the two objects .

5. Finally is the event database revision. Each valid object history is added to the case base. On completion of the training period, statistical analysis can determine event models from object histories contained in the case base. The induced event models can be used to analyze subsequence new camera input which has been classified. Not only is it necessary to search for equivalent database entries, it is also necessary to search the database for entries that match a continuous subset of new entry. Such subsets represent simpler event patterns that compose the new event episode.

The most important aspects of this work is the reinforcement that closeness and relative location are important factors in describing vehicular movement. Depth profiles encode closeness and relative location of bodies in a domain. This type of work is similar to our goal in plan recognition, except we are working in the Situation Calculus using depth profiles as our form of knowledge representation. Although we are not using automated learning techniques in this thesis, the similarities between how we represent knowledge does allow the possibility of this method being applied in future work.

2.5.4 Bayesian Frameworks

Pynadath and Wellman, [**?**], present a general Bayesian framework encompassing the view that a plan should account for the context in which the plan was generated, including the mental state and planning process of the agent. They focus on how context can be exploited in plan recognition. Context in this case means the agent's initial mental state and domain situation. Unlike previous

methods that just look at observed effects, accounting for context means looking at the reasons behind the observed effects. The problem of plan recognition is to induce the plan of action driving an agent's behaviour, based on partial observation of its behaviour up to the current time. Deriving the underlying plan can be useful for many purposes, such as predicting the agent's future behaviour, interpreting its past behaviour, or generating actions designed to influence the plan itself. A common underlying theme of most plan recognition work is that the object to be induced is a plan and that this plan is the cause of observed behaviour. While previous work has emphasized the relationship between plans and their observable effects, they argue that it is equally necessary to consider the context under which the plan was executed.

Plans are considered to be structured linguistic objects. From the simple to the complex plan languages, the recognizer should be able to induce the plan from partial observations of the actions comprising the plan. Another feature of plans is that they are rational constructions. A rational agent has a mental state that can constrain the possible plans. An agent's mental state consists of its goals, beliefs and available actions. Beliefs are the agent's knowledge about the state of the world. Plans have associated with them probabilities that are related to the mental state of the agent and its probability to execute the plan given its mental state.

Observations of the state of the world provide two types of evidence about the plan being executed. First, the world influences the agent's initial mental state which provides the context for plan generation. Second, changes in the world state reflect the effects of the agent's actions, which result from executing the plan. Typically, one does not view an agent's actions directly; instead we view the effect of the agent's actions on the world. In this way, actions are only partially observable.

This work attempts to show how context is important in a plan recognition framework. Context provides intent for an agent when choosing the most probable plan. They use a traffic domain with clear goals (driver had an intended exit and a target driving speed) for the agent they are observing but do consider the idea of partial observations. They do provide a reasonable argument as to why context is important and show that it is beneficial in a real world plan recognition framework. This type of work can be used to get useful knowledge about complex environments that an agent is put into and needs to reason about unknown situations quickly.

In relation to this thesis, the idea of context is an interesting approach. Depending on a num-

ber of factors that aren't necessarily observed actions, it is possible to associate different plans with similar actions in different situations, e.g. highway driving vs. urban driving. Considering the mental state of the agent as well as observed context is very important in determining what behaviour is being performed by an agent. However, the generality and scalability of this framework remains to be seen. Actual real world examples may be too complex to express in this stated model. Therefore we do not use the ideas expressed in this work, but include it as an interesting addition for future work.

Chapter 3

Plan Recognition with Golog and Sensing

Introduction

The system that we have developed for plan recognition is composed of logical successor state axioms and precondition axioms that together provide input to recognition automata. In the first section, of this chapter, we will describe all logical successor state axioms about peaks. The precondition axioms were already discussed in Section **??**. In the second section we will discuss the system's architecture and how the changes in fluents over time act as input to the recognition automata. In the third section we will discuss all aspects of the finite state automata that will receive input from the fluents for plan recognition. We will explore the general structure of the finite state automata as well as give specific examples used in our system. We will conclude this chapter with a complexity analysis of the system.

3.1 Logical Successor State Axioms for Motion

In this section we will discuss all successor state axioms and the transitions they represent between attributes of single peaks. The SSA's for peaks can be found in the previous Section **??**. These transitions account for the perception of moving bodies. We will describe each axiom using its formal logic structure, as well as an English description of each axioms major features.

3.1.1 Logical Extending

Extending is the transition on a peak which states that the peak, $peakOf(body, depth, size, lbd)$, representing an object, $body$, is perceived from the robot location, $loc(x^r, y^r)$, as extending or increasing in angular size in the current situation, s. There are three possible actions that may cause the size of a peak to extend and these are explained in the logical formulas below, (**??**) - (**??**). As well the frame axiom is explained in the logical formulas (**??**) - (**??**).

The SSA for extending (**??**) states that a peak is observed to be extending in the situation, $do(a, s)$, if there was a sensing action (**??**) that observed the angular size, $size'$, as greater in the current situation, $do(a, s)$, than the angular size, $size''$, in the previous situation s. Also the fluent extending holds in the current situation if the robot (**??**) or object (**??**) moved to a position such that the calculated angular size, $size'$, of the object in the current situation, $do(a, s)$, is larger than the angular size, $size''$ in the previous situation s. In all three cases the depth, $depth'$ and $depth''$, in both situation $do(a, s)$ and s must be less than the limit boundary, l. The limit boundary is the furthest point noticeable by the robots sensors. In these cases it would represent a difference between extending and appearing, (**??**). If an object were to be outside the limit boundary in the previous situation, and then be inside the limit boundary in the current situation, the object would be perceived as appearing instead of extending, iff there was an increase in perceived angular size of the object.

In general, a frame axiom can be described as a logical statement saying that a fluent is true in situation $do(a, s)$ iff it was true in the previous situation s and the action a in the current situation did not have the effect of falsifying the fluent in the current situation. The formula (**??**) is the frame axiom for extending. The first line provides a truth value for the previous situation s. If true, the subformulas (**??**) - (**??**) check to see if the action a in the current situation $do(a, s)$ has made the SSA extending false. Subformula (**??**) states that if there was a sensing action then the perceived peaks angular size, $size''$ in situation $do(a, s)$ is not smaller than the angular size, $size'$ in the previous situation s. Subformulas (**??**) and (**??**) state that if the robot or the object moved to a new position such that the computed angular size, $size''$ of the object in situation $do(a, s)$ is not smaller than the angular size, $size'$ in the previous situation s. Also for all three cases the depth, $depth'$, in the previous situation, s, is not outside the limit boundary, l, and the depth, $depth''$ of the

object in the current situation, $do(a, s)$, is not outside the limit boundary, l, too. This axiom covers the cases that an action would make $extending$ false. When the current angular size is smaller than the angular size in the previous situation, this represents shrinking, which is the opposite of extending. If the previous depth was inside the limit boundary and the current depth in $do(a, s)$ is outside the limit boundary, this case corresponds to vanishing. The frame axiom states that if vanishing occurs it cannot also be extending. Finally, to make sure an object is extending and not appearing is covered by the fact in the frame axiom that the depth in the previous situation must be within the limit boundary. Appearing only occurs when the depth changes from outside the limit boundary to inside the limit boundary. The frame axiom for extending covers all the cases that could potentially falsify the fluent $extending$.

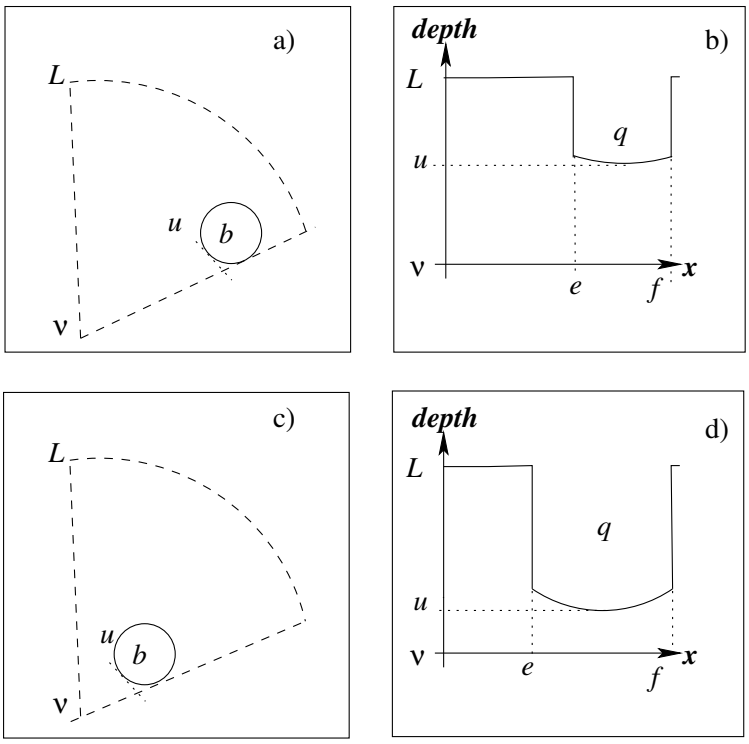

Figure 3.1: Figure of an Object's depth peak extending

Figure **??** is a graphical representation of how an object in motion may cause its associated depth peak to extend. The object, b, moves closer to the observer, v, and because of this motion the depth peak decreases in depth and increases in size. In **??**a) the object b is close to the right hand side of the observers field of view at a distance u. In **??**b) the corresponding depth peak graph is

shown. The depth u, left boundary distance e and angular size $f - e$ corresponds to the peak q created by object b in **??**a). In **??**c) the object b has moved closer to the observer v and as can be seen in **??**d) its depth peak has increased in angular size and decreased in depth and left boundary distance. An increase in angular size is indicative of an object extending.

$$extending(peakOf(body, depth', size', lbd'), loc(x^r, y^r), do(a, s)) \equiv \tag{3.1}$$

$$\{\exists(t, profile, depth'', size'', lbd'', l)a = sense(profile, loc(x^r, y^r), t) \wedge \tag{3.2}$$

$$location(robot, loc(x^r, y^r), s) \wedge$$

$$peakOf(body, depth', size', lbd') \in profile \wedge limitBoundary(l) \wedge$$

$$size(peakOf(body, depth'', size'', lbd''), size'', loc(x^r, y^r), s) \wedge size' > size'' \wedge$$

$$depth' < l \wedge depth'' < l\} \vee$$

$$\{\exists(t, x^{old}, y^{old}, t, x^b, y^b, depth'', size'', lbd'', r, number, l) \tag{3.3}$$

$$a = endMove(robot, loc(x^{old}, y^{old}), loc(x^r, y^r), t) \wedge$$

$$radius(body, r) \wedge location(body, loc(x^b, y^b), s) \wedge limitBoundary(l) \wedge$$

$$euclDist(loc(x^b, y^b), loc(x^r, y^r), r, depth') \wedge number = r/(depth' + r) \wedge$$

$$size' = 2 * asin(number) * (180/\pi) \wedge$$

$$size(peakOf(body, depth'', size'', lbd''), size'', loc(x^{old}, y^{old}), s) \wedge$$

$$size' > size'' \wedge depth'' < l \wedge depth' < l\} \vee$$

$$\{\exists(x^1, y^1, x^2, y^2, r, l, depth'', size'', lbd'', t, number) \tag{3.4}$$

$$a = endMove(body, loc(x^1, y^1), loc(x^2, y^2), t) \wedge location(robot, loc(x^r, y^r), s) \wedge$$

$$radius(body, r) \wedge location(body, loc(x^1, y^1), s) \wedge limitBoundary(l) \wedge$$

$$euclDist(loc(x^2, y^2), loc(x^r, y^r), r, depth') \wedge number = r/(depth' + r) \wedge$$

$$size' = 2 * asin(number) * (180/\pi) \wedge$$

$$size(peakOf(body, depth'', size'', lbd''), size'', loc(x^r, y^r), s) \wedge$$

$$size' > size'' \wedge depth'' < l \wedge depth' < l\} \vee$$

$$extending(peakOf(body, depth', size', lbd'), loc(x^r, y^r), s) \wedge \tag{3.5}$$
$$\exists(x^b, y^b, l) location(body, loc(x^b, y^b), s) \wedge location(robot, loc(x^r, y^r), s) \wedge$$
$$limitBoundary(l) \wedge depth' < l \wedge$$
$$(\neg\exists(profile, t, size'', depth'', lbd'', x^{new}, y^{new}, x^1, y^1, x^2, y^2, number, r)\{$$
$$(a = sense(profile, loc(x^r, y^r), t) \wedge \tag{3.6}$$
$$peakOf(body, depth'', size'', lbd'') \in profile \wedge$$
$$(size'' < size' \vee depth'' >= l)$$
$$) \vee$$
$$(a = endMove(robot, loc(x^r, y^r), loc(x^{new}, y^{new}), t) \wedge \tag{3.7}$$
$$euclDist(loc(x^b, y^b), loc(x^{new}, y^{new}), r, depth'') \wedge$$
$$number = r/(depth'' + r) \wedge$$
$$size'' = 2 * asin(number) * 180/\pi \wedge$$
$$(size'' < size' \vee depth'' >= l)$$
$$) \vee$$
$$(a = endMove(body, loc(x^1, y^1), loc(x^2, y^2), t) \wedge \tag{3.8}$$
$$euclDist(loc(x^2, y^2), loc(x^r, y^r), r, depth'') \wedge$$
$$number = r/(depth'' + r) \wedge$$
$$size'' = 2 * asin(number) * 180/\pi \wedge$$
$$(size'' < size' \vee depth'' >= l)$$
$$)\}$$

3.1.2 Logical Shrinking

Shrinking is the transition of a peak which states that the peak, $peakOf(body, depth, size, lbd)$, representing an object, $body$, is perceived from the current robot location, $loc(x^r, y^r)$,as shrinking or decreasing in angular size in the current situation, s. There are three possible actions that may cause the size of a peak to shrink and these are explained in the logical formula below, (**??**) - (**??**).

As well, the frame axiom part of the SSA is explained in the logical formulas, (**??**) - (**??**).

The SSA for shrinking, (**??**), states that a peak is observed to be shrinking in the situation, $do(a, s)$ if there was a sensing action (**??**) that observed the angular size, $size'$, as lesser in the current situation $do(a, s)$ than the angular size, $size''$, in the previous situation s. Or the robot (**??**) or object (**??**) moved to a position such that the calculated angular size, $size'$, of the object in the current situation, $do(a, s)$ is smaller than the angular size, $size''$ in the previous situation s. In all three cases the depth, $depth'$ and $depth''$, in both situation $do(a, s)$ and s must be less than the limit boundary, l. In these cases it would represent a difference between shrinking and vanishing, (**??**). If an object were to move outside the limit boundary, even though its peak may be shrinking in angular size, its depth from the observer gives it the perceived effect of vanishing rather than shrinking.

The frame axiom for shrinking follows very close to the frame axiom for extending. The only major difference is the opposite relationship between current and past angular sizes. The formula (**??**) is the frame axiom for shrinking. The detailed explanation of the frame axiom for shrinking is identical to extending, except for the opposite relationship between current and past angular sizes.

The Figure, **??**, is a graphical representation of how an object in motion may cause it associated depth peak to shrink. The object, b, moves farther from the observer, v, and because of this motion the depth peak increases in depth and decreases in size. In **??**a) the object b is close to the right hand side of the observers field of view at a distance u. In **??**b) the corresponding depth peak graph is shown. The depth u, left boundary distance e and angular size $f - e$ corresponds to the peak q created by object b in **??**a). In **??**c) the object b has moved farther from the observer v and as can be seen in **??**d) its associated depth peak has decreased in angular size and increased in depth and left boundary distance. A decrease in angular size is indicative of an object shrinking.

$$shrinking(peakOf(body, depth', size', lbd'), loc(x^r, y^r), do(a, s)) \equiv \tag{3.9}$$

$$\{\exists(t, profile, depth'', size'', lbd'', l)a = sense(profile, loc(x^r, y^r), t) \wedge \tag{3.10}$$

$$location(robot, loc(x^r, y^r), s) \wedge$$

$$peakOf(body, depth', size', lbd') \in profile \wedge limitBoundary(l) \wedge$$

$$size(peakOf(body, depth'', size'', lbd''), size'', loc(x^r, y^r), s) \wedge size' < size'' \wedge$$

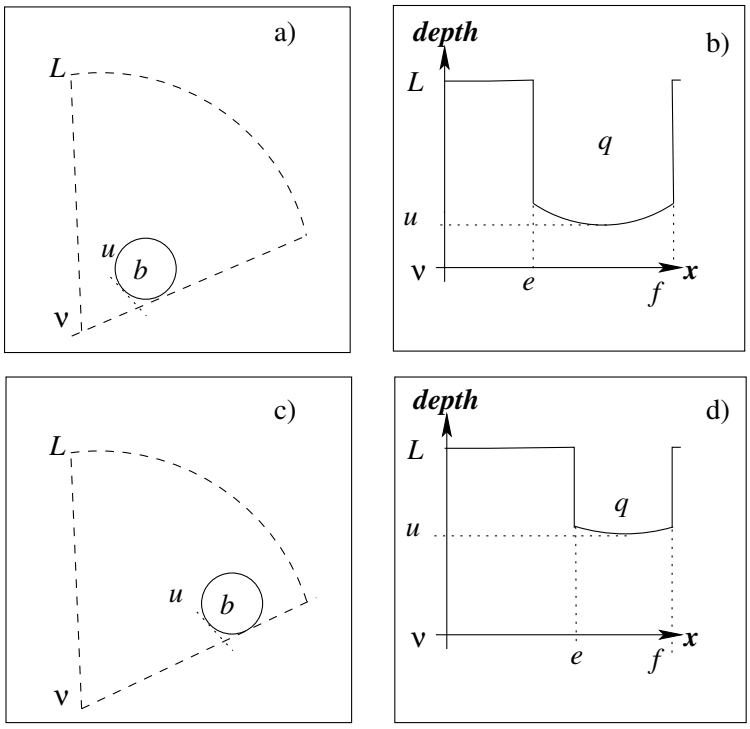

Figure 3.2: Figure of an Object's depth peak shrinking

$depth' < l \wedge depth'' < l\}\vee$

$$\{\exists(t, x^{old}, y^{old}, t, x^b, y^b, depth'', size'', lbd'', r, number, l) \tag{3.11}$$

$a = endMove(robot, loc(x^{old}, y^{old}), loc(x^r, y^r), t)\wedge$

$radius(body, r) \wedge location(body, loc(x^b, y^b), s) \wedge limitBoundary(l)\wedge$

$euclDist(loc(x^b, y^b), loc(x^r, y^r), r, depth') \wedge number = r/(depth' + r)\wedge$

$size' = 2 * asin(number) * (180/\pi)\wedge$

$size(peakOf(body, depth'', size'', lbd''), size'', loc(x^{old}, y^{old}), s)\wedge$

$size' < size'' \wedge depth'' < l \wedge depth' < l\}\vee$

$$\{\exists(x^1, y^1, x^2, y^2, r, l, depth'', size'', lbd'', t, number) \tag{3.12}$$

$a = endMove(body, loc(x^1, y^1), loc(x^2, y^2), t) \wedge location(robot, loc(x^r, y^r), s)\wedge$

$radius(body, r) \wedge location(body, loc(x^1, y^1), s) \wedge limitBoundary(l) \wedge$

$euclDist(loc(x^2, y^2), loc(x^r, y^r), r, depth') \wedge number = r/(depth' + r) \wedge$

$size' = 2 * asin(number) * (180/\pi) \wedge$

$size(peakOf(body, depth'', size'', lbd''), size'', loc(x^r, y^r), s) \wedge$

$size' < size'' \wedge depth'' < l \wedge depth' < l\} \vee$

$$shrinking(peakOf(body, depth', size', lbd'), loc(x^r, y^r), s) \wedge \tag{3.13}$$

$\exists(x^b, y^b, l) location(body, loc(x^b, y^b), s) \wedge location(robot, loc(x^r, y^r), s) \wedge$

$limitBoundary(l) \wedge depth' < l \wedge$

$(\neg\exists(profile, t, size'', depth'', lbd'', x^{new}, y^{new}, x^1, y^1, x^2, y^2, number, r)\{$

$$(a = sense(profile, loc(x^r, y^r), t) \wedge \tag{3.14}$$

$peakOf(body, depth'', size'', lbd'') \in profile \wedge$

$(size'' > size' \vee depth'' >= l)$

$) \vee$

$$(a = endMove(robot, loc(x^r, y^r), loc(x^{new}, y^{new}), t) \wedge \tag{3.15}$$

$euclDist(loc(x^b, y^b), loc(x^{new}, y^{new}), r, depth'') \wedge$

$number = r/(depth'' + r) \wedge$

$size'' = 2 * asin(number) * 180/\pi \wedge$

$(size'' > size' \vee depth'' >= l)$

$) \vee$

$$(a = endMove(body, loc(x^1, y^1), loc(x^2, y^2), t) \wedge \tag{3.16}$$

$euclDist(loc(x^2, y^2), loc(x^r, y^r), r, depth'') \wedge$

$number = r/(depth'' + r) \wedge$

$size'' = 2 * asin(number) * 180/\pi \wedge$

$(size'' > size' \vee depth'' >= l)$

$)\}$

3.1.3 Logical One Peak Static

One peak static is the transition of a single peak which states that the peak, $peakOf(body, depth, size, lbd)$, representing an object, $body$, is perceived from the current robot location, $loc(x^r, y^r)$, as remaining both the same angular size and depth in situation s when compared to the previous situation. There are three possible actions that may cause the angular size and depth of a peak to remain static and these are explained in the logical formula below, (**??**) -(**??**). As well, the frame axiom part of the SSA is expressed in the logical formula, (**??**) -(**??**).

The SSA for one peak static, (**??**), states that a peak is observed to be static in the situation, $do(a, s)$, if there was a sensing action (**??**) that observed the angular size, $size$ and depth, $depth$, as the same in the previous situation, s. Also the fluent holds if an action occurred such that the robot (**??**) or object (**??**) moved to a position such that the calculated angular size, $size$ and $depth$ of the object in the current situation, $do(a, s)$ is the same as the calculated angular size and depth in the previous situation, s. In all cases the depth must be within the limit boundary, l, as otherwise the object would not be visible to the observer.

The frame axiom for one peak static differs in that any action that causes a change in depth or size results in the peak no longer remaining static. The logical formula below, (**??**) is the frame axiom for one peak static. The first line provides a truth value for the previous situation, s. If true, the subformulas (**??**) - (**??**) check to see if the action a in the current situation, $do(a, s)$ has made the fluent for one peak static false. Subformula (**??**) states that if there was a sensing action, the the perceived peaks angular size, $size''$, and depth, $depth''$, have changed from the angular size and depth in the previous situation, s, then the peak is no longer static. SubFormulas (**??**) and (**??**) state that if the robot or the object moved to a new position, then if the computed angular size, $size''$, and depth, $depth''$, have changed from the angular size and depth in the previous situation, s, then the peak is no longer static. These conditions on angular size and depth all state the same conclusion, that if the objects depth or angular size change then the peak cannot be remaining static. Unlike the previous frame axioms for extending and shrinking, there is no need to check if the depth is within the limit boundary, l. As the truth value obtained from the first line of the right hand side of formula (**??**) will only be true if one peak static was true in the previous situation, s. In order for it to be static in the previous situation the depth of the peak must be within the limit boundary as was

stated in the formula (**??**). These conditions on depth and angular size all state that if an object is not changing then it must be remaining static. This frame axiom for one peak static covers all the cases where an action would make one peak static false.

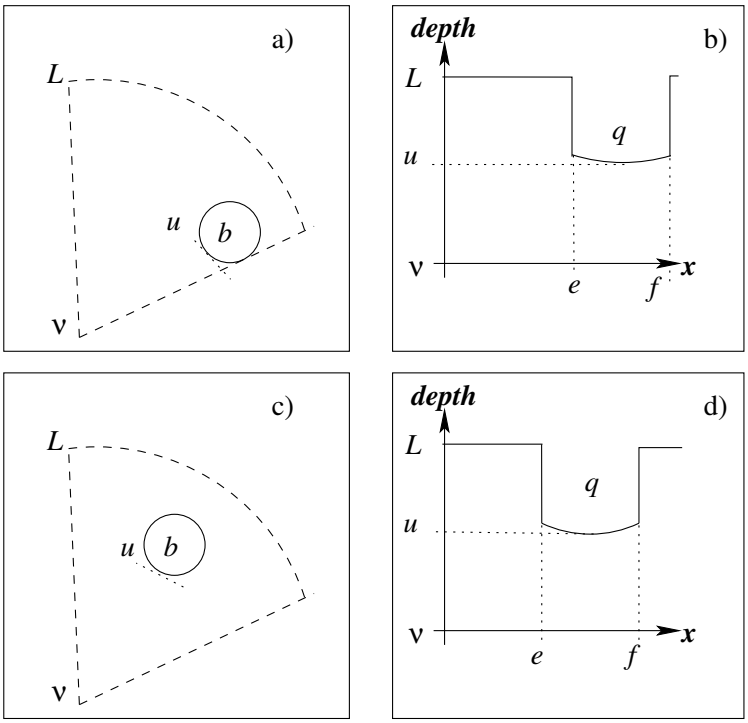

Figure 3.3: Figure of an object's depth peak remaining static

Figure **??** is a graphical representation of an object that is in motion while its associated depth peak remains static. The object, b moves closer the the left boundary of the observers, v, field of view and despite the motion, the depth and angular size of the peak **??**b), q, remains the same. As can be seen in **??**b) and **??**d), the depth, u, and angular size, $f - e$, remain the same, even though the left boundary distance, e has changed. Also, as is evident in **??**a) and **??**c) the object has clearly moved, but the distance from the observer and relative angular size has remained the same. These unchanging attributes are indicative of an objects' peak remaining static.

$$one\_peak\_static(peakOf(body, depth, size, lbd), loc(x^r, y^r), do(a, s)) \equiv \tag{3.17}$$

$$\{\exists(profile, l, t, lbd'')\, a = sense(profile, loc(x^r, y^r), t) \wedge \tag{3.18}$$

$$peakOf(body, depth, size, lbd) \in profile \wedge$$

$location(robot, loc(x^r, y^r), s) \wedge limitBoundary(l) \wedge depth < l \wedge$

$size(peakOf(body, depth, size, lbd''), size, loc(x^r, y^r), s)\} \vee$

$\{\exists(x^{old}, y^{old}, l, x^b, y^b, r, t, number, lbd'')$ (3.19)

$a = endMove(robot, loc(x^{old}, y^{old}), loc(x^r, y^r), t) \wedge$

$radius(body, r) \wedge location(body, loc(x^b, y^b), s) \wedge$

$location(robot, loc(x^r, y^r), s) \wedge limitBoundary(l) \wedge$

$euclDist(loc(x^b, y^b), loc(x^r, y^r), r, depth) \wedge$

$number = r/(depth + r) \wedge size = 2 * asin(number) * 180/\pi \wedge depth < l \wedge$

$size(peakOf(body, depth, size, lbd''), size, loc(x^{old}, y^{old}), s)\} \vee$

$\{\exists(x^1, y^1, x^2, y^2, l, t, r, lbd'')$ (3.20)

$a = endMove(body, loc(x^1, y^1), loc(x^2, y^2), t) \wedge$

$location(robot, loc(x^r, y^r), s) \wedge$

$radius(body, r) \wedge location(body, loc(x^1, y^1), s) \wedge$

$limitBoundary(l) \wedge$

$euclDist(loc(x^2, y^2), loc(x^r, y^r), r, depth) \wedge$

$number = r/(depth + r) \wedge$

$size = 2 * asin(number) * 180/\pi \wedge depth < l \wedge$

$size(peakOf(body, depth, size, lbd''), size, loc(x^r, y^r), s)\} \vee$

$one\_peak\_static(peakOf(body, depth', size', lbd'), loc(x^r, y^r), s) \wedge$ (3.21)

$\exists(x^b, y^b, l, r) \wedge$

$location(robot, loc(x^r, y^r), s) \wedge location(body, loc(x^b, y^b), s) \wedge$

$limitBoundary(l) \wedge radius(body, r) \wedge$

$(\neg\exists(profile, t, size'', depth'', lbd'', x^{new}, y^{new}, x^2, y^2, number, r)\{$ (3.22)

$$
\begin{aligned}
&(a = sense(profile, loc(x^r, y^r), t) \wedge \\
&peakOf(body, depth'', size'', lbd'') \in profile \wedge \\
&depth'' \neq depth' \wedge size' \neq size'' \\
&) \vee \\
&(a = endMove(robot, loc(x^r, y^r), loc(x^{new}, y^{new}), t) \wedge \qquad (3.23) \\
&euclDist(loc(x^{new}, y^{new}), loc(x^b, y^b), r, depth'') \wedge \\
&number = r/(depth'' + r) \wedge \\
&size'' = 2 * asin(number) * 180/\pi \wedge \\
&depth' \neq depth'' \wedge size' \neq size'') \\
&) \vee \\
&(a = endMove(body, loc(x^b, y^b), loc(x^2, y^2), t) \wedge \qquad (3.24) \\
&euclDist(loc(x^2, y^2), loc(x^r, y^r), r, depth'') \wedge \\
&number = r/(depth'' + r) \wedge \\
&size'' = 2 * asin(number) * 180/\pi \wedge \\
&depth' \neq depth'' \wedge size' \neq size'') \\
&)\}
\end{aligned}
$$

3.1.4 Logical Appearing

Appearing is the transition of a peak which states that the peak, $peakOf(body, depth, size, lbd)$, representing an object, $body$, is perceived from the current robot location, $loc(x^r, y^r)$ as appearing , i.e., having a change in depth such that it enters into the observers field of view in the current situation $do(\alpha, s)$. There are three possible actions that may cause the size of a peak to appear and these are explained below in the logical formulas (**??**) - (**??**).

The SSA for appearing, (**??**) states that an object is observed to be appearing in the situation, $do(a, s)$ if there was a sensing action (**??**) that observed the depth, $depth'$, as less than the limit boundary, l, in the current situation, $do(a, s)$, and that the depth, $depth''$ in the previous situation, s, is greater or equal to the limit boundary. Otherwise the fluent $appearing$ still holds if the robot

(**??**) or object (**??**) moved to a new position such that the calculated depth, $depth'$, of the object in the current situation, $do(a, s)$, is less than the limit boundary, l, and the depth, $depth''$, in the previous situation, s, is greater or equal to the limit boundary. In all three cases the angular size, $size'$ in the current situation is expected to increase compared to the angular size, $size''$, from the previous situation. Appearing is different from extending in that although a peak is increasing in angular size, it is doing this while also crossing the limit boundary and becoming visible by the observer.

Appearing does not have a frame axiom, because of the special nature of this axiom. The act of appearing is an instantaneous action, an object cannot appear more than once without vanishing first. The fact that in order for an object to be appearing it must change from $depth >= l$ to $depth < l$, leads to the fact that an object cannot be appearing again until some action changes the depth of the object to be, $depth >= l$. For this reason, once an object has appeared, it cannot reappear until it has vanished first. Therefore, there is no reason to have a frame axiom for appearing.

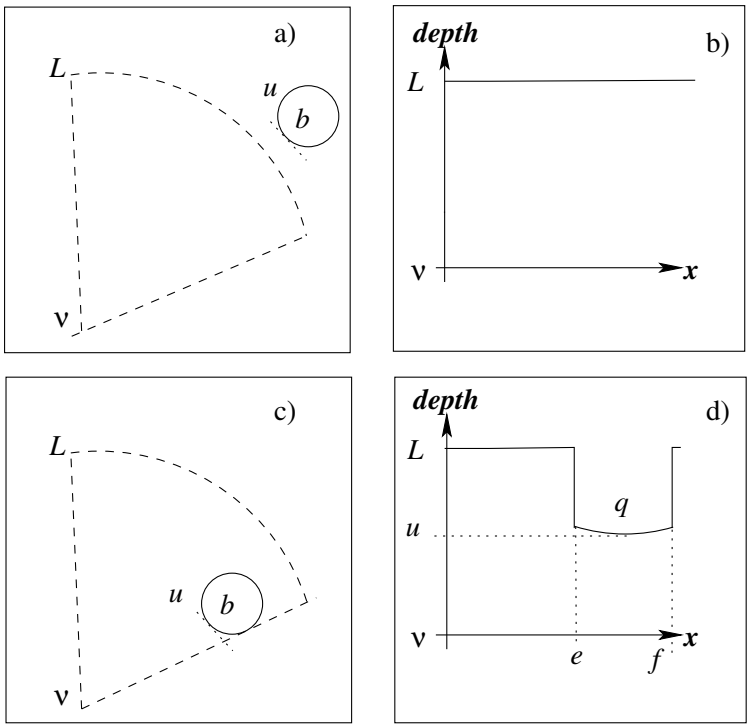

Figure 3.4: Figure of an Object's depth peak appearing

The Figure, **??**, is a graphical representation of how an object in motion may cause its associated depth peak to appear. Initially the object, b is beyond the limit boundary, L, of the observer, v. It then moves within the limit boundary, decreasing in depth, increasing in angular size and is perceived as appearing. As can be seen in **??**a) the object is outside the limit boundary, and therefore does not have an associated depth peak on the graph visible in **??**b). In **??**c) the object has moved within the limit boundary and its associated depth peak has appeared in **??**d). A object crossing the limit boundary toward the observer is indicative of an object appearing.

$$appearing(peakOf(body, depth', size', lbd'), loc(x^r, y^r), do(a, s)) \equiv \tag{3.25}$$

$$\{\exists(t, profile, depth'', size'', lbd'', l)a = sense(profile, loc(x^r, y^r), t) \wedge \tag{3.26}$$
$$location(robot, loc(x^r, y^r), s) \wedge$$
$$peakOf(body, depth', size', lbd') \in profile \wedge limitBoundary(l) \wedge$$
$$size(peakOf(body, depth'', size'', lbd''), size'', loc(x^r, y^r), s) \wedge size'' < size' \wedge$$
$$depth'' >= l \wedge depth' < l\} \vee$$

$$\{\exists(t, x^{old}, y^{old}, t, x^b, y^b, depth'', size'', lbd'', r, number, l) \tag{3.27}$$
$$a = endMove(robot, loc(x^{old}, y^{old}), loc(x^r, y^r), t) \wedge$$
$$radius(body, r) \wedge location(body, loc(x^b, y^b), s) \wedge limitBoundary(l) \wedge$$
$$euclDist(loc(x^b, y^b), loc(x^r, y^r), r, depth') \wedge number = r/(depth' + r) \wedge$$
$$size' = 2 * asin(number) * (180/\pi) \wedge$$
$$size(peakOf(body, depth'', size'', lbd''), size'', loc(x^{old}, y^{old}), s) \wedge$$
$$size'' < size' \wedge depth'' >= l \wedge depth' < l\} \vee$$

$$\{\exists(x^1, y^1, x^2, y^2, r, l, depth'', size'', lbd'', t, number) \tag{3.28}$$
$$a = endMove(body, loc(x^1, y^1), loc(x^2, y^2), t) \wedge location(robot, loc(x^r, y^r), s) \wedge$$
$$radius(body, r) \wedge location(body, loc(x^1, y^1), s) \wedge limitBoundary(l) \wedge$$
$$euclDist(loc(x^2, y^2), loc(x^r, y^r), r, depth') \wedge number = r/(depth' + r) \wedge$$

$size' = 2 * asin(number) * (180/\pi) \wedge$

$size(peakOf(body, depth'', size'', lbd''), size'', loc(x^r, y^r), s) \wedge$

$size'' < size' \wedge depth'' >= l \wedge depth' < l\}$

3.1.5 Logical Vanishing

Vanishing is the transition of a peak which states that the peak, $peakOf(body, depth, size, lbd)$ representing an object, $body$, is perceived from the robot location, $loc(x^r, y^r)$ as vanishing or having a change in depth such that it exits the observers field of view in the current situation s. Vanishing is the opposite of appearing. There are three possible actions that may cause the size of a peak to vanish and these are explained in the logical formula below, (**??**) - (**??**). The axioms for vanishing are almost identical to appearing, except that the body moves outside the limit boundary. The detailed explanation is the same as appearing, except with the opposite change in depth. Also, like appearing, there is no reason to have a frame axiom for vanishing because of the instantaneous nature of the axiom.

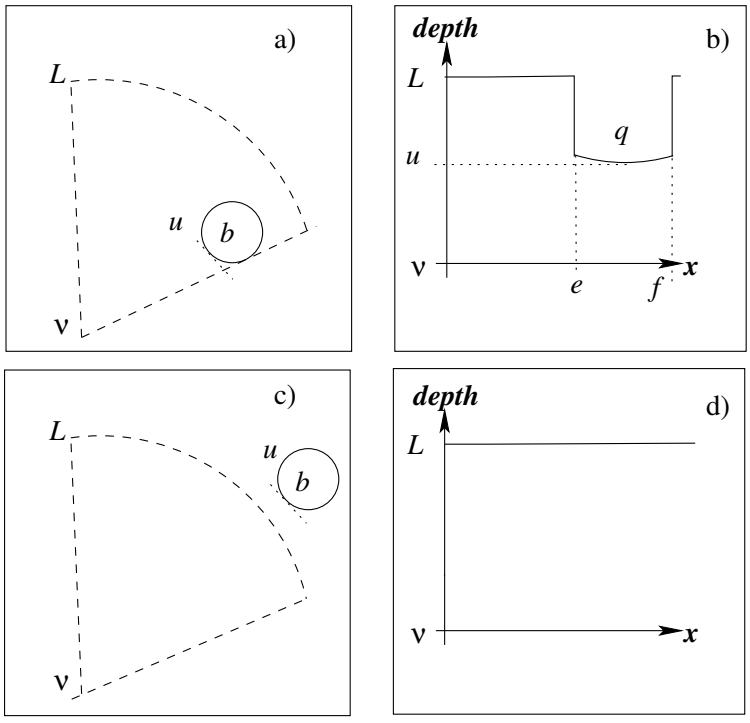

Figure 3.5: Figure of an Object's depth peak vanishing

The Figure, **??**, is a graphical representation of how an object in motion may cause its associated depth peak to vanish. Initially the object, b, is within the limit boundary, L, of the observer, v. It then moves outside the limit boundary, increasing in depth, decreasing in angular size and is perceived as vanishing. As can be seen in **??**a) the object is within the limit boundary, and its associated depth peak can be seen in **??**b). In **??**c) the object has moved outside the limit boundary and its associated depth peak has vanished in **??**d). A object crossing the limit boundary away from the observer is indicative of an object vanishing.

$$vanishing(peakOf(body, depth', size', lbd'), loc(x^r, y^r), do(a, s)) \equiv \tag{3.29}$$

$$\{\exists(t, profile, depth'', size'', lbd'', l)a = sense(profile, loc(x^r, y^r), t)\wedge \tag{3.30}$$
$$location(robot, loc(x^r, y^r), s)\wedge$$
$$peakOf(body, depth', size', lbd') \in profile \wedge limitBoundary(l)\wedge$$
$$size(peakOf(body, depth'', size'', lbd''), size'', loc(x^r, y^r), s) \wedge size' < size''\wedge$$
$$depth' >= l \wedge depth'' < l\}\vee$$

$$\{\exists(t, x^{old}, y^{old}, t, x^b, y^b, depth'', size'', lbd'', r, number, l) \tag{3.31}$$
$$a = endMove(robot, loc(x^{old}, y^{old}), loc(x^r, y^r), t)\wedge$$
$$radius(body, r) \wedge location(body, loc(x^b, y^b), s) \wedge limitBoundary(l)\wedge$$
$$euclDist(loc(x^b, y^b), loc(x^r, y^r), r, depth') \wedge number = r/(depth' + r)\wedge$$
$$size' = 2 * asin(number) * (180/\pi)\wedge$$
$$size(peakOf(body, depth'', size'', lbd''), size'', loc(x^{old}, y^{old}), s)\wedge$$
$$size' < size'' \wedge depth' >= l \wedge depth'' < l\}\vee$$

$$\{\exists(x^1, y^1, x^2, y^2, r, l, depth'', size'', lbd'', t, number) \tag{3.32}$$
$$a = endMove(body, loc(x^1, y^1), loc(x^2, y^2), t) \wedge location(robot, loc(x^r, y^r), s)\wedge$$
$$radius(body, r) \wedge location(body, loc(x^1, y^1), s) \wedge limitBoundary(l)\wedge$$
$$euclDist(loc(x^2, y^2), loc(x^r, y^r), R, depth') \wedge number = r/(depth' + r)\wedge$$

$size' = 2 * asin(number) * (180/\pi) \wedge$

$size(peakOf(body, depth'', size'', lbd''), size'', loc(x^r, y^r), s) \wedge$

$size' < size'' \wedge depth' >= l \wedge depth'' < l\}$

3.1.6 Logical FrontOf

The directional axioms, such as $frontOf$, are axioms that are used to determine where an object is in relation to the observer on a two-dimensional plane. As can be seen in the Figure, **??**, the domain is broken up into 8 equal regions that are based on the direction the observer is facing. As the observer moves the regions are relative to its front facing direction. We will discuss the SSA, $frontOf$, for an object being in front of the observer. The other direction SSA's follow very closely to $frontOf$ and can be seen in their entirety in the appendices.

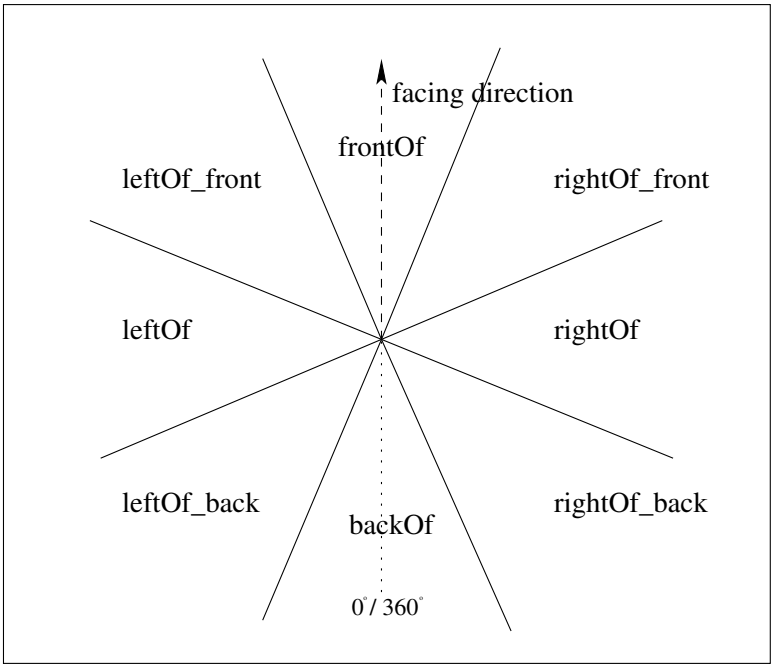

Figure 3.6: Figure of the relative directional axiom regions

The SSA for $frontOf$, (**??**), states that an object is in front of the observer in the current situation iff the center of the object's angular distance from the observers left boundary is within the range $157.5°$ to $202.5°$, where $157.5° = 180° - 45°/2$ and $202.5° = 180° + 45°/2$. As the

subformula, (**??**) states, if a sensing action occurred and the observed left boundary distance, lbd, is within the appropriate range then the object, $body$, is in front of the observer. Otherwise, if the robot, (**??**), or object, (**??**), moved to a new position such that the calculated left boundary distance, lbd, is with the appropriate range then the object is in front of the observer. Or, if the observer rotated,(**??**) and the new calculated left boundary distance, lbd, is within the appropriate range then the object is in front of the observer. When rotating, the previous left boundary distance, ldb', can be used to calculate the current left boundary distance, lbd, as the action of rotating does not affect the positions of any objects. It only affects the direction that the robot is facing, and therefore only the left boundary distances are affected in the current situation by the amount of rotation that the observer has completed. It is important to note than an object can only be classified as being in one region at a time.

The frame axiom for $frontOf$, (**??**), will return the truth value of whether an action that has occurred has had the effect of moving the object outside the $frontOf$ region. The first line checks to see if the object was in front of the observer in the previous situation, s. If the object is in front, and a sensing action (**??**) has occurred, the frame axiom checks to see if the objects left boundary distance, lbd' in the current situation, $do(a, s)$ is outside the region $frontOf$. Otherwise, if the robot(**??**) or object (**??**) has moved to a new position then the calculated left boundary distance in the current situation, $do(a, s)$, is not outside the region, $frontOf$. Finally, that if the observer rotated (**??**), then the calculated left boundary distance in the current situation, $do(a, s)$, is not outside the region, $frontOf$. This frame axiom states that if any of these actions occured, then none of them has caused the object to move outside the $frontOf$ region, then the object must still be in the $frontOf$ region.

$$frontOf(peakOf(body, depth, size, lbd), loc(x^r, y^r), do(a, s)) \equiv \tag{3.33}$$

$$\{\exists(profile, t, regsize)a = sense(profile, loc(x^r, y^r), t) \wedge \tag{3.34}$$

$$peakOf(body, depth, size, lbd) \in profile) \wedge$$

$$regionsize(regsize) \wedge lbd >= 180 - regsize/2 \wedge lbd =< 180 + regsize/2\} \vee$$

$$\{\exists(x^{old}, y^{old}, t, x, y, \theta, alpha, leftborder, bodyangle, regsize) \tag{3.35}$$

$a = endMove(robot, loc(x^{old}, y^{old}), loc(x^r, y^r), t) \wedge$

$location(body, loc(x, y), s) \wedge location(robot, loc(x^{old}, y^{old}), s) \wedge$

$facing(robot, \theta, s) \wedge fieldView(alpha) \wedge leftborder = \theta + alpha \wedge$

$calcAngle(bodyangle, x, y, x^r, y^r) \wedge lbd = leftborder - bodyangle \wedge$

$regionsize(regsize) \wedge lbd >= 180 - regsize/2 \wedge lbd =< 180 + regsize/2\} \vee$

$\{\exists(x', y', x, y, t, \theta, alpha, leftborder, bodyangle, regsize)$ (3.36)

$a = endMove(body, loc(x', y'), loc(x, y), t) \wedge$

$location(body, loc(x', y'), s) \wedge location(robot, loc(x^r, y^r), s) \wedge$

$facing(robot, \theta, s) \wedge fieldView(alpha) \wedge leftborder = \theta + alpha \wedge$

$calcAngle(bodyangle, x, y, x^r, y^r) \wedge lbd = leftborder - bodyangle \wedge$

$regionsize(regsize) \wedge lbd >= 180 - regsize/2 \wedge lbd =< 180 + regsize/2\} \vee$

$\{\exists(x, y, t, \theta, alpha, \omega, regsize, depth', size', lbd')$ (3.37)

$a = endPan(\omega, t) \wedge$

$location(body, loc(x, y), s) \wedge location(robot, loc(x^r, y^r), s) \wedge$

$facing(robot, \theta, s) \wedge fieldView(alpha) \wedge$

$lbd(peakOf(body, depth', size', lbd'), lbd', loc(x^r, y^r), s) \wedge lbd = lbd' + \omega \wedge$

$regionsize(regsize) \wedge lbd >= 180 - regsize/2 \wedge lbd =< 180 + regsize/2\} \vee$

$frontOf(peakOf(body, depth, size, lbd), loc(x^r, y^r), s) \wedge$ (3.38)

$\exists(x, y, alpha, regsize, \theta, leftborder)$

$location(body, loc(x, y), s) \wedge location(robot, loc(x^r, y^r), s) \wedge$

$regionsize(regsize) \wedge facing(robot, \theta, s) \wedge fieldView(alpha) \wedge$

$leftborder = \theta + alpha \wedge$

$\{\neg\exists(profile, t, x^{new}, y^{new}, x', y', bodyangle, depth', size', lbd')$ (3.39)

$$(a = sense(profile, loc(x^r, y^r), t) \wedge$$
$$peakOf(body, depth', size', lbd') \in profile) \wedge$$
$$lbd' =< 180 - regsize/2 \vee lbd' >= 180 + regsize/2) \vee$$

$$(a = endMove(robot, loc(x^r, y^r), loc(x^{new}, y^{new}), t) \wedge \tag{3.40}$$
$$calcAngle(bodyangle, x, y, x^{new}, y^{new}) \wedge lbd' = leftborder - bodyangle \wedge$$
$$lbd' =< 180 - regsize/2 \vee lbd' >= 180 + regsize/2) \vee$$

$$(a = endMove(body, loc(x, y), loc(x', y'), t) \wedge \tag{3.41}$$
$$calcAngle(bodyangle, x', y', x^r, y^r) \wedge lbd' = leftborder - bodyangle \wedge$$
$$lbd' =< 180 - regsize/2 \vee lbd' >= 180 + regsize/2) \vee$$

$$(a = endPan(\omega, t) \wedge lbd' = lbd + \omega \wedge \tag{3.42}$$
$$lbd' =< 180 - regsize/2 \vee lbd' >= 180 + regsize/2)\}$$

3.2 System Architecture

The generalized plan recognition system can be broken into three separate conceptual parts. The first is the situation term, which is a recorded history of all observed actions. The second is the successor state axioms which use as input the situation term to determine the truth value of each fluent. Finally, the truth values of all fluents are pre-processed into conjunctive normal form(CNF) logic formulas and passed to recognition automata that represent plans in the domain.

The situation term is a recorded history of observed actions in the domain. The structure of the situation term and actions are discussed in **??**. Observations in the domain are coming from the point of view of the robot. The robot has the ability to move, rotate, and sense depth peaks of objects in its environment. Although the robot is only aware of its own movement from location to location, it is able to reason, using information from depth peaks, about the location of other objects in the domain.

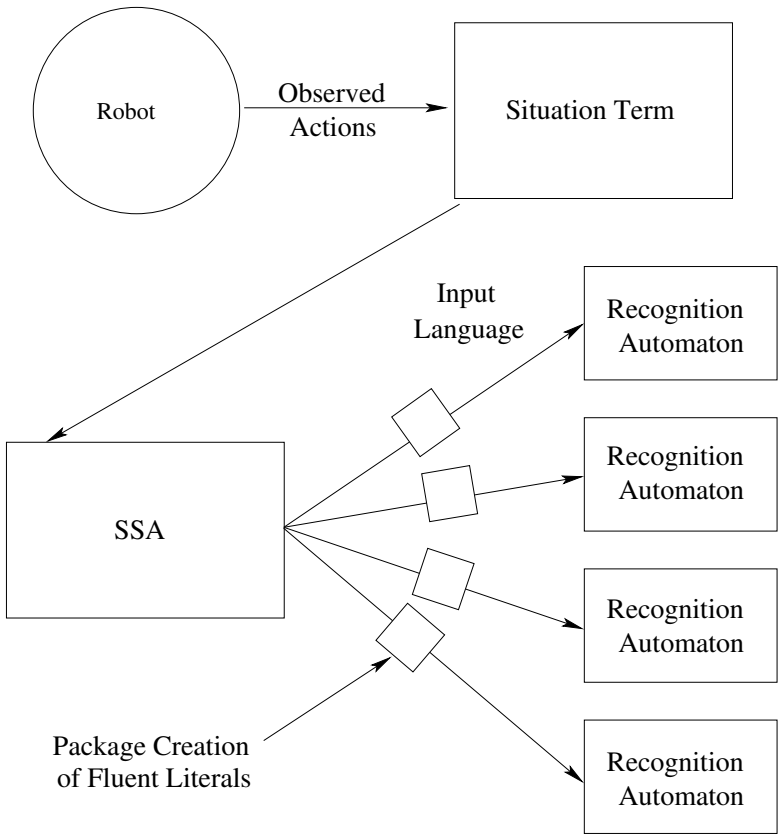

Figure 3.7: System Architecture Diagram

The situation term is used by successor state axioms to reason about relationships in the domain. The SSA have been explained in both Section **??** and the previous section. The truth values of fluents, determined by the SSA, may change over time and this information is packaged into CNF formulas to be the input language to finite state automata. Each FSA has an input language that is defined by boolean combinations of fluent values. A different recognition automaton is defined for each plan in the system. Upon execution of the system, a separate recognition automaton is given input for each object being reasoned about. It is unknown at the beginning of execution which behaviour an object may be performing. Therefore, every object is associated with a separate automaton for all plans in the system. As the system continues its execution, automaton can reach goal or fail states and will no longer need to receive input as object's behaviours are recognized. The recognition automata are formally defined as FSA in the next section.

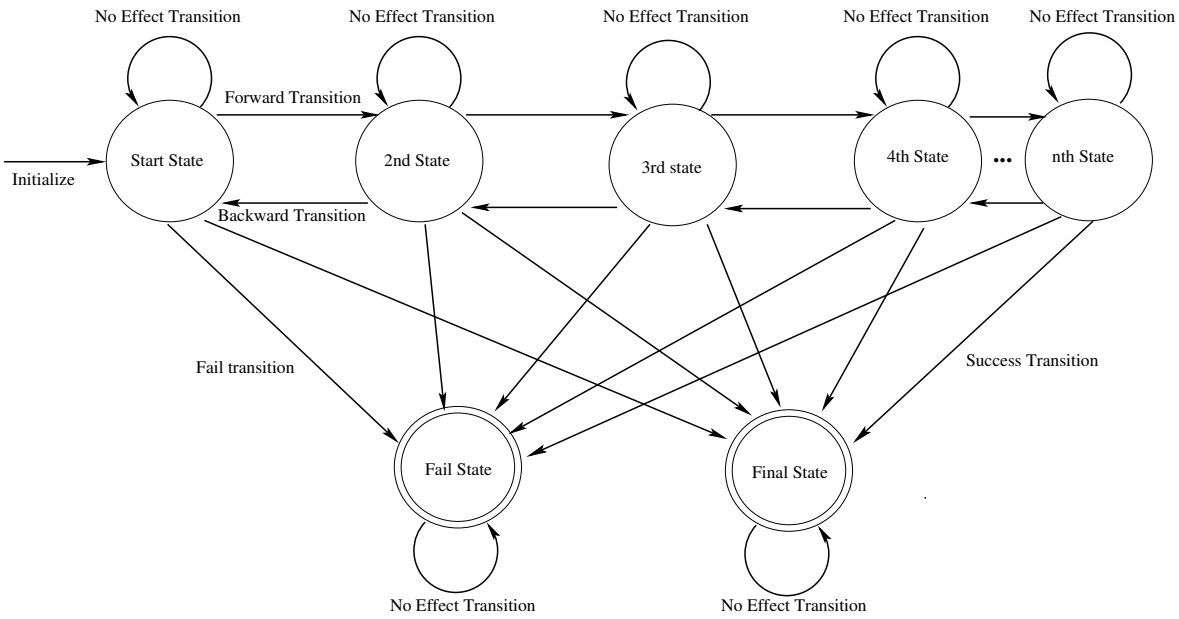

Figure 3.8: Generic Plan Recognition Procedure as FSA

3.3 Generic Structure of Plan Recognition Procedures

Each plan recognition procedure can be described as a deterministic finite state automaton(FSA). The input to the FSA are CNF formulas built from fluent truth values determined from successor state axioms described in the previous section. State changes in a FSA are determined by a transition function defined using boolean combinations of fluent truth values. Every FSA has two final states, one representing failure to recognize the plan and one representing successful plan recognition. Each plan recognition procedure is described by its own FSA with its own transition function.

The general structure of the system can be visualized as multiple FSA accepting the same fluent values to change states using the transition function of the associated FSA. FSA's are described by their input alphabet, acceptable strings, states and transition function. I will describe these different attributes in the following section. FSA are a convenient software engineering tool that we apply to structure our plan recognition programs. Also, they add clarity to the process of designing plan recognition programs.

3.3.1 Alphabets, Strings, and Language of the FSA

An alphabet is a finite, nonempty set of symbols. As per convention we will use the symbol Σ for an alphabet. Unlike many FSA which use alphabets of atomic symbols, such as binary digits, letters, or ASCII code, I will define a special alphabet that contains the minimal information needed for a state transition. The alphabet Σ contains fluents that represent information available to the robot at specific units of time in the real world. Fluents values are used to define the language accepted by each FSA.

Alphabet

Unlike traditional alphabets for FSA, the alphabet for these plan recognition procedures is made up of boolean fluent combinations represented using the following predicate symbols. Each fluent can hold the value of true or false, and we will identify the value of each fluent by prefixing the predicate symbol with either a $+$ or $-$ representing true and false respectively. The input to FSA are consistent boolean combinations of fluents representing alphabet symbols. In practice, we observed that these boolean combinations are short and they mention no more than 3 fluents. Below are the individual fluent symbols available that can be combined into short boolean combinations (at most 3 fluent symbols) for each FSA from the list below:

$$\begin{aligned} &extending(peak, loc, s), shrinking(peak, loc, s), appearing(peak, loc, s), \\ &vanishing(peak, loc, s), one\_peak\_static(peak, loc, s), frontOf(peak, loc, s), \\ &leftOf\_front(peak, loc, s), leftOf(peak, loc, s), leftOf\_back(peak, loc, s), \\ &backOf(peak, loc, s), rightOf\_back(peak, loc, s), \\ &rightOf(peak, loc, s), rightOf\_front(peak, loc, s) \end{aligned} \tag{3.43}$$

Strings and Lanugage

A string is a concatenations of alphabet symbols. In our FSA a string will be a concatenation of the boolean combinations from the alphabet defined by the FSA. For example, in situation, s, the boolean combination of fluent symbols may be $extending \wedge backOf$. In the next situation

the next boolean combination may be $static \wedge leftOf\_back$. Therefore the string would become $extending \wedge backOf; static \wedge leftOf\_back$, where ; represents concatenation of fluent symbols.

Each FSA accepts a language of boolean combinations of symbols. Due to how we define certain fluents, alphabet symbols must represent consistent fluent combinations. For example the fluents $frontOf$ and $backOf$ cannot be true at the same time for the same body. Fluents are packed into an alphabet symbol before being passed to a FSA. Therefore we define the language Σ^* as the language of our FSA, where each acceptable string in the language is a concatenation of boolean combinations of fluents that leads to an accepting state. The strings that are acceptable to each FSA, are the strings that are acceptable by the language used to define the individual FSA.

States

Before we define the transition functions, we will look at the four distinctly different states found in plan recognition procedure represented as FSA's, start states, intermediate states, fail states and success states.

Every procedure has the ability to complete, and because of this every FSA will enter its **Start State** upon initialization of the FSA. After the first symbol is read by the FSA, it will have four options, whether to move forward in the procedure, move to the fail state, move to the success state or have no effect and stay in the start state.

Every procedure has multiple steps, these in general need to be completed for the FSA to reach the final state. These multiple steps are defined by the **intermediate states**, 2^{nd}, 3^{rd}, n^{th} etc as shown in (Figure: **??**). Intermediate states allow us to know at what stage a plan recognition procedure . It has all the same transitions as the start state, but for each intermediate state there can be different conditions on the transition.

The **Fail State**, is a special state that represents when a plan recognition procedure is considered not executable and enters this fail state that has no transition out. No matter the input, the

FSA will remain in the fail state. In this state, it is considered that this FSA does not recognize the plan being executed by an object in the domain, because the object is executing some other plan.

The **Success State** represents a procedure being completed. When the success state is entered, regardless of the input it will not change states. Once the success state is entered the procedure is considered complete and other input can be ignored. This means that the FSA has recognized the plan that the object is executing. For example, this can be used to represent a body overtaking the robot. Once the body has overtaken the robot, the procedure is completed, and unless the FSA is restarted from the beginning, the same FSA cannot represent another overtaking procedure.

3.3.2 Transition Functions

Transition functions define under which conditions a FSA will move from state to state. Transitions are triggered by conditional statements formulated using fluents. Here we will show how to generally define these transition functions and their place in the transition diagram (Figure: **??**).

A **fail transition** can occur from any state. This transition is caused by a string of inputs that makes a conditional statement true which represents the plan recognition procedure as unable to complete. Any state can make a transition to the fail state.

A **no effect transition** can be considered for any input that does not affect the current state's transition function of the FSA. The current input string does not trigger any other transitions, so a no effect transition occurs and keeps the FSA in the same state.

A **success transition** is a transition that leads to the success state. Once in the success state the procedure will ignore all other input and be considered complete. Although a FSA can move from any of the intermediate states or the start state to the success state, it is recommended when writing transitions to only define a success transition from the final intermediate state.

A **forward transition** is a transition that moves the procedure partially forward. In terms of FSA, it moves up one state towards the n^{th} intermediate state. And similarly a **backward transition** is a transition that moves the procedure partially backwards.

We introduce the concept of a counter on **no effect transitions**. When considering forward and backward transitions, these transitions show that the procedure is changing. Success and fail

transitions lead to end states of a FSA, but no effect transitions have no effect on the procedure. The concept of the counter is such that if a no effect transition occurs x number of times then the procedure is not occurring, i.e. the object is engaged in a behaviour that is different from what the FSA is designed to recognize. The limit on the counter is to be defined separately for each state and is reset if the procedure makes a transition other than **no effect**. Examples of how to define a counter can be seen in later sections.

Generic Transition Functions

Transition functions should follow a general structure where based on the fluents provided we define conditional statements on what should and should not be happening in the current state for the state to change. This leads to a natural structure of having positive conditions and negative conditions defining each state's transition conditions. The logical operators used are a subset of those that are allowed in Golog test conditions ($\wedge, \vee, \neg$). We find that these operators are expressive enough to define transition functions for the domain. When defining transition functions we use a short form where we only write the boolean combination of fluent values we are interested in. For example if we are interested in if the object is extending and to the right of the robot then:

$$
\begin{aligned}
&(+extending(peak, loc, s) \wedge +rightOf(peak, loc, s)) \equiv \\
&(+extending(peak, loc, s) \wedge -shrinking(peak, loc, s) \wedge -appearing(peak, loc, s) \wedge \\
&-vanishing(peak, loc, s) \wedge -one\_peak\_static(peak, loc, s)) \wedge \\
&(-leftOf\_front(peak, loc, s) \wedge -leftOf(peak, loc, s) \wedge -leftOf\_back(peak, loc, s) \wedge \\
&-backOf(peak, loc, s) \wedge -rightOf\_back(peak, loc, s) \wedge \\
&+rightOf(peak, loc, s) \wedge -rightOf\_front(peak, loc, s))
\end{aligned}
\tag{3.44}
$$

This short form makes it easy to focus on only relevant fluent values for each transition and it also saves space (and computation time) when writing transition functions. Recall that evaluating each fluent in the current situation takes time, but evaluation of irrelevant fluents does not contribute anything to making decision which transition automata should take. If we do not want the fluent $rightof$ to be true, then there are 7 other possible directions the object could be in, and 7 possible CNF formula that could match and still make that transition happen. For these reasons we will use

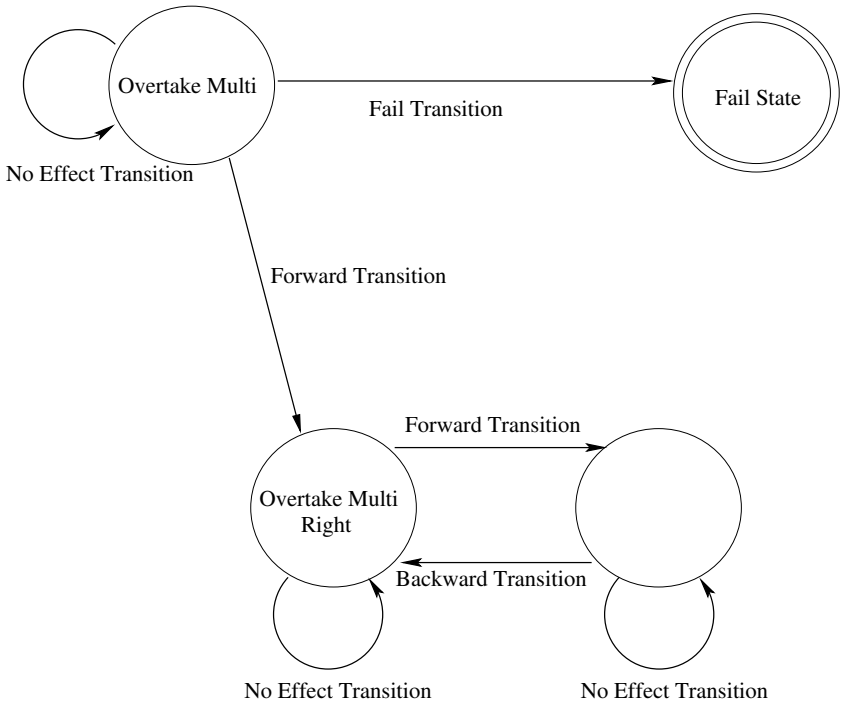

Figure 3.9: Partial Figure of Overtaking Right FSA

this short form. From this we can define any single transition function using a general structure. For example, consider the transition label, $((A \wedge B) \wedge \neg(C \wedge D))$. This statement can be read as $A \wedge B$ are the conditions that must be met for this transition to happen and $C \wedge D$ are conditions that must not occur for this transition to happen. The symbols A, B, C, D represent fluents truth values. We can use as a simple example the act of overtaking in the traffic domain. In order for an object to start overtaking on the right it must be moving faster and be moving from behind to the right of the robot. Since it is moving faster its peak will appear to be extending. A transition label for this would take the form and can be seen labeled, Forward Transition, in Figure **??**:

$$(+extending(peakof(object, depth, size, lbd), loc, s) \wedge$$
$$+ rightofback(peakof(object, depth, size, lbd), loc, s))$$

Including conditions that should not occur for a transition to happen helps to avoid false-positives. As this is only a single transition from a single state, one has to consider that there are multiple transitions from a single state. When defining multiple transitions, including conditions that can't happen, limit the transition function to only choosing one possible transition, even though the state

has many possible transitions. In the example above, the FSA could perform a no effect transition, a forward transition, or a fail transition, Figure **??**. We imagine that each transition is labelled with some alphabet symbol (boolean combination of fluents) that effects this transition. These labels are used in the transition function below. We are assuming the FSA is in its start state and are only considering the possibility of overtaking on the right for simplicity. A complete transition function for all possible transitions from the initial state in the example provided above is as follows:

$$\begin{aligned} ForwardTransition = (&+extending(peakOf(object, depth, size, lbd), loc, s) \wedge \\ &+ rightOf\_back(peakOf(object, depth, size, lbd), loc, s)) \end{aligned} \tag{3.45}$$

$$\begin{aligned} FailTransition = &+shrinking(peakOf(object, depth, size, lbd), loc, s) \\ &\wedge (+backOf(peakOf(object, depth, size, lbd), loc, s) \vee \\ &+ rightOf\_back(peakOf(object, depth, size, lbd), loc, s)) \\ &\wedge -extending(peakOf(object, depth, size, lbd), loc, s) \end{aligned}$$

$$\begin{aligned} NoEffectTransition = (&+backOf(peakOf(object, depth, size, lbd), loc, s) \wedge \\ &- shrinking(peakOf(object, depth, size, lbd), loc, s)) \end{aligned}$$

The forward transition function states that an object is moving faster than the robot and is moving to the right of the robot. This is representative of an object starting to overtake the robot. The Fail transition covers the cases that object is not moving closer to the robot, or may just be changing lanes. Finally, the no effect transition states that the object remains in place behind, and has not made a change that would represent overtaking, but has also not taken any action to represent that it may not start overtaking in the future. Similar transition functions are built for all states in each recognition automata.

3.4 Computational Geometry

In this section we define some computation geometry algorithms that were developed to calculate values of numerical fluents. Using information about location, facing direction, depth and size we are able to calculate the location of objects in the domain.

Euclidean Distance Although this distance is determined by a simple formula, we make a modification to it because objects in our domain have a definite size. The algorithm is modified to find the nearest point on the object to the observer. Here the superscript r represents the robot and superscript o represents the object in the domain.

$$euclDist = [\sqrt{(x^r - x^o)^2 - (y^r - y^o)^2}] - radius^o \tag{3.46}$$

Calculate Location From Sensing Data In the successor state axiom for location we mention that it is possible to calculate the location of the object based on the peak of the object gathered from sensing data. A peak of an object contains the distance, $depth$, to the nearest point on the object, the angular size of the object, $size$, and the angular distance from the left boundary, lbd, of the robot's field of view to the center of the object. Using this information and the robot's location we are able to calculate the location of any object in the domain using the following algorithm.

As seen in Figure **??**, the angles used are the angles that the objects make with the x-axis. The x-axis is absolute in the domain and is equivalent to the real world, east/west axis. The left boundary of the robots field of view and lbd are relative to the facing direction of the robot in the current situation. The angle of the left boundary of the field of view is measured from the positive x-axis of the domain, which is refered to as the $leftborder$. The lbd is calculated from the $leftborder$ counterclockwise to the center point of a body. The angles, θ, can be calculated from the difference between left boundary of the robots field of view, and the lbd. The following equations define these angles, θ, in terms of the robots left boundary and lbd. The angles, θ are calculated to based on what Cartesian quadrant the body lies in, this is because of the limits on $\cos$ and $\sin$. Therefore depending on the quadrant the body lies in relation to the robot, we use a different equation to calculate its location.

$$\begin{aligned}
\theta^1 &= (leftborder - lbd) \\
\theta^2 &= 180 - (leftborder - lbd) \\
\theta^3 &= (leftborder - lbd) - 180 \\
\theta^4 &= 360 - (leftborder - lbd)
\end{aligned} \tag{3.47}$$

$$x^o = x^r + \cos(\theta^1) \wedge y^o = y^r + \sin(\theta^1) \quad / * 1^{st}\, quadrant * /$$

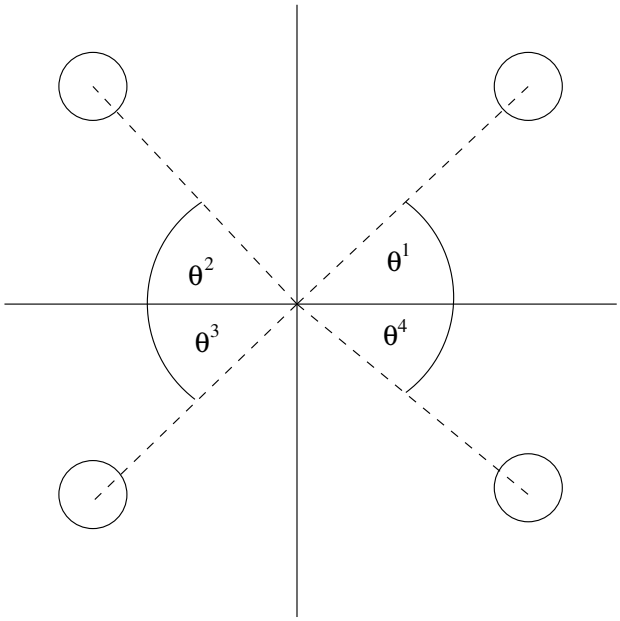

Figure 3.10: Angles Used to Calculate Location

$$x^o = x^r - \cos(\theta^2) \wedge y^o = y^r + \sin(\theta^2) \quad /*\, 2^{nd}\, quadrant\, */$$
$$x^o = x^r - \cos(\theta^3) \wedge y^o = y^r - \sin(\theta^3) \quad /*\, 3^{rd}\, quadrant\, */$$
$$x^o = x^r + \cos(\theta^4) \wedge y^o = y^r - \sin(\theta^4) \quad /*\, 4^{th}\, quadrant\, */$$

Calculating Left Boundary Distance After Movement After an object or a robot moves, it is simple to calculate distance: the euclidean distance formula can be used for this. Calculating the left boundary distance is more complex. The following equations are for these occurrences, when lbd is unknown after movement, but the location of the robot and the body are known. We follow a similar procedure as before, breaking the equations up based on the Cartesian location of the object in relation to the observer. We initially calculate the angle the object makes with the positive x-axis, then using this angle and the observers left boundary, we can calculate the difference which provides the left boundary distance of the object, lbd.

$$\begin{aligned} lbd = leftborder - \arctan(|y^o - y^r|/|x^o - x^r|) \quad & (quadrant\, 1) \\ leftborder - \arctan(|x^o - x^r|/|y^o - y^r|) + 90^\circ \quad & (quadrant\, 2) \\ leftborder - \arctan(|y^o - y^r|/|x^o - x^r|) + 180^\circ \quad & (quadrant\, 3) \end{aligned} \tag{3.48}$$

$$leftborder - \arctan(|x^o - x^r|/|y^o - y^r|) + 270^\circ \qquad (quadrant\,4)$$

These are the three common formulas used throughout the successor state axioms. There was also a modified sweep line algorithm developed, but this was explained in Section **??** and is related to calculating depth profiles, [**?**, **?**]

3.5 Complexity Analysis

In this section we will look at how well our system can scale both in terms of size of plan libraries and number of objects being reasoned about. We will look at a worst case, and an expected average case using our system. The average case is what would be expected if the design guidelines put forth are followed correctly by the system designer, in turn, limiting the number of useless computations performed.

3.5.1 Limiting Transitions Between States

We will look at the two cases for transitions between states. First, is the worst case, where there are 5 transitions from each state. Second, is the average case, where we limit transitions to the Success state. The idea behind this is that by requiring FSA to follow a specific number of steps in a certain order to reach the Success state, we will limit the number of false recognitions on plan recognition programs. We let, n, represent the number of states in the following analysis.

First we will be looking at an individual automata and the probability that it will reach an end state. Using the general structure of a recognition automata from Figure **??** we know that the number of transitions from each state in the worse case is 5. The number of states in a automata is n. The total number of transitions in an automata is therefore $5n$ and only $2n$ of these can lead to an end state (either a Success or Fail state) . The goal of designing an automata is to get to an end state, whether Fail or Success, as quickly as possible without making any false recognitions or false failures. In the worst case (worst case design of an automata) we assume that the automata is able to make it to the Fail or Success state from any intermediate state. Although we describe a worst case, to avoid false positives we assumed a plan must finish a certain number of steps from a procedure before it would complete the plan and be correctly recognized. Our worst case

scenario does not necessarily describe the least efficient in terms of reach a final state, but instead describes the worst probability of being correct upon reaching the final state. In the worst case we can determine that there is a 40% probability of reaching a final state on each transition made and a 60% chance of the automata remaining in an intermediate state. That is $2/5$ of the possible transitions will lead to a final state, assuming all transitions are equally probable.

In the average case, where an automaton has to reach the n^{th} state to be able to reach the Success state, there is a much smaller probability of reaching the Success state. To reach the Success state the automata would have to perform n forward transitions, with the probability of a forward transition being 25%. The probability of reaching the Success state is therefore $\frac{25\%}{n}$. Since there are only 4 transitions per state in the average case, the probability of reaching the fail state is 25%. The probability of a automata remaining in an intermediate state rises to 75%. Even though there is a smaller probability of reaching an end state, the probability of having a false positive/negative is greatly reduced by restricting the number of transitions from each state. It is more important to have a system that will produce more correct answers than a system that produces more errors.

3.5.2 Scalability of the System

We will now take a look at how this system scales as more automata are added with more bodies being reasoned about. We will use the probabilities above to see how many automata have not reached a final state after each transition, and how long it would take to recognize a plan on a large set of automata.

Lets assume there are x recognition automata each with n states. In the worst case 60% of the automata will remain in the intermediate states after the first symbol is received. If we look at the number that would be in a final state after the first input is received, the result is much worse. 20% of automata would have reached the Success state . This is from the fact that 1 in 5 transistions lead to the Success state in the worst case. That is to say that out of 100 different plans, 20 would be recognized as being performed by an object. We can therefore say that in the worse case we may have 20% as false positives.

In the average case, following the guidelines, it would take at least n inputs for a automata to reach the Success state, as it takes n forward transitions to reach the Success state in the average

case. Although the number of automata after each input remaining would be larger than the worse case, as there are less transitions to an end state in teh average case. The chance of a false positive happening after n inputs on all automata in the domain is $\frac{25\%}{x*n}$. This is calculated from the fact that 1 in 4(25%) transitions are forward transitions. And it takes n transitions to reach the Final state and there are x automata. We can see that the chance of a false positives is almost negligible when there is a large number of automata in the domain and a large number of states in each automata.

Although the probability of a false negative increase in the average case, if we look at the structure of transition functions this makes more sense. Since we only have 13 fluents(8 directional and 5 about peaks), and only 2 can be true about an object at any given time (1 directional and one about peaks). This means there are 11 fluents remaining, that if true, we have an undesired result. Having a greater possibility to move to a fail state, when there is a larger percentage of possible false fluents is logical. Meanwhile, looking at the worse case, having such a high possibility of success with a small possibility of true fluents is illogical. In our design choices we aim to have more automaton reach a Fail state sooner, trimming the plan library. While, at the same time, those automaton that reach a Success state are there because of a correct matching of a plan recognition program.

While the system starts off with a large number of automata, it quickly prunes the number of running automata, and if following the suggested design guidelines it is possible to avoid many false positives/negatives. Below we include some theoretical scaling results in graph form, of how the worst case and best case scale. What can be seen obviously is that even though the worse case trims running automata faster, the advantage even for a large number of automata is not nearly large enough to justify the possibility of incorrect results.

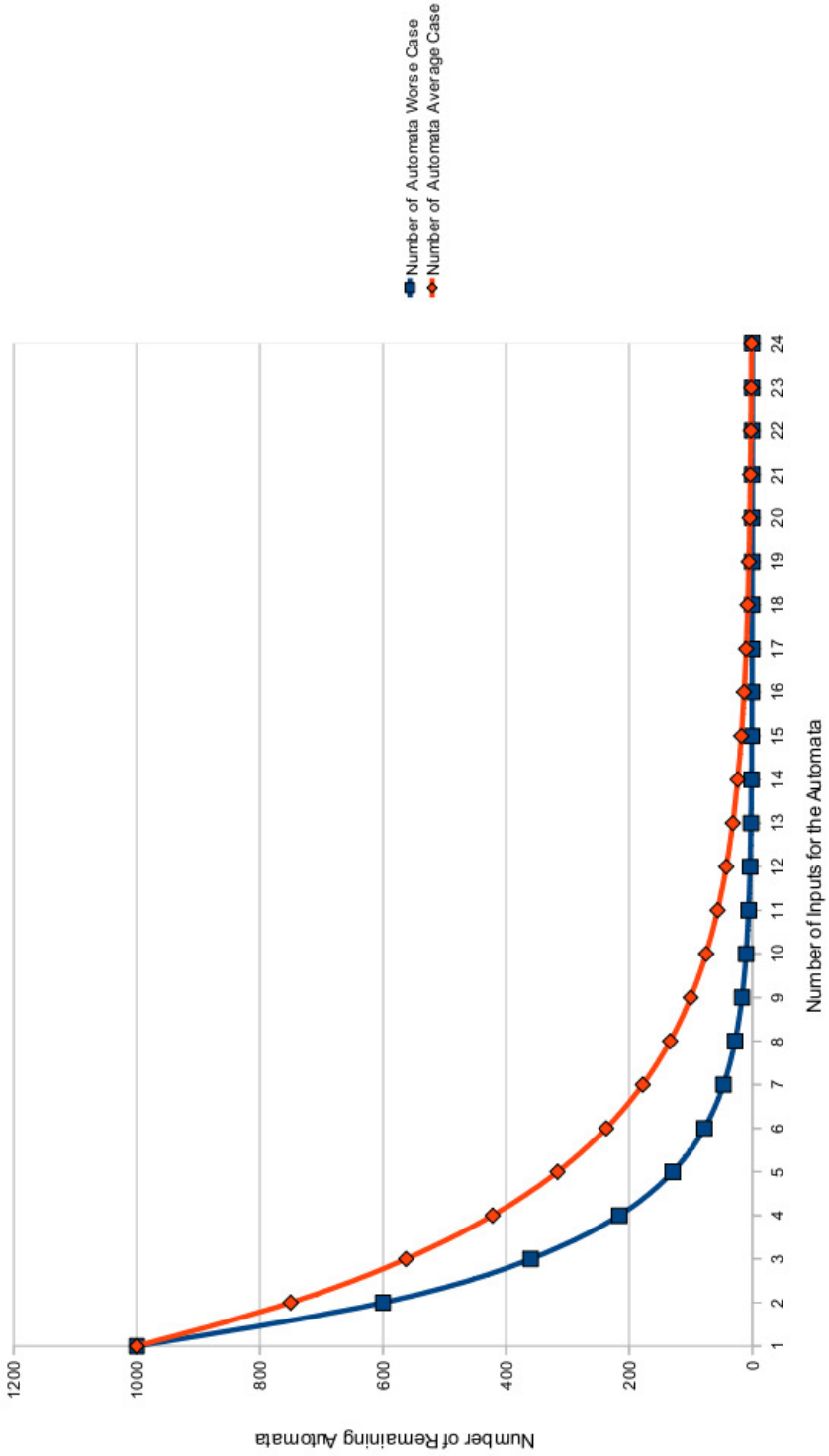

Figure 3.11: Worst and Average Case Comparison

Chapter 4

Plan Recognition in the Traffic Domain

4.1 Introduction and Motivation

Our ultimate goal is to create a system that can determine what plans agents in the domain are performing based on their relations to a user controlled observer, who is also interacting with the domain. This chapter will demonstrate that we can define relevant plan recognition procedures in a FSA format using logical relations between an observer and the agent. We will explore plan recognition specific to the traffic domain. Although other significant work [**?**, **?**, **?**, **?**, **?**, **?**] in plan recognition of traffic scenarios have been done, this work is unique in that observations are performed from the eye level of the observer as the observer interacts with the domain too.

In this chapter, we define the constraints on our domain and represent these in a simulation. We then demonstrate the applicability of our approach as a plan recognition system to both recognize and define plans. We use the traffic domain as an easily understood real world domain and also because of its potential complexity to stretch the limits of our developed system.

4.2 Domain Representation

To represent traffic in our simulation we use a circle as an approximation of our vehicles. We discussed this in detail in Chapter 2, that circles are accurate and simple approximation slices through cylindrical objects and they lead to depth profiles as a simple representation of a relation between observer and agent. Vehicles in the domain are placed on a 2-dimensional Cartesian grid

and allowed free movement and varying speeds. This allows us to create traffic scenarios involving single or multiple vehicles performing real world manoeuvres.

The observer is also represented by a circle in the simulation but has a sensor array that can calculate the depth profile of the vehicles around it. We use the previously defined sweep line algorithm to act as our sensory array for calculations. Also, we are aware of our observers current location, speed and facing direction, as these are all attributes that are easy to track and calculate.

Using this representation we are able to simulate real world vehicle movements performing different plans. We can simulate turning, acceleration, as well as multiple bodies. We limit the data available to the observer about other vehicles to those which can be calculated from our simulated sensors. Location, sizes and movements of vehicles need to be calculated from depth profiles and their relation to the observer. Although this type of data collection is limited it mimics how a real world system would be used and how it is possible to develop a plan recognition system that is capable of performing from the eye level of an observer. It is important that we do not assume the observer knows which actions vehicles are executing. The observer uses sensing data to derive what actions and plans other agents execute.

4.3 Implementation

4.3.1 Modified Trans Interpreter

In this section we discuss the modifications we have made to the Trans/Final interpreter so we can recognize multiple plans over multiple bodies at the same time. We also show how we use actions in Golog as an interpretation of the movements performed by the observer. We create a logical model of the world, using the Situation Calculus and Golog, of the observer's movements and sensor data. We build upon the ideas proposed in the plan recognition framework by [**?**]. Through our modifications to the Trans/Final interpreter we maintain the idea of Golog procedures as plans and that procedures can include calls to other procedures. We do this to increase the modularity of the plan library. Also, we include the process of incremental plan recognition, where a plan can be recognized as being partially executed. Depending on the actions recently observed we can revise the set of plans that we initially believed were being performed by agents in the

environment.

We make three changes to the interpreter first proposed by [**?**]. First, we consider a modification of the single-step transition. Second, we limit the Golog constructs that can be used in the plan library(Golog procedures). Third, we require separation of actions from the plan recognition library. We separate actions from the plan library for two reasons. Plans that are being recognized should not have the ability to make the observer perform any actions. We are not developing a planning system, and therefore in the development of the plan library there should not be this option. Second,the observer is the only agent in the environment who's actions we are aware of. Although we are aware of the observer's actions, this system is not meant to control the observer. Therefore we have created a separate control statement in the interpreter which will perform logically equivalent actions to those which the observer performs in the simulator. At no point does the interpreter request the observer to perform an action. We use these logically equivalent actions to keep updating situation term that represents the actions the observer is performing independently in the simulator.

Since we recognize what plans agents in the domain are performing based on relations we have defined in the Situation Calculus, the plan recognition library does not contain actions. This limits the Golog operators we allow in the plan library to $?\phi$(tests),π(non-deterministic choice) , $if - then - else$ and $proc$ constructs. This allows us to construct the FSA using Golog's syntax that we described in the previous chapter, while allowing for modular procedures. Even with these limited operators, the expressiveness of the plan recognition library is only limited by the spatial relations between agents. We will show in the next section that these limitations in no way restrict the capabilities of the plan recognition system.

Finally, we discuss the modifications we have made to the single-step transition semantics. We have modified the original Trans/Final interpreter to allow parallel recognition of multiple plans and to get information about the observer's actions and sensor data from the interpreter. As well, we have created a generic structure for representing FSA as Golog procedures. We have designed the following modifications to build on top of the existing framework as much as possible.

The original online interpreter was defined as:

$$online(Prog, S0, Sf) \equiv \tag{4.1}$$

$$Final(Prog, S0) \wedge S0 = Sf \vee$$
$$Trans(Prog, S0, Prog1, S1) \wedge online(Prog1, S1, Sf).$$

We create another layer on top of the initial online interpereter that allows us to perform Trans and Final across multiple plans. We use the same ConGolog definitions of Trans/Final first defined in Section **??**, but create the higher level functions $doFinal\_All$ and $doTrans\_All$ to apply to multiple plans at a time.

$$doFinal\_All(\delta_L, S, \delta_{fin}, \delta_{rem}) \doteq \tag{4.2}$$
$$Final(\delta_{fin}, S) \wedge Final(\delta_{rem}, S)$$

$$doTrans\_All(\delta_L, S, \delta_{rem}) \doteq \tag{4.3}$$
$$Trans^*(\delta_L, S, \delta_{rem}, S) \wedge \forall \delta \, \epsilon \, \delta_{rem}(\delta = proc(ProcedureName(args), Body))$$

Where δ_L is a set of all plan recognition programs to which we wish to apply $Final$ or $Trans$. δ_{fin} and δ_{rem} are the sets of all plan recognition programs that are completed and all plan recognition programs that are remaining, respectively. Finally, S represents the current state. Equation 4.2 states that δ_{fin} is the set of all plan recognition programs that have been completed in the current situation S and that δ_{rem} is the set of all plan recognition programs that are not completed in the current situation S. δ_L is the set of plan recognition programs that contains both $\delta_{fin} \cup \delta_{rem}$.

According to Equation 4.3, a transition is performed on the set of all plans δ_L, until the only remaining plan recognition programs, δ_{rem}, is a set of plan recognition programs that only contain procedures as defined in the ConGolog definitions from Section **??**. We use this modified $Trans$ interpretation because we represent plan recognition programs as special types of procedures. These special procedures are designed to mimic the FSA we defined in the previous chapter. We will show in the next section how procedures can be defined to represent FSA and how the modified $Trans$ function, $Trans^*$, is capable of representing state changes that are integral to FSA. Specifically, how a single FSA state transition corresponds to a single-shot execution of a procedure body.

Using these new higher level $Final/Trans$ predicates we can define a new online interpreter

for plan recognition. In this new interpreter, we show how we have separated actions from the plan library, and in doing this can share a single situation argument across multiple plans. As we only have one agent who's actions we observe, having a single situation argument makes sense. We have one situation term representing the actions of our observer, and only these actions we can use with the relations we have defined to develop procedures. Our new online interpreter is below:

$$
\begin{aligned}
&online\_multi(\delta_{robot}, \delta_L, S_0, S) \equiv \\
&\quad \delta_L = nil \vee && (4.4a)\\
&\quad (doFinal\_All(\delta_L, S_0, \delta_{fin}, \delta_{rem}) \wedge \\
&\quad online(\delta_{robot}, S_0, S) \wedge && (4.4b)\\
&\quad doTrans\_All(\delta_{rem}, S, \delta_{next}) \wedge online\_multi(\delta_{robot}, \delta_{next}, S, SN)). && (4.4c)
\end{aligned}
$$

In all cases, $Final(nil, s)$ is true in any situation s. As stated earlier we are building this system on top of the previous ConGolog definitions, so we attempt reuse as much as possible. This new interpreter for multiple plan recognition on multiple bodies follows very closely the original $online$ interpreter but with some minor changes to increase its functionality and change its intended purpose from planning to plan recognition.

$online\_multi$ has four arguments, one of them is a procedure, δ_{robot}, that mimics the actions performed by the robot in the simulator updating the situation term each iteration. The plan recognition library, δ_L, which consists of a set of all plans that we will perform plan recognition for, along with the associated objects we are trying to recognize plans for. At the initial stage, we assume that all objects in the domain can be performing any plan, and therefore the initial list δ_L contains a separate plan with an argument representing the object we are associating with that plan. We will see how this works more in the following sections. The final two arguments are the initial situation term, $S0$, and the uninitiated variable, S, representing the next situation term. If $online\_multi$ is not at the initial iteration, the argument $S0$ can be viewed as the current situation.

The interpreter makes 3 steps in each iteration. First, it performs $doFinal\_All$, to trim from the plan library all plans, δ_{fin} that have completed in the current situation, $S0$. A plan is considered completed if a behaviour for an object has been recognized successfully or if an object is recognized as not able to complete this behaviour, regardless of future actions. Comparing this to a FSA,

a plan is completed if the automaton has reached a goal state, or fail state respectively. Therefore in this step we are checking if the FSA is in a goal or fail state. This leaves only the remaining plans, δ_{rem} after this step or FSA's that have not reached an end state.

Next, we call the original online interpreter on the special procedure δ_{robot}. We use this procedure to communicate with the simulator to gather data about the current situation. With this data, logically equivalent actions are executed by the interpreter to represent the actions performed by the robot in the simulator. These actions are added to the situation term, updating it, so further plan recognition can happen. This procedure starts with a wait action. We use this wait action to allow objects in the simulation to change position, data is not needed as a continuous stream. We make this choice to have small wait times, to limit the number of irrelevant calculations. A small change in data may have no affect on any of the plans, so we use the small wait time to decrease the possibility of calculations not having an effect on any plans currently being recognized. The amount of time to wait each cycle is calculated in the helping predicate $calcWait(Bodies, WaitTime)$, based on the speed of the objects in relation to the robot, as well as the distance of the object from the robot. If objects are moving faster, it is better to sample data more often. As well, if objects are closer to the robot, sampling data more often is important as these object will have more of an effect on the robot's immediate surroundings. After the waiting period, we connect to the simulated robot to get new data since we last connected to it. This includes new sensor data and a current position. We use this data to create logically equivalent actions to be added to the situation term. Our actions represent the current location of the robot($start/endmove$), how much it has rotated($start/endpan$), and the most current depth profile($sense$). Recall that each of these instantaneous actions requires a temporal argument representing a moment of time when the action is executed. All arguments, including locations $L1, L2$, angle of rotation, ω, and a depth profile are instantiated with data obtained from the simulator. Also, recall pi, a programming construct for a nondeterministic choice of arguments. In the procedure below, pi instantiates the variables in its argument and they are assigned values as the procedure executes. A robot is capable of not moving over a given period of time. In this case the same actions are matched, but the the location of the robot does not change. This procedure is below:

$$proc(robotLoop(Bodies), \tag{4.5}$$

$pi([WaitTime, L1, L2, T1, T2, T3, T4, T5, \omega, Profile],$

$?(calcWait(Bodies, WaitTime)) : wait(WaitTime) :$

$getSimulationData(L1, L2, T1, T2, T3, T4, T5, \omega, Profile) :$

$endMove(robot, L1, L2, T1) : startPan(\omega, T2) :$

$endPan(\omega, T3) : startMove(robot, L2, L3, T4) : sense(Profile, L2, T5))).$

After the online interpreter has updated the situation term, we can perform the next step. The next step is to use the new situation term to see how the changes have affected the current plans in the library. We perform $doTrans\_All$ on the remaining plans δ_{rem} using the current situation S. Using the current situation, $doTrans\_All$, we are able to calculate how the new situation, S, has affected the current set of plans being recognized. This step is the equivalent of an FSA receiving its next input and based on its transition function, changing its state. After $doTrans\_All$ has completed we have a new set of plans δ_{next}. However, note that the situation S does not change because the Golog programs in δ_{rem} have no occurrences of physical actions, but only tests and other constructs that do not change situations. From here the interpreter calls itself again, but with a new set of arguments, $online\_multi(\delta_{robot}, \delta_{next}, S, SN)$. This loop can continue until all plans are complete, successfully or not, or no more simulation data is received.

4.3.2 Golog Procedures as Plans

$proc(beginning(Body),$

$).$

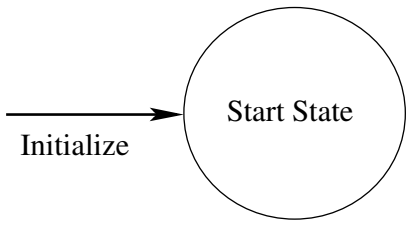

Figure 4.1: Beginning of a plan as a FSA

In this section we show how we can use Golog to define recognizable plans, that we use in our plan recognition system. We use Golog procedures to define our plans in a modular fashion, and from this we will see how our plans resemble the FSA we defined in the previous chapter. We will show step by step how a generic plan is defined, and its equivalence to a FSA. Upon

initialization of the plan recognition system, the plan library is populated with all plans in the library for all bodies. Each plan starts at the beginning of its Golog procedure, which is equivalent to the start state of an FSA. This is the beginning of the plan, we show below the Golog code and equivalent FSA representation of this step:

Next we need to define some constructs that can represent transitions between states of a FSA will work on. We use the $if - then - else$ statement in Golog to mimic the transition functions of an FSA. The condition that are used for our $if - then - else$ statements are fluents whose truth values are computed from the previously defined successor state axioms. Using nested $if - then - else$ we are able to limit the number of conditionals to check during each plan, as we can combine transitions with the same conditions into a single conditional statement.

$proc(beginning(Body),$

$\quad if(condition1,$

$\quad\quad ForwardTransition,$ $\quad\quad \%then$

$\quad\quad if(condition2,$ $\quad\quad \%else$

$\quad\quad\quad NoEffectTransition,$ $\quad\quad \%then$

$\quad\quad\quad FailTransition$ $\quad\quad \%else$

$))).$

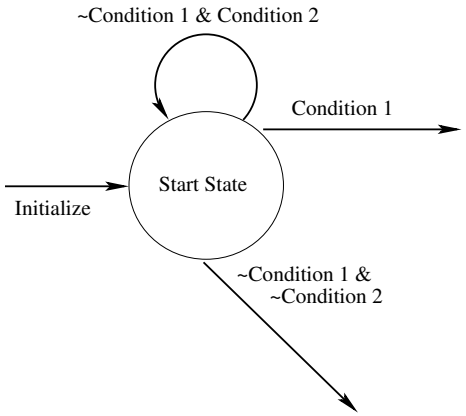

Figure 4.2: Adding Transitions to a FSA

Next, we connect this procedure to other procedures. We do this in a modular fashion, by calling procedures from within the procedure. Each procedure that is part of a plan recognizing program, is equivalent to a single state in a FSA. The conditionals that control flow from procedure to procedure within a plan are equivalent to transition functions in a FSA. This process of creating procedures, using nested $if - then - else$ statements and calling other procedures can be repeated until a complete FSA is modelled in the Golog language. Below is a one completed part of what a plan recognizing program can look like, along with its associated part of an FSA:

$proc(beginning(Body),$

$\quad if(condition1,$

$\quad\quad nextProcedure(Body),$ $\quad\quad \%then$

$\quad\quad if(condition2,$ $\quad\quad \%else$

$\quad\quad\quad beginning(Body),$ $\quad\quad \%then$

$\quad\quad\quad endProc(Body, fail)$ $\quad\quad \%else$

$))).$

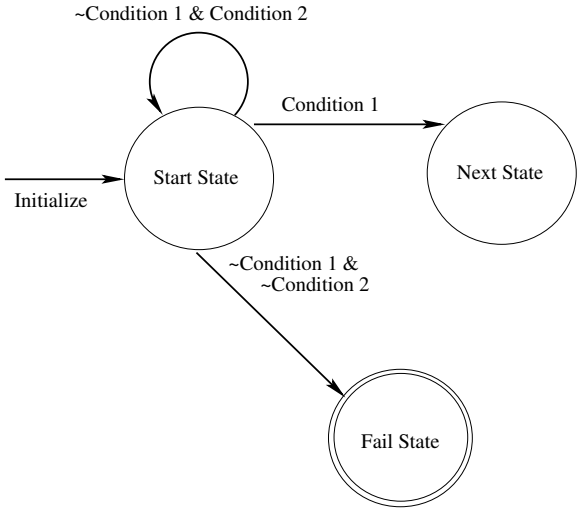

Figure 4.3: Connecting States in a FSA

We introduced in this final example a special procedure call $endProc(Body, fail)$. This is a special procedure that is only used when the plan has reached a point where a behaviour is not being performed, or a behaviour has been recognized. When a behaviour has been recognized, this special procedure will be called with a different second argument, $endProc(Body, success)$. This procedure is unique as it allows us to keep track of when a plan has ended and with what status. We introduce a special $Final$ construct to the original transition semantics introduce in Section **??** to handle this special procedure. This construct is $Final(endProc(Body, Status), S) \stackrel{\circ}{=} True$.

We also discussed earlier a modification to the single-step transition. This can be more clearly seen looking at the $Final$ generic procedure above. When $Trans^*$ is called on each plan, it does multiple single-step transitions until the next procedure is encountered, see Equation, **??**. When the next procedure is encountered this becomes the next step of the plan to be executed. This

allows us to check multiple conditional statements leading towards different possible procedures in one transition. These modifications allow us to use Golog to define plans that resemble the FSA we defined earlier, and recognize behaviours similar to the FSA from chapter 3.

4.3.3 Overtake

In this section we look at the example of an object attempting to overtake the robot. The Golog program responsible for recognizing this behaviour consists of several parts. We discuss below each part in turn. The overtaking behaviour is a continuous motion in space, but it can be divided into several natural segments with transitions between them. Each sub-motion corresponds to a part of overtaking behaviour.

FSA

We show the FSA for the overtake example as well as the conditions on each transition. Although in this example we do not include termination conditions on loops, this is shown in later examples. We exclude termination conditions on loops from this initial example to simplify the concepts being exemplified.

When designing an FSA for a procedure, we attempt to break that procedure into steps, where we can easily recognize changes in the bodies around the robot. We define the procedure representing overtaking behaviour that uses both relative direction of bodies to the robot and changes in size of depth peaks of bodies around the robot. We break each step into a sub-procedure as shown above in the general case, to show how we can define a modular plan for overtaking. Note: in Figure **??** there are some unlabelled transitions that lead to the fail state. These transitions happen only when none of the other transitions happen. A similar format for fail state transitions is used on other diagrams describing FSA's in this chapter.

If we look at Figure **??** there are a minimum of four transitions needed to reach the final state of this procedure. In the initial state, $overtake\_multi$, the FSA moves to the fail state if an object is not in a position to begin the overtaking procedure, this means the body is not behind the robot and the procedure will exit, and the FSA will move to the fail state. We define behind as a body being in one of three relative directions to the robot, $backof$, $rightof\_back$, or $leftof\_back$, see Section

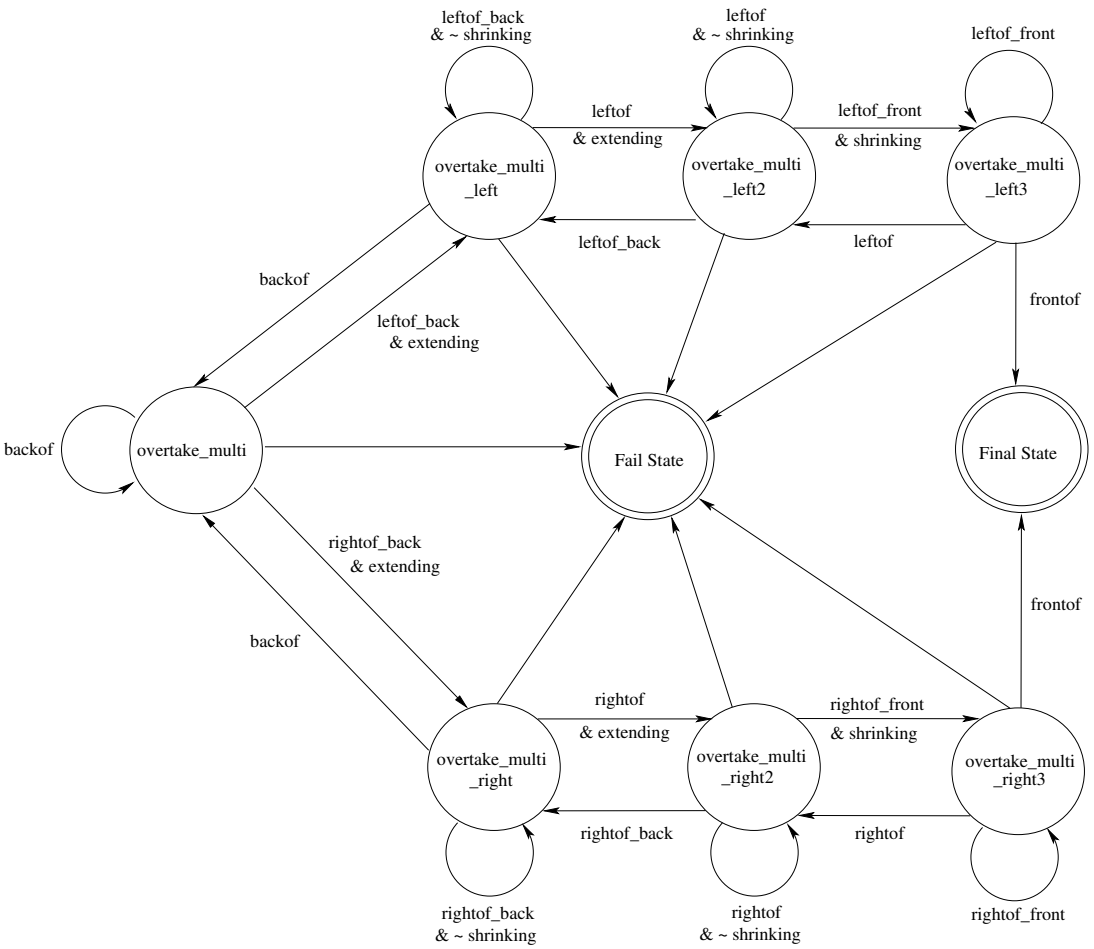

Figure 4.4: The FSA for Overtake

??. If a body remains behind the observer, then it remains in the initial state of the procedure. The body could still be meaning to overtake the robot, but it has not made a definite move toward overtaking, and it hasn't made a move that would let our system know it is not overtaking. If the body makes a move to the left or right of the robot and moves closer, than it has started the process of overtaking. The FSA will recognize this and move into the next state, $overtaking\_multi\_left$ or $overtaking\_multi\_right$, respectively. We include the condition that the body's depth peak be extending, as in a typical overtaking behaviour a body must increase its speed to complete such a manoeuvrer which would cause its depth peak to increase in size. The following Golog code is representative of the state $overtake\_multi$ and the transitions possible from it.

$$
\begin{aligned}
&proc(overtake\_multi(Body), \qquad (4.6)\\
&\quad pi([LBD, Depth, Size, L2],\\
&\quad if(backof(peakOf(Body, \_, \_, LBD), L2),\\
&\qquad (overtake\_multi(Body)),\\
&\qquad (if(leftof\_back(peakOf(Body, \_, \_, LBD), L2) \,\&\, extending(peakOf(Body, Depth, Size, LBD), L2),\\
&\qquad\quad (overtake\_multi\_left(Body)),\\
&\qquad\quad (if(rightof\_back(peakOf(Body, \_, \_, LBD), L2) \,\&\, extending(peakOf(Body, Dept, Size, LBD), L2),\\
&\qquad\qquad (overtake\_multi\_right(Body)),\\
&\qquad\qquad (end\_Proc(Body, fail))\\
&\qquad\quad))\\
&\qquad))\\
&\quad))\\
&).
\end{aligned}
$$

As overtaking on the left or right are identical paths an object can take, we will only look at the chance an object is overtaking on the right side. After the first transition toward the final state, the FSA is now in the state $overtake\_multi\_right$ as seen in Figure **??**. From this state we define four distinct transitions. First, there is the forward transition, if a body moves to the right of the robot and its depth peak is continuing to extend, then the body is also continuing the procedure of overtaking the robot. This represents the body moving in to a parallel position beside the robot. If the body moves behind the robot, to the position, $backof$, then the body wasn't able to continue forward in the procedure. The FSA moves back to the initial state, $overtake\_multi$, and the body has a chance to attempt to overtake the robot again. If the body remains in a similar position to the robot, $rightof\_back$, then it hasn't made a significant enough change in position to change states in the FSA. As long as its depth peak is not shrinking, it will remain in the same state until more information is received. Again, we exclude termination conditions on loops to keep this example simple. But if it was shrinking, then the body may be attempting some other procedure that is not overtaking. Finally, if none of these transition conditions are met, then we can assume the body

is not overtaking and the fourth transition occurs to the fail state and the procedure ends. The corresponding Golog code to this state can be seen below in procedure **??**

$$
\begin{aligned}
&proc(overtake\_multi\_right(Body), \\
&\quad pi([LBD, Depth, Size, L2], \\
&\quad if(rightof(peakOf(Body, \_, \_, LBD), L2) \,\&\, extending(peakOf(Body, Depth, Size, LBD), L2), \\
&\qquad (overtake\_multi\_right2(Body)), \\
&\qquad (if(backof(peakOf(Body, \_, \_, LBD), L2), \\
&\qquad\quad (overtake\_multi(Body)), \\
&\qquad\quad (if(rightof\_back(peakOf(Body, \_, \_, LBD), L2) \,\&\, \sim shrinking(peakOf(Body, Depth, Size, LBD), L2), \\
&\qquad\qquad (overtake\_multi\_right(Body)), \\
&\qquad\qquad (end\_Proc(Body, fail)) \\
&\qquad\quad)) \\
&\qquad)) \\
&\quad)) \\
&).
\end{aligned}
\tag{4.7}
$$

The next state, $overtake\_multi\_right2$, from Figure **??** is represented in the Golog code from equation **??**. Like the previous state this state also has four similar transitions. If the body remains to the right of the robot and its depth peak is not shrinking then it remains in the current state, as it has not made a significant enough change in position to change states. If the body changes relative position to be to the $rightof\_back$ of the robot, then the FSA will move to the previous state. This means the body wasn't able to yet finish overtaking the robot, but it hasn't made any distinct moves that show it will not finish overtaking in the future. Where this part of the plan changes is the forward transition. We are representing when a body starts to move ahead of the robot to be able to move in front of the observer, to complete overtaking. If the body moves to the $rightof\_front$ and its depth peak shrinks then it moves to the next state. The peak is shrinking as the body has to eventually move in front of the observer to complete overtaking. To do this it needs to move ahead of the observer, and this kind of action would cause the peak to shrink. Otherwise,

if none of these transition conditions are met, the FSA moves to the fail state and the procedure exits.

$$
\begin{aligned}
&proc(overtake\_multi\_right2(Body), \qquad (4.8)\\
&\quad pi([LBD, Depth, Size, L2],\\
&\quad if(rightof(peakOf(Body, \_, \_, LBD), L2)\ \&\ \sim shrinking(peakOf(Body, Depth, Size, LBD), L2),\\
&\qquad (overtake\_multi\_right2(Body)),\\
&\qquad (if(rightof\_front(peakOf(Body, \_, \_, LBD), L2)\ \&\ shrinking(peakOf(Body, Depth, Size, LBD), L2),\\
&\qquad\quad (overtake\_multi\_right3(Body)),\\
&\qquad\quad (if(rightof\_back(peakOf(Body, \_, \_, LBD), L2),\\
&\qquad\qquad (overtake\_multi\_right(Body)),\\
&\qquad\qquad (end\_Proc(Body, fail))\\
&\qquad\quad))\\
&\qquad))\\
&\quad))\\
&).
\end{aligned}
$$

In the last state before reaching the final state in Figure **??**, $overtake\_multi\_right3$, has four transitions. The forward transition happens when the body moves in $frontof$ the robot. This represents the body changing lanes to be in front of the observer, and completing the overtaking procedure by reaching the final state. The FSA remains in the current state if the body remains to the $rightof\_front$ of the observer. We do not include the conditions representing negations, $not\ extending$ or $not\ shrinking$ as the body in this position may be doing either of these changes, depending on how much space it needs to change lanes. It may be $shrinking$ if it needs to have more forward space before changing lanes or it may be $extending$ if it has started changing lanes, but has not made a significant enough change in position to change states. Similarly to the previous state, if the body moves to the right of the robot, then it changes state to the previous state. Otherwise if none of these transition conditions are met it moves to the fail state. This state is represented by the Golog code in equation **??**.

$$
\begin{aligned}
&proc(overtake\_multi\_right3(Body), \\
&\quad pi([LBD, Depth, Size, L2], \\
&\quad if(rightof\_front(peakOf(Body, \_, \_, LBD), L2), \\
&\qquad (overtake\_multi\_right3(Body)), \\
&\qquad (if(frontof(peakOf(Body, \_, \_, LBD), L2), \\
&\qquad\quad (end\_Proc(Body, success)), \\
&\qquad\quad (if(rightof(peakOf(Body, \_, \_, LBD), L2), \\
&\qquad\qquad (overtake\_multi\_right2(Body)\ ), \\
&\qquad\qquad (end\_Proc(Body, fail)) \\
&\qquad\quad)) \\
&\qquad)) \\
&\quad)) \\
&).
\end{aligned}
\tag{4.9}
$$

By splitting the procedure into smaller sub-procedures we are able to encode each state from our FSA and the transitions between the states. We will now look at two examples of a body overtaking and how the bodies relative positions to the observer relate to the corresponding Golog Code and states in the FSA from Figure **??**,

Examples

In the first example of overtaking we show a number of screen shots from the software that show important moments of a body overtaking. In Figure **??**, the initialization has just started and both the body and observer are about to start moving. Initially the body starts behind the observer and its depth peak reflects this in the graph. Note that although Figure **??** appears to have two peaks for one body, it is in fact only a single peak split around the boundary limits of the window. The window has a range from 0 to 360 degrees. If a body has any portion of its perimeter lying on the boundary, then it will be split across the outside boundaries of the window.

Figure 4.5: Initialization For Overtake Example

Figure **??** represents the first time that the robot has collected data from its sensors and applied the $Trans*$ predicate to the plans it is trying to recognize. In this case we are only trying to recognize if the body is overtaking us. When Trans is applied, we find out that the body remains in the initial state, $overtake\_multi$ from **??** , as the body is still behind the observer.

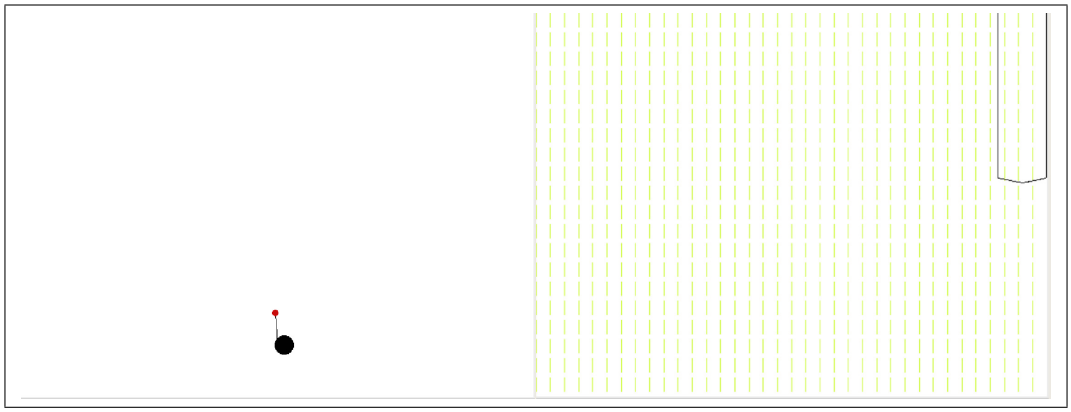

Figure 4.6: Body Remains Behind Observer

In Figure **??**, the observer samples more data from its sensors. It finds that the body has moved to the back of right, and that the peak is extending because the body has moved more closely to the robot. It therefore has met the conditions to change states, and this is evident after $Trans*$ was completed at this stage. The procedure that remains to be executed is $overtake\_multi\_right$. Which matches up with what we would expect based on the graph in Figure **??** and the conditions

on state transitions from Figure **??**.

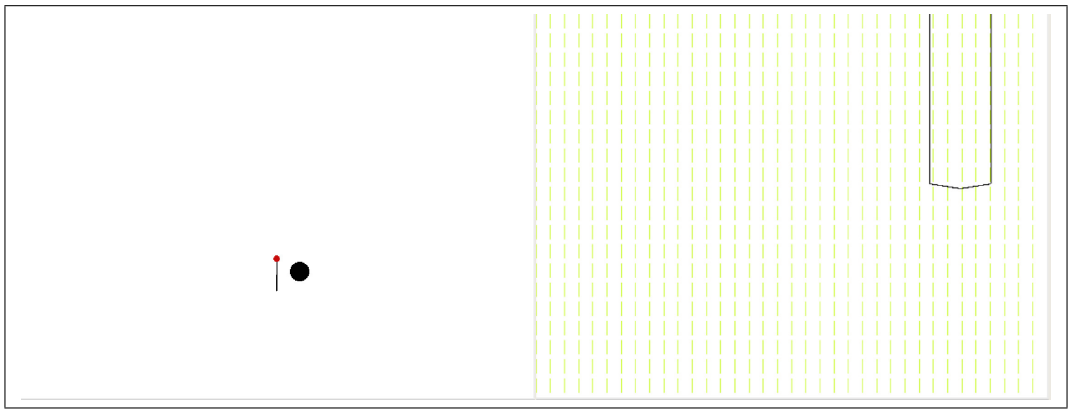

Figure 4.7: Body's Peak Continues to Extend and Moves to the Right of Observer

In Figure **??**, the observer samples more data from its sensors. It finds that the body has moved to be to the right of the observer. The peak has also become larger than it was when data was previously sampled because the body keeps approaching the robot. Therefore, according to Figure **??** it has met conditions again to change states. This is backed up again by our program output, which states the remaining procedure to be executed after $Trans*$ was applied to the current situation is, $overtake\_multi\_right2$.

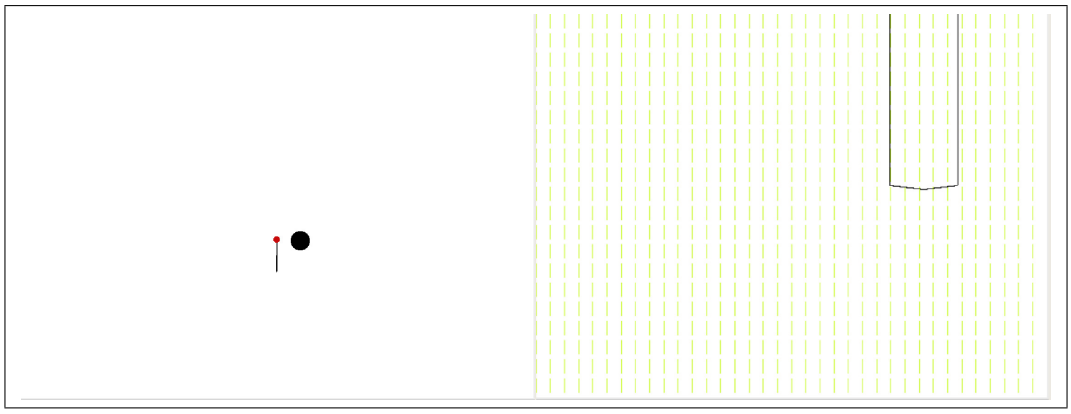

Figure 4.8: Body's Peak Continues to Extend and Remains to the Right of the Observer

In Figure **??** we notice that the peak has started shrinking and the body has moved into a position relative to the $rightof\_front$ of the observer. According to Figure **??** this would cause

a transition change. This is supported by the program output after $Trans*$ is applied to the most recent sensor data, that the remaining procedure is $overtake\_multi\_3$. This is easy to see from the position of the body in relation to the observer in Figure **??**.

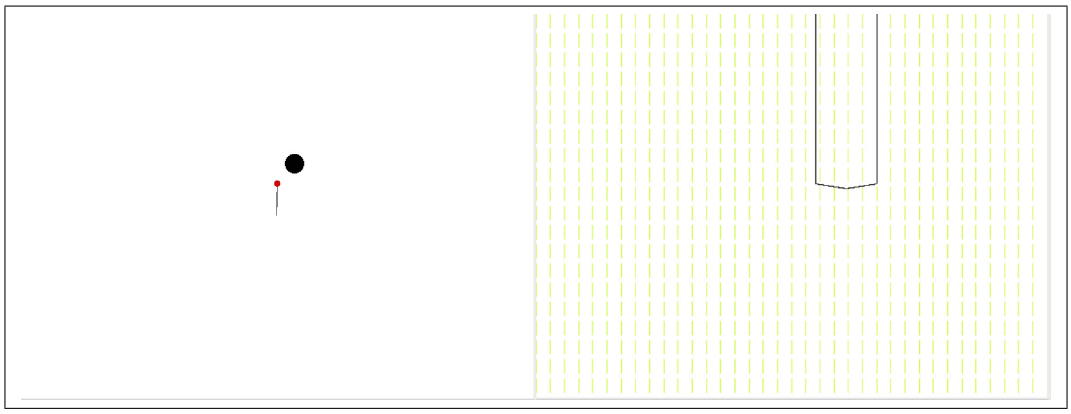

Figure 4.9: Body's Peak Starts to Shrink and moves to the Right of Front of the Observer

At this point the procedure is almost complete. After $Trans*$ is applied to the new data gathered from its sensors the procedure reaches the special predicate $endProc(Body, success)$. This is supported by Figure **??**, where it shows the body starting to change lanes and its peak is extending compared to the previous situation.

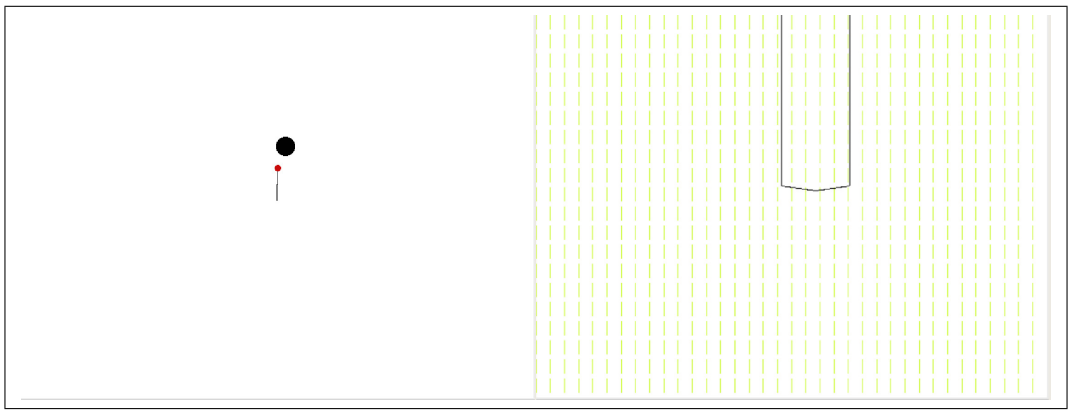

Figure 4.10: Body's Peak Extends and Moves in Front Of the Observer

At this point in Figure **??**, there are no more procedures to be executed in the plan library. No more data is gathered, and it is clear from the first figure up to this point that the body has overtaken the observer.

Figure 4.11: Body has Completed Overtaking the Observer

In the previous example, the overtaking procedure happened relatively straightforward, with only forward transitions being applied. In the next example, we show the power of our method of recognition. We show how using relative direction and position to define procedures allow us define a single plan recognition program that can encode the movement of objects in many absolute directions. Such as, if the bodies are overtaking from north to south, or east to west, the same program can be used to recognize this behaviour. Also, in this next example, how using our method of defining plans we can handle the case where a procedure can take longer or shorter amounts of time or steps to execute, but still be recognized by the same plan recognition program.

Similar to the previous example, the body starts behind the observer. In this example of overtaking, the body takes a longer amount of time to pass the observer. In this case it is due to the observer and body moving at similar speeds, therefore it takes longer to complete the procedure.

After the $Trans$ predicate is applied to the first set of data gathered from the sensors, the remaining procedure is $overtake\_multi\_right$. We know that the transition from the initial state to achieve this is $rightof\_back \,\&\, extending$ and this condition being true is supported by the changes visible in the graph and diagram in Figure **??**.

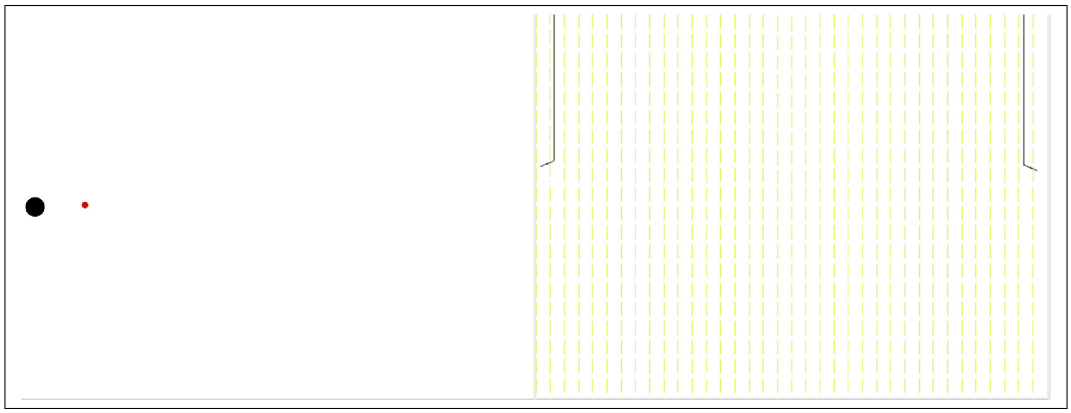

Figure 4.12: Initialization of Overtake Example

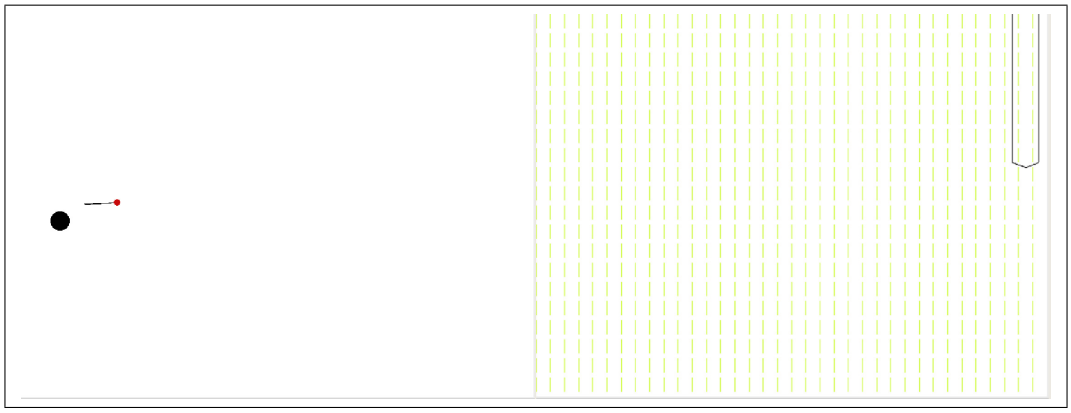

Figure 4.13: Body is Right of Back of Observer and Extending

In the next Figure, **??** the body has remained to the $rightof\_back$ and the peak is not $shrinking$. We expect that after the $Trans*$ predicate is applied to the current situation that it will remain in the same state, $overtake\_multi\_right$ from Figure **??**. The output from our software supports this as the remaining procedure to be executed is $overtake\_multi\_right$. The same occurs in Figure **??**, after more sensor data is gathered and $Trans*$ is applied.

Looking at Figure **??** we see that the body has moved to the right of the observer and its peak has extended. When $Trans*$ was applied to the current situation it returned the remaining procedure as $overtake\_multi\_right2$, which is what we expected to happen based on the FSA we constructed in Figure **??**. We can see that even though it has taken longer to get to this state than in the first example, the body still follows a similar pattern. Because of our abstraction of movements

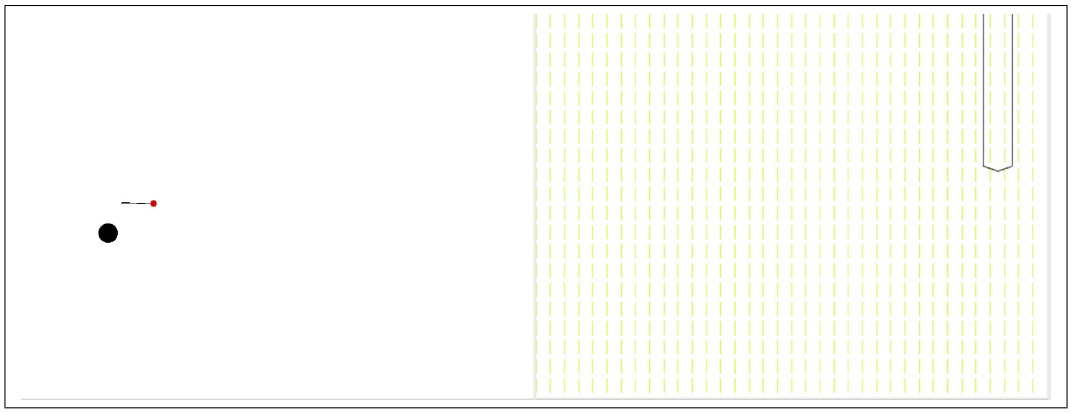

Figure 4.14: Body is Still Right of Back of Observer and Not Shrinking

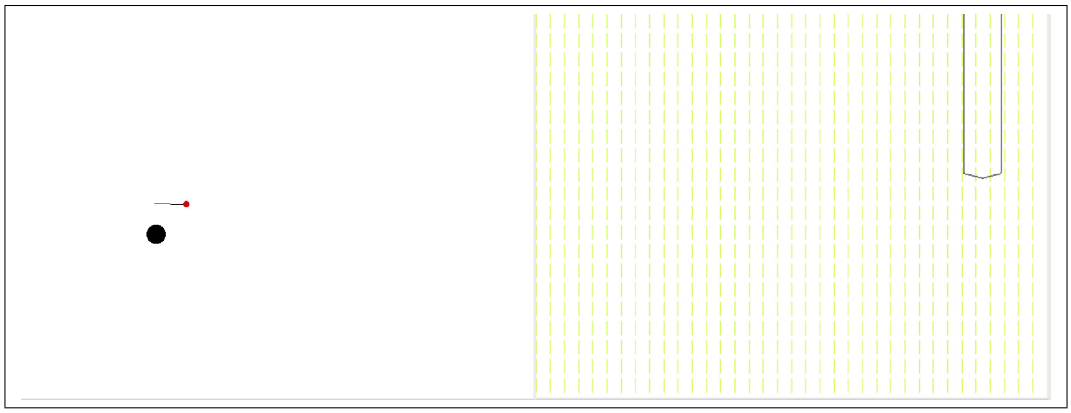

Figure 4.15: Body is Still Right of Back of Observer and Not Shrinking

to relations between objects, we are able to encode both examples in a single FSA.

New sensor data is collected in Figure **??**. As can be seen from the Figure, the body remains to the right of the observer. The body will remain in the current state, $overtake\_multi\_right2$, and this is supported both by the return from our $Trans*$ predicate and by looking at the FSA in Figure **??**.

Finally in Figure **??**, the body is getting close to completing the procedure. The peak in the Figure has clearly shrunk from the previous situation and the body is clearly to the $frontof\_right$. According to our FSA, the body would change states under these conditions and this is supported by the output of our $Trans*$ predicate. They both show that the procedure remaining and the

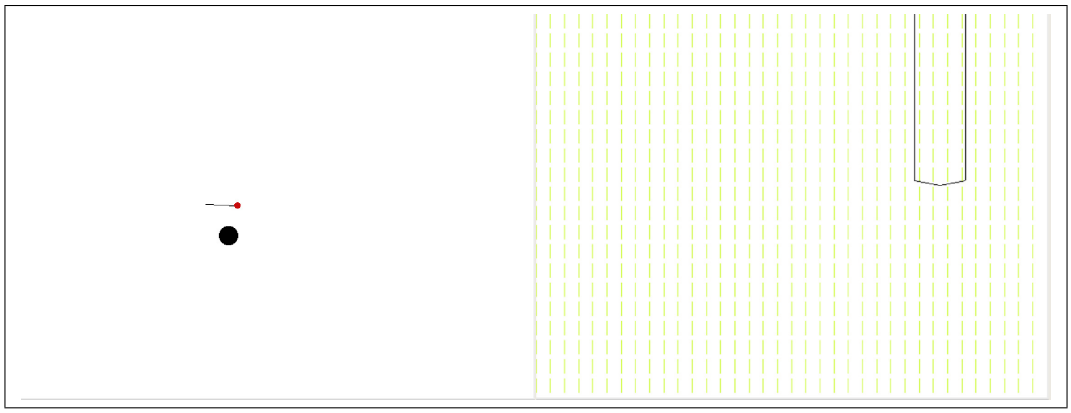

Figure 4.16: Body Has Moved to Right of Observer and is Extending

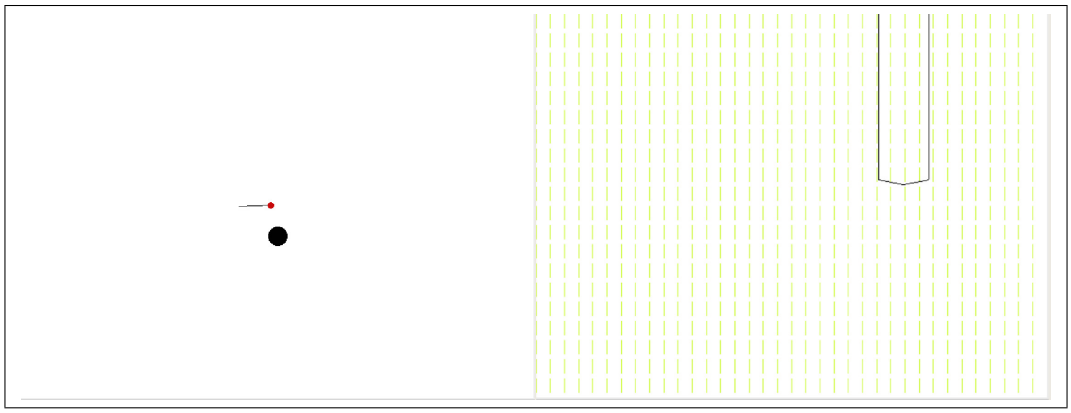

Figure 4.17: Body is Still to Right of Observer and is Not Shrinking

current state is $overtake\_multi\_right3$.

In Figure **??** the body has clearly changed lanes to be in front of the observer and its peak has extended as a result. The FSA has reached the final state and the procedure has completed successfully after the $Trans*$ predicate was applied to the current situation. We have shown that the plan developed for $overtake$ can recognize situations where the procedure takes longer to compute and when this behaviour happens in different directions. We will now move on to look at a more complex procedure, bodies making right turns at an intersection in relation to the observer.

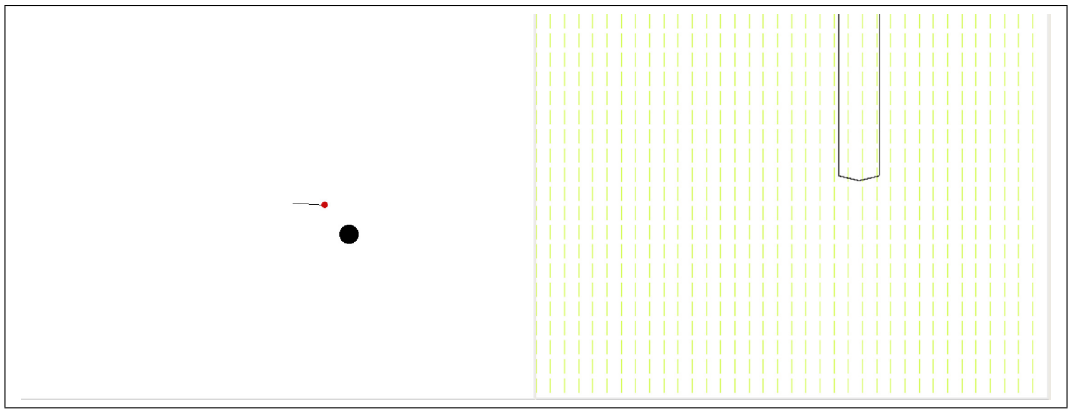

Figure 4.18: Body is to Right of Front of Observer and is Shrinking

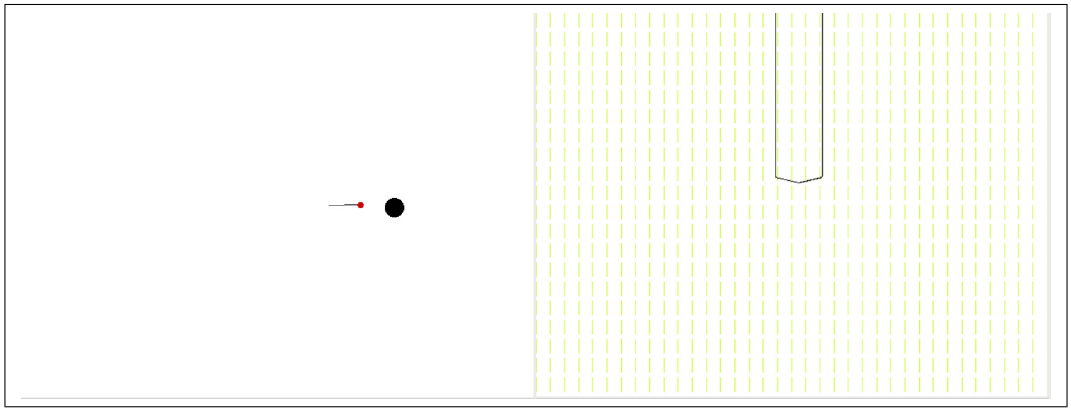

Figure 4.19: Body has Moved in Front of Observer and is Extending

4.3.4 Right Turn

FSA

In this example we have developed a plan to describe three different right turns a body could be making in relation to the observer.

The Figure **??** shows two of the possible right turns from the FSA in Figure **??**. Also, in Figure **??** we do not label the transitions from states that lead to the fail state. The FSA will transition to the fail state if no other transition occurs. In the right turn example we also use for the first time a counter to limit the amount of time that can be spend in the same state, this was first discussed in Section **??**. The limits on the counters were found through experimentation on

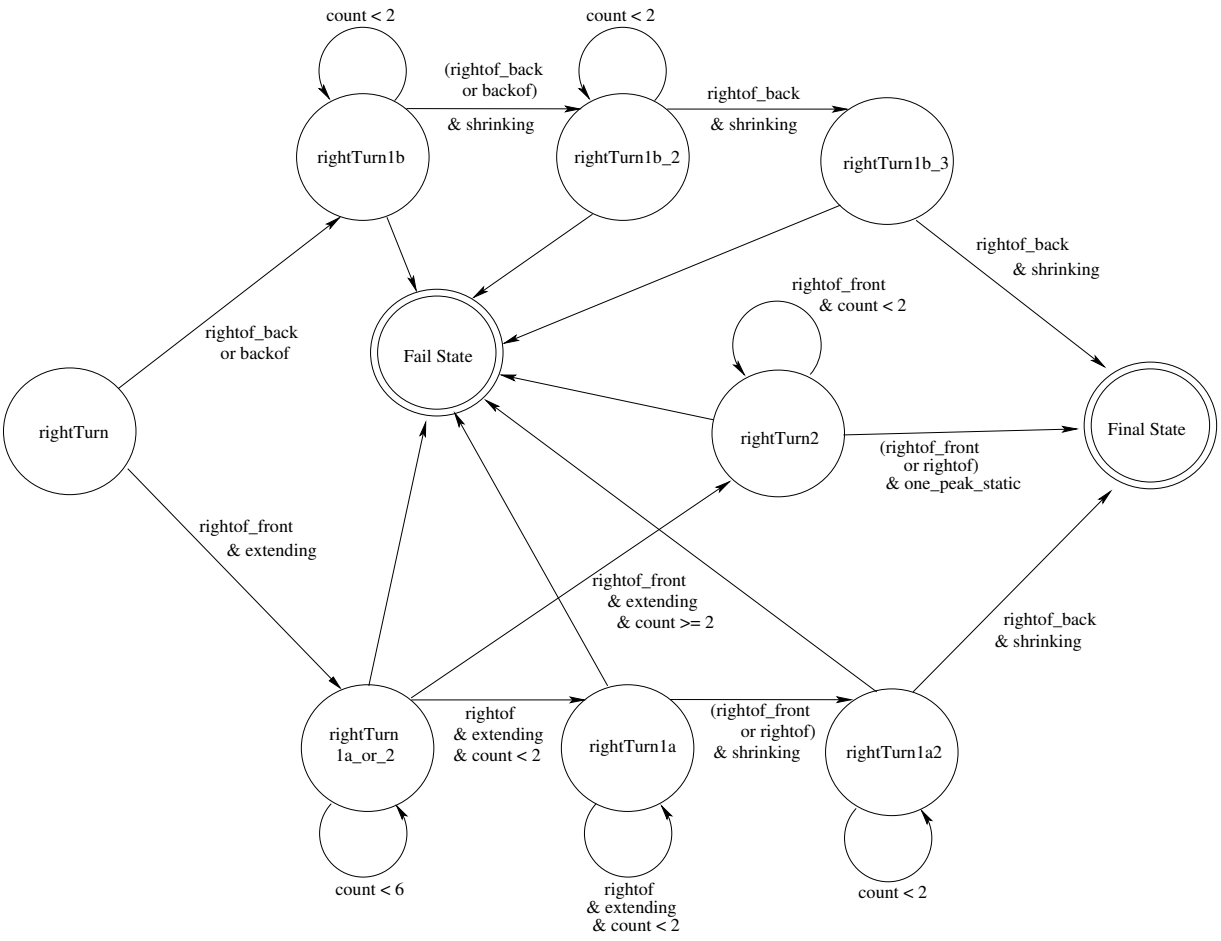

Figure 4.20: The FSA for 3 Different Right Turns

test data. The limits produced the most accurate results on a range of data. In each case the bodies are expected to approach the turn, slow down as they are turning and then gain speed as they enter back into a straight path. Although we don't explicitly recognize the speed of objects, we see this as being a useful extension of the axioms explored in this thesis to recognize more complex behaviours. We will show how we can use our plan recognition system to recognize the more complex movements of turning and recognize two different bodies at the same time. The numbers in the Figure represent the path the bodies will be taking as well as the observer's path. The body at the bottom of the diagram is about to execute a right turn that is represented by $rightTurn1b$ in the FSA. The second body in the diagram represents $rightTurn2$ from the FSA.

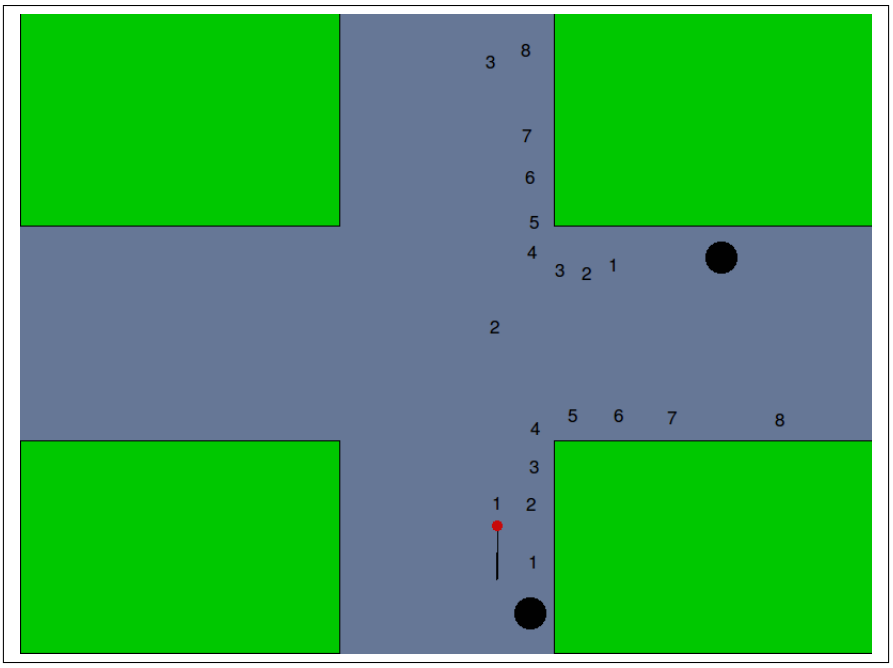

Figure 4.21: Example Paths of Right Turn 1b and Right Turn 2 from FSA

Recognition

The corresponding Golog code for this example can be found in the Appendix **??**. It follows the same approach for converting FSA to Golog that we have used in the previous example and throughout the paper. In this example we will look more closely at the simulator output, and how it follows the modified Trans interpreter that we described previously. Output from the program will be highlighted in bold. We will refer to the FSA and images from the software to understand the output.

In Figure **??** we have reached the initialization phase of the plan. At the initialization stage the interpreter sets up the procedures it wishes to recognize. In this case it is trying to recognize if the bodies in the domain are attempting right turns. We include a complete trace in Appendix **??**. The initial output of the software at this stage is **Initialized Multi Procedures: [rightTurn(body0), rightTurn(body1)]**. At this point we are ready to enter into the recognition loop $online\_multi$ explained earlier in Section **??**.

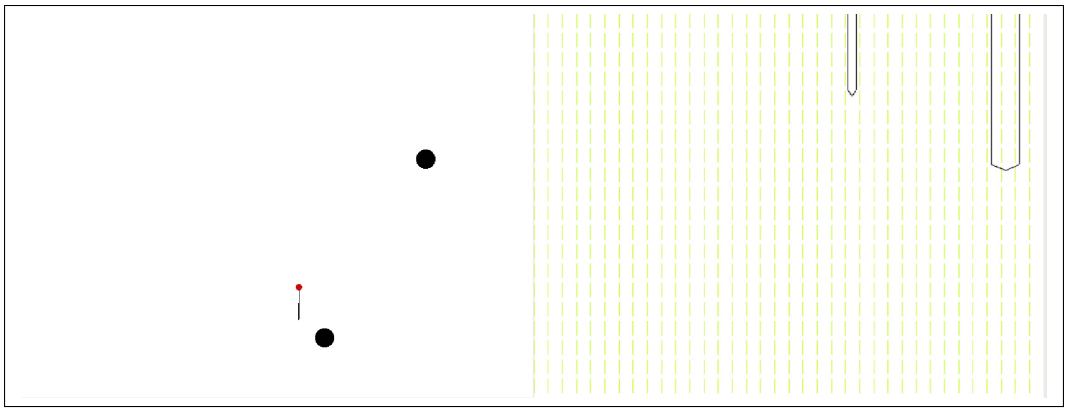

Figure 4.22: Initialization of Right Turn With Two Bodies

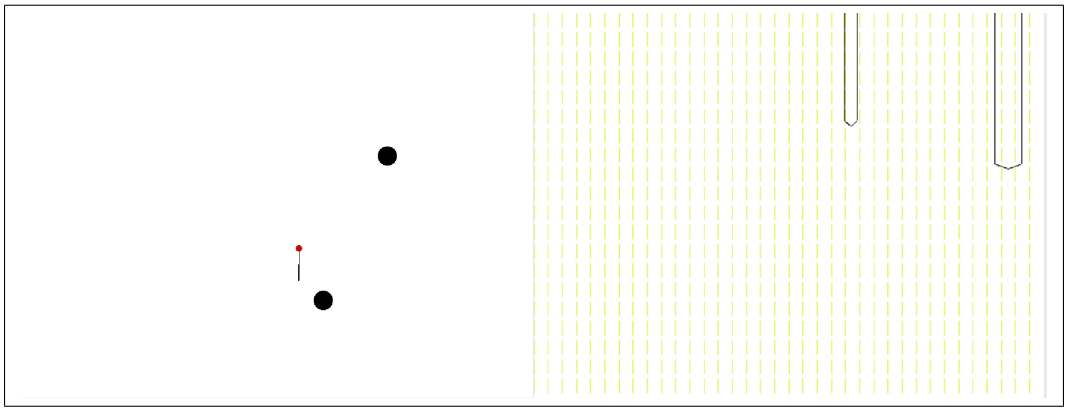

Figure 4.23: All Objects Have Started Moving

In Figure **??** the bodies have started moving and the first set of data is collected. As we can see from the output below, none of the procedures have completed, as when $Final$ is applied to the remaining procedures there are none that are in a final stage. But after the data is collected and $Trans*$ is applied to the procedures we see that the bodies may be performing a right turn. One body is behind us and the other is in front of us and this corresponds with the results from $Trans*$ below. We can see the first example of the counter as the second argument of both plan recognition programs below.

Done Do Final:

Robot Control Loop

Done Do Trans: [rightTurn1b(body0, 0), rightTurn1a_or_2(body1, 0)]

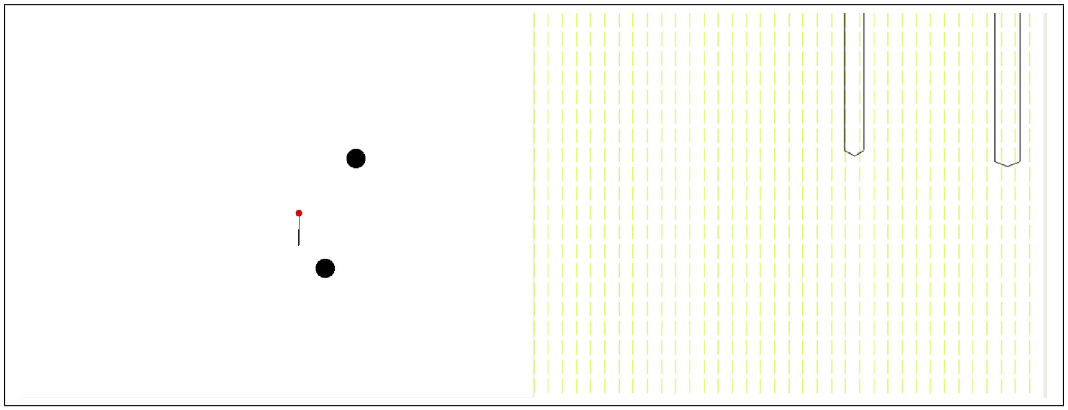

Figure 4.24: One Body is Starting Its Turn, While The Other May Be Turning

In Figure **??** we can be rarely certain that the body behind us is going to be completing a turn soon. Although the other body has not made any definite moves as to which right turn it may be performing. This is supported by the output from the $Trans*$ predicate below, where body0 is into the next state, but body1 is still close to the start of the FSA.

Done Do Final:

Robot Control Loop

Done Do Trans: [rightTurn1b_2(body0, 0), rightTurn1a_or_2(body1, 1)]

In Figure **??** one body's peak continues to shrink and change states in its path in the FSA. The other body base on the Figure is most likely not making a $rightTurn1a$ as defined in the FSA for right turns. But at this time it also has not made a definite decision to merge into traffic with a right turn or stop at the intersection. This is supported by the program output below, where the second body has not changed states after Trans was applied.

Done Do Final:

Robot Control Loop

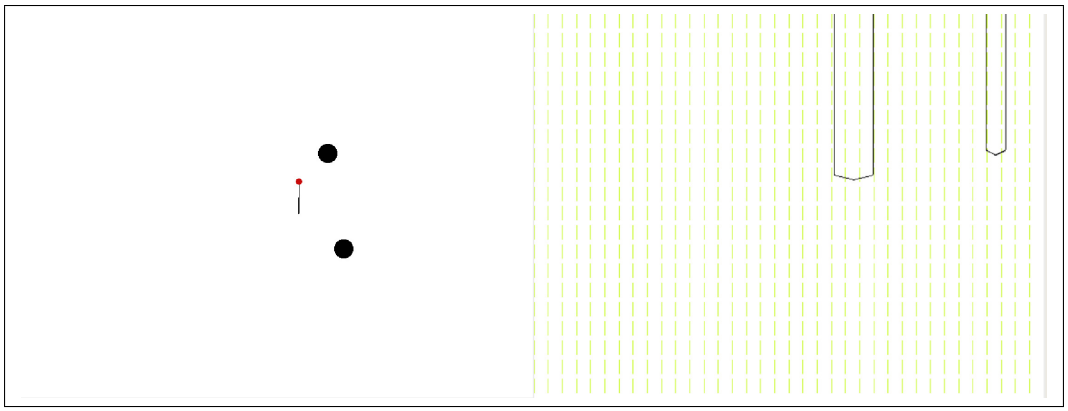

Figure 4.25: One Body is Close to Completing a Turn, While Another Approaches the Turn

Done Do Trans: [rightTurn1b_3(body0), rightTurn1a_or_2(body1, 2)]

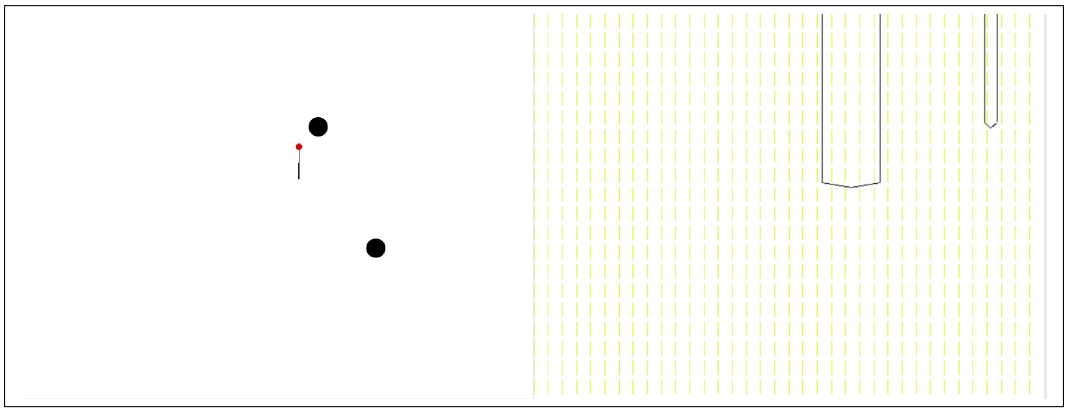

Figure 4.26: One Body Completes a Right Turn, Another Gains Speed and Starts to Merge

Based on Figure **??** the body behind has completed its right turn and this is supported by the output of the program below. The other body has entered into a lane slightly in front of the observer and this is supported by the output below as it has change states to $rightTurn2$ which is designed to represent merging. If this body is making a right turn we can expect it to gain speed and try to match the speed of traffic around it.

Done Do Final:

Robot Control Loop

Program Output: ”body0 Made a Right Turn”

Done Do Trans: [end_Proc(body0, success), rightTurn2(body1, 0)]

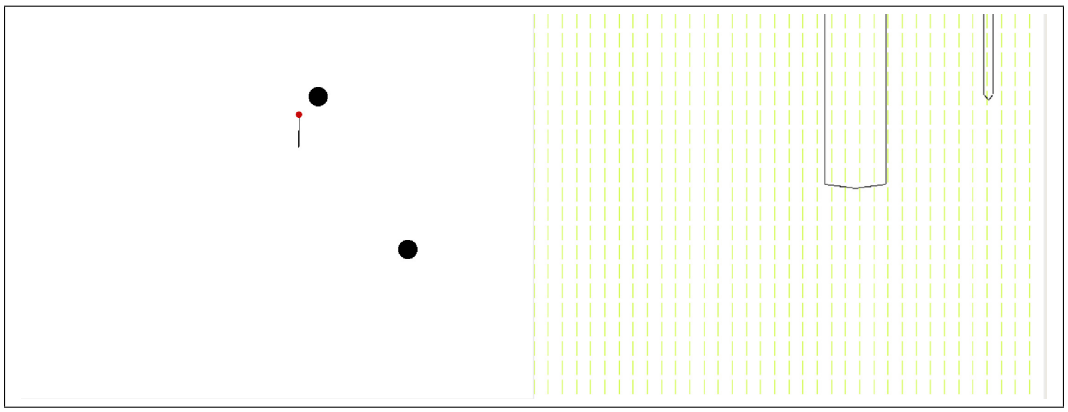

Figure 4.27: One Body is Far Behind, Another is Attempting to Match Speed

From the output below we see that one body has completed its turn and is removed from the list of procedures to be recognized by the $Final$ predicate. The remaining body has not managed to match the speed of surrounding traffic yet, and unless it does soon, it may have been performing some other operation.

Done Do Final: [end_Proc(body0,Success)]

Robot Control Loop

Done Do Trans: [rightTurn2(body1, 1)]

Based on Figure **??** the body is close to matching the speed of traffic around it, as the size of its depth peak has changed very little. We will see in the next iteration if it finally matches the speed of traffic around it.

Done Do Final:

Robot Control Loop

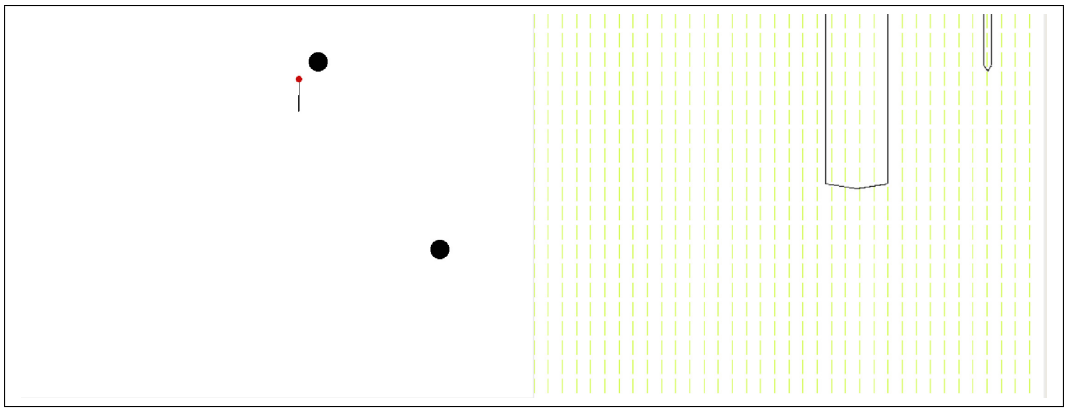

Figure 4.28: A Body is Trying to Match the Speed of Traffic Around It

Done Do Trans: [rightTurn2(body1, 2)]

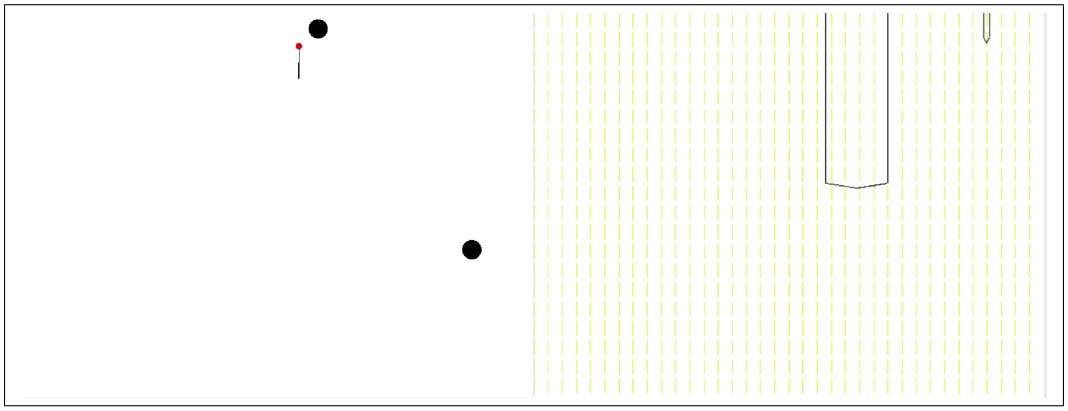

Figure 4.29: The Body Matches Speed of Traffic

If we compare the depth peaks from Figures **??** and **??**, we can see that the remaining body has matched the speed of traffic around it. It has successfully made a right turn merging into traffic.

Done Do Final:

Robot Control Loop

Program Output: ”body1 Completed right turn (joined traffic)”

Done Do Trans: [end_Proc(body1,success)]

Done Do Final: [end_Proc(body1,success)]

In this example we have shown that we can recognize what multiple bodies are doing at a time and by following the output of the interpreter we can make guesses about what the body may be doing before it has fully completed the procedure. In the next example we will look at how this technique can be applied to multiple bodies with different procedures and still produce the desired results.

4.3.5 Multiple Body Plan Recognition Example

In this section we bring together all the examples so far and introduce a new plan recognition program for a left turning behaviour. After showing the new left turn behaviour, we show an example of recognition with multiple bodies and multiple plan recognition programs.

Left Turn FSA

We have created a FSA that can recognize four different left turns from the point of view of the observer if the observer is moving through an intersection. In Figure **??** we do not include transitions to the fail state to make it easier to read, but if no other transition is made then it is assumed that the FSA will transition to the fail state.

Left Turn Example

We will look at a single example of left turns before moving onto the multiple body plan recognition example. This left turn example corresponds to $leftTurn2$ in the FSA, Figure **??**. Other examples of left turn behaviours can be found in Appendix **??**, as well as the related Golog code for these examples.

$$
\begin{aligned}
&proc(leftTurn(Body), \\
&\quad pi([LBD, D, S, L2], \\
&\quad if(frontof(peakOf(Body, \_, \_, LBD), L2), \\
&\qquad leftTurn1(Body), \\
&\qquad if(leftof(peakOf(Body, \_, \_, LBD), L2) \vee leftof\_front(peakOf(Body, \_, \_, LBD), L2),
\end{aligned}
\tag{4.10}
$$

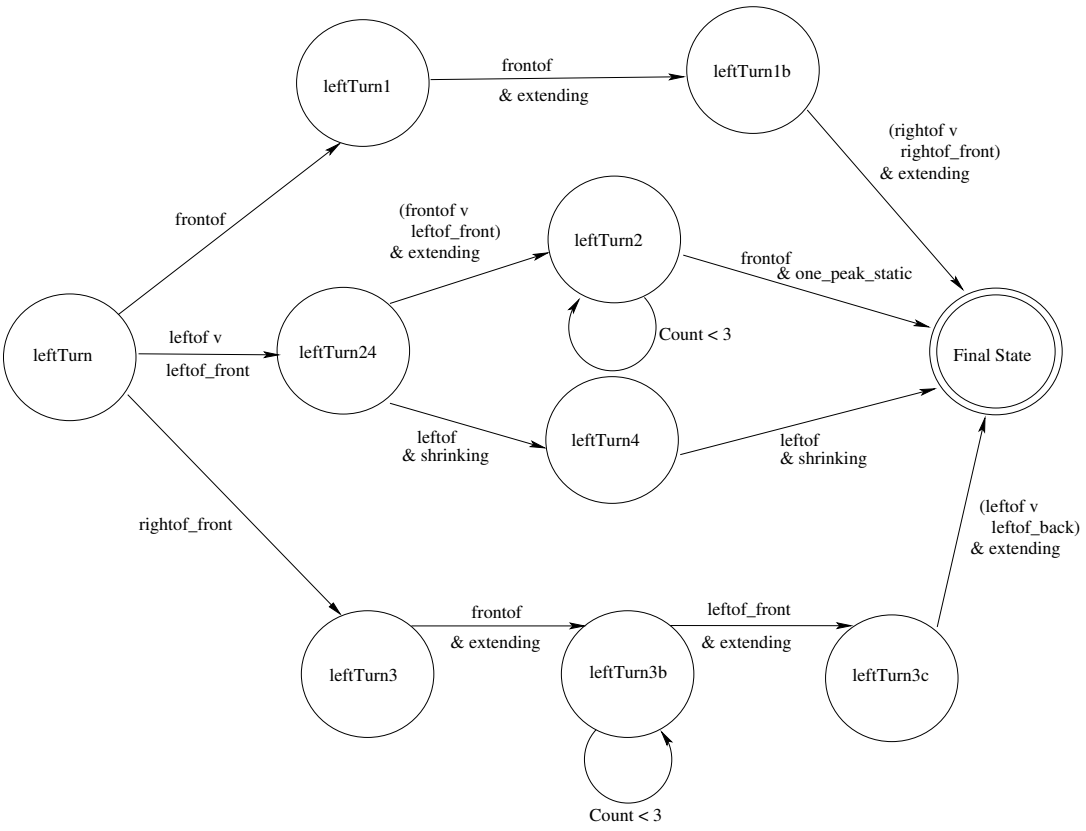

Figure 4.30: The FSA for 4 Different Left Turns

$$
\begin{array}{l}
leftTurn24(Body), \\
if(rightof\_front(peakOf(Body, \_, \_, LBD), L2), \\
\quad leftTurn3(Body), \\
\quad end\_Proc([Body, failure]) \\
\quad) \\
\quad) \\
) \\
)
\end{array}
$$

).

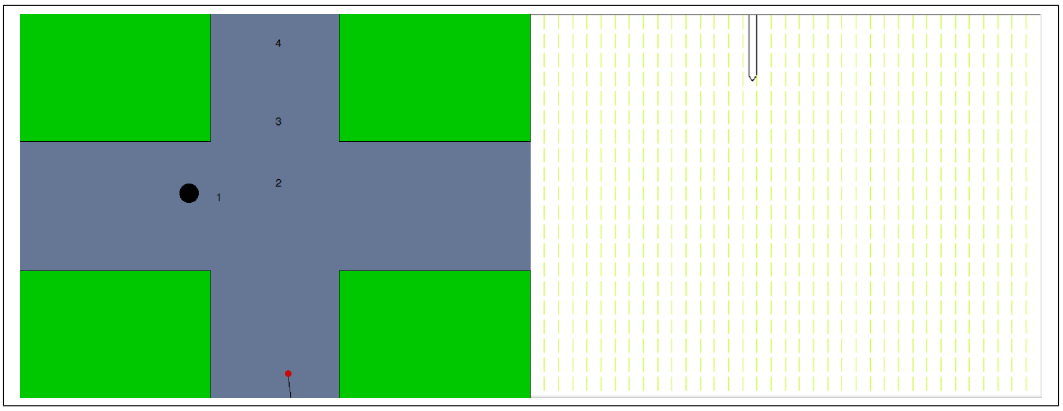

Figure 4.31: Initialization

In Figure **??**, the simulator has just started and is gathering the initial situation. In this set of figures we are trying to recognize a left turn behaviour.

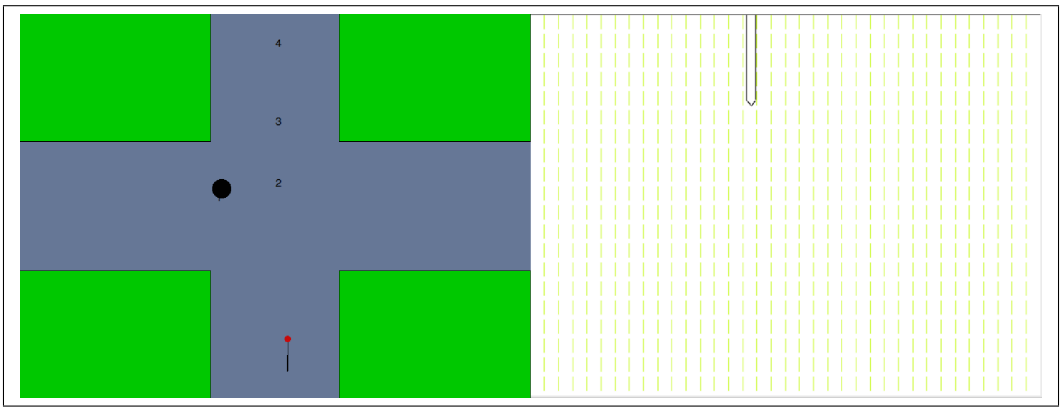

Figure 4.32: Body is to the Left of Front of the Observer

In Figure , we have gathered our first set of data. If we look at the Figure and at the Golog code in **??**, we can see only one of the three possible transitions make sense. The transition that matches the Figure and data gather is $leftTurn24$.

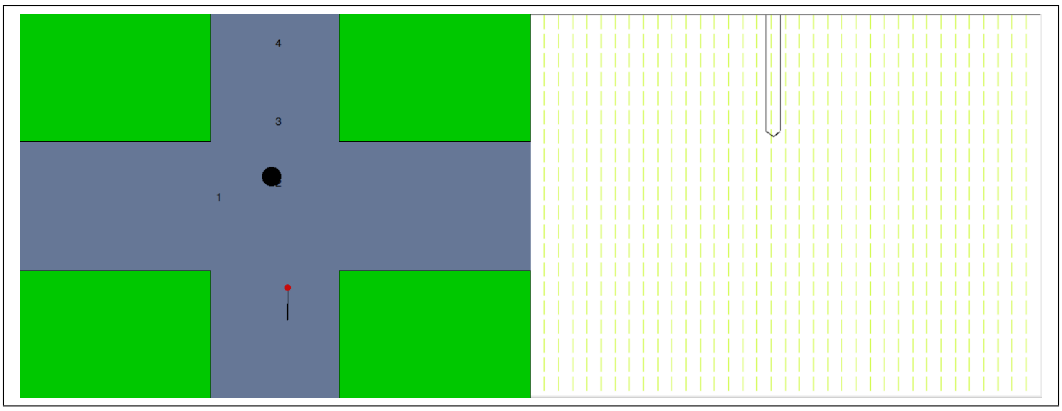

Figure 4.33: Body is Moving in Front of Us and Peak is Extending

Since we have made a transition, we will look at the next section of the relevant Golog plan recognition program:

$$
\begin{aligned}
&proc(leftTurn24(Body), \\
&\quad pi([LBD, D, S, L2], \\
&\quad if((frontof(peakOf(Body, \_, \_, LBD), L2) \vee leftof\_front(peakOf(Body, \_, \_, LBD), L2)) \\
&\qquad \&extending(peakOf(Body, \_, \_, LBD), L2), \\
&\qquad leftTurn2(Body, 0), \\
&\qquad if(leftof(peakOf(Body, \_, \_, LBD), L2) \,\&\, shrinking(peakOf(Body, \_, \_, LBD), L2), \\
&\qquad\quad leftTurn4(Body), \\
&\qquad\quad end\_Proc([Body, failure]) \\
&\qquad) \\
&\quad) \\
&\quad) \\
&).
\end{aligned}
\tag{4.11}
$$

In Figure **??** we notice that the body is now in front of the observer and that is peak has extended. This cooresponds to a transition in from the Golog plan recognition program **??**. In this case, the transition would be made to $leftTurn2$. Also, we notice that $leftTurn2$ has two

arguments. As mentioned previously the second argument is a counter, that counts how many times a transition has been made to the same state.

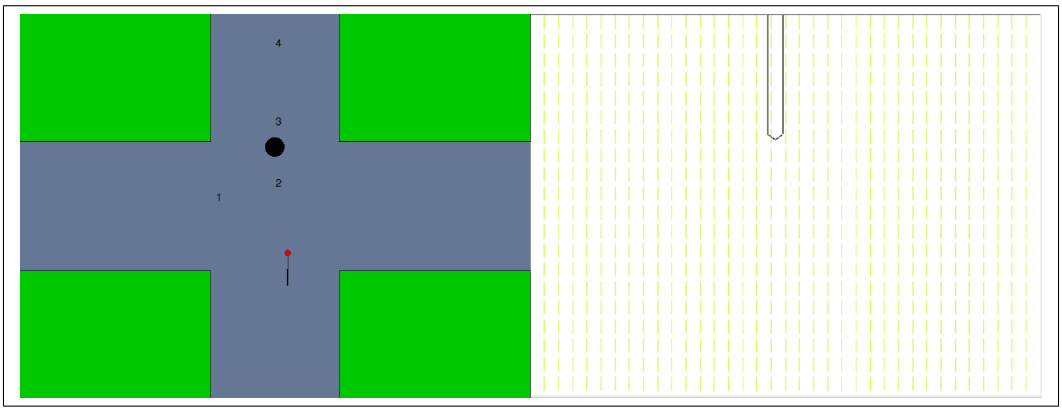

Figure 4.34: Body Remains In Front of Observer

The next section of Golog code, **??** is the final part of the plan recognition program. As long as the body remains in front of the observer, it is allowed up to 3 iterations by the $Count$ limit, to match the speed of the observer. Figures **??** to **??** exemplify how it takes multiple iterations for the body to match the speed of the observer, and therefore for its peak to remains static.

$$
\begin{aligned}
&proc(leftTurn2(Body, Count), \\
&\quad pi([LBD, D, S, L2], \\
&\quad if(frontof(peakOf(Body, \_, \_, LBD), L2)\,\&\,one\_peak\_static(peakOf(Body, D, S, LBD), L2), \\
&\qquad (?(write(\text{"}LeftTurn2\text{"})) : end\_Proc([Body, success])), \\
&\qquad if(Count < 3, \\
&\qquad\quad ?(CountPisCount + 1) : leftTurn2(Body, CountP), \\
&\qquad\quad end\_Proc([Body, failure]) \\
&\qquad) \\
&\quad) \\
&\quad)
\end{aligned}
\tag{4.12}
$$

).

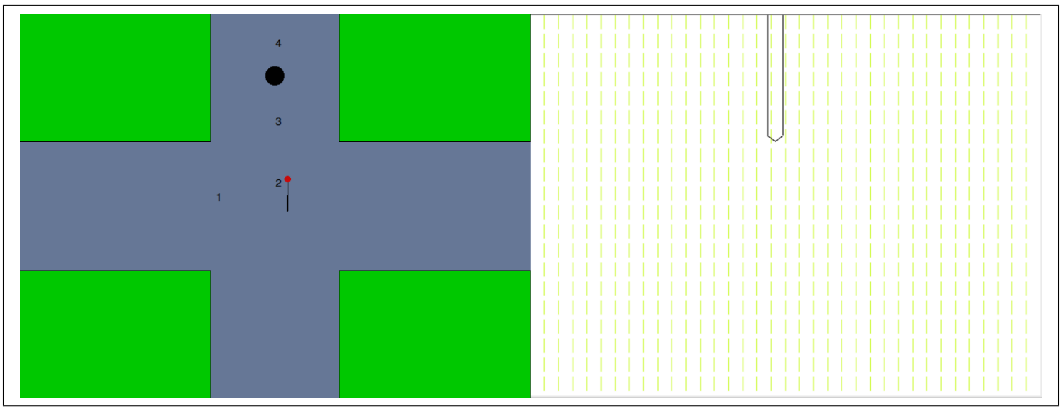

Figure 4.35: Body Starts to Match Observer's Speed

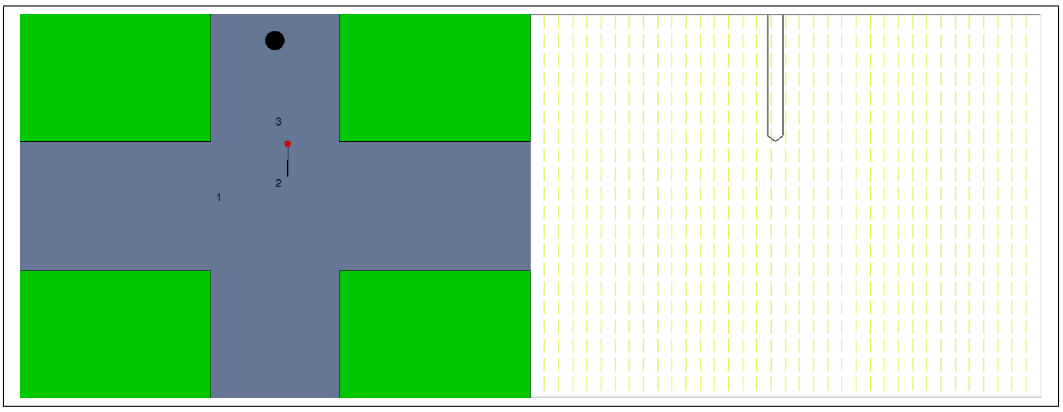

Figure 4.36: The Body's Peak is Remaining Static and Recognition Finishes.

Multiple Bodies Example

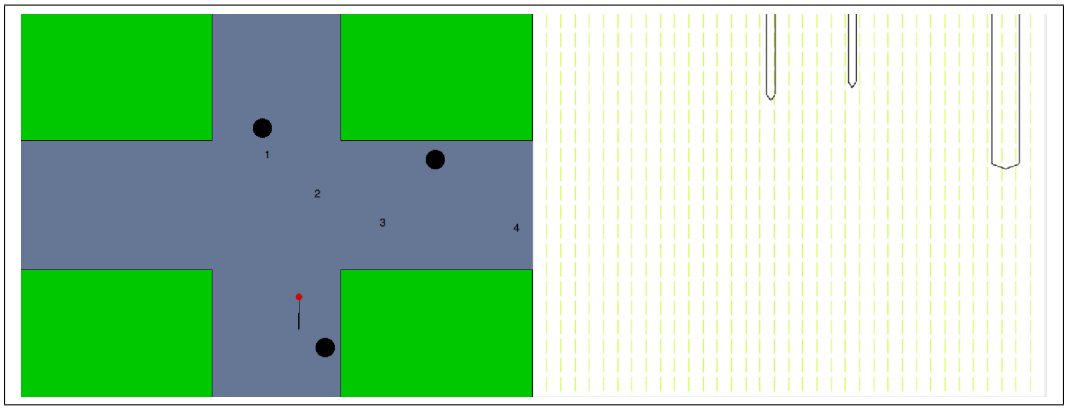

Figure 4.37: Initialization

In the initial situation we notice there are three bodies, and we assume they may be performing any of the plans in the library. The list of plans is:

$$[leftTurn(body0), leftTurn(body1), leftTurn(body2),$$
$$rightTurn(body0), rightTurn(body1), rightTurn(body2)].$$

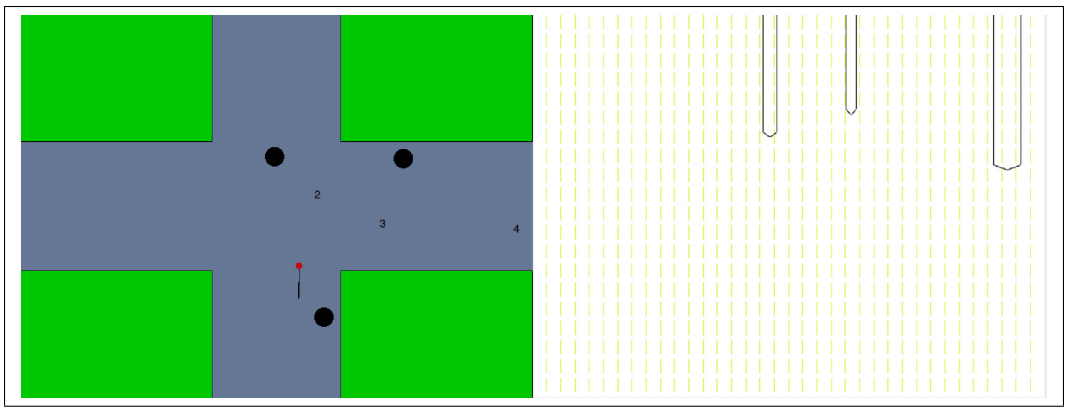

Figure 4.38: Already bodies are trimmed down to a minimal number of plans

In Figure **??** the number of possible plans is already trimmed down to a more reasonable number based on the data collected. The remaining plans are as follows:

$$[leftTurn1(body2), rightTurn1b(body0, 0), rightTurn1a\_or\_2(body1, 0)]$$

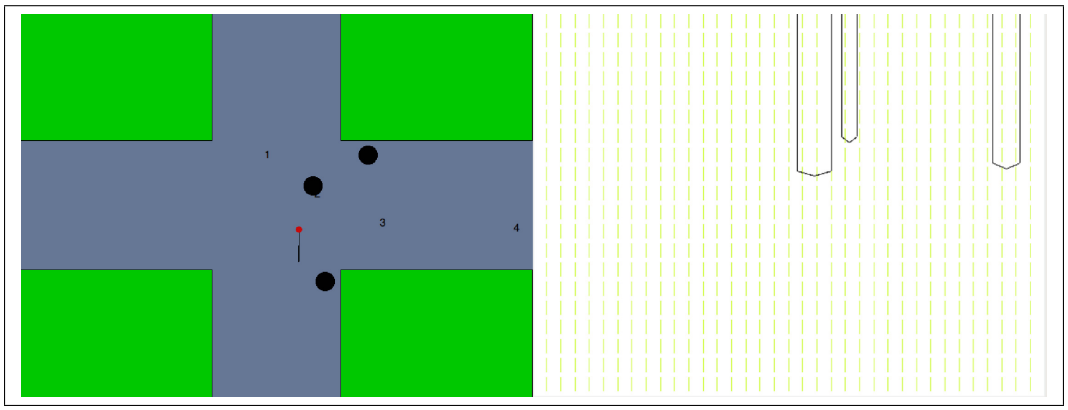

Figure 4.39: Bodies continue to complete plans

As the situation continues in Figure **??** each body has become matched to a singular plan, and as the process continues it is to see if they will complete that plan. The current plans remaining at this stage is:

$$[leftTurn1b(body2), rightTurn1b\_2(body0, 0), rightTurn1a\_or\_2(body1, 1)]$$

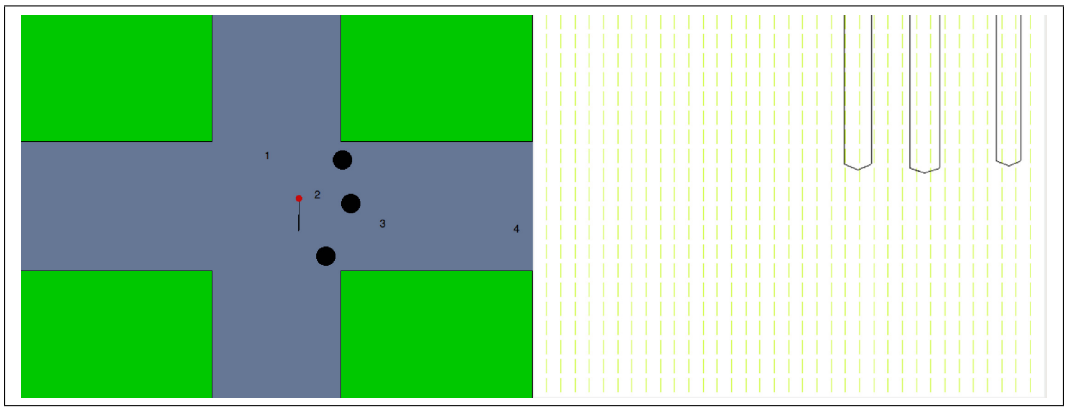

Figure 4.40: A body successfully completes a left turn

Already in Figure **??** one of the bodies has completed its plan. $body2$ has completed its left turn in front of the observer successfully. From this point on we are only concerned with the two remaining bodies :

$$[rightTurn1b\_3(body0), rightTurn1a\_or\_2(body1, 2)] \tag{4.13}$$

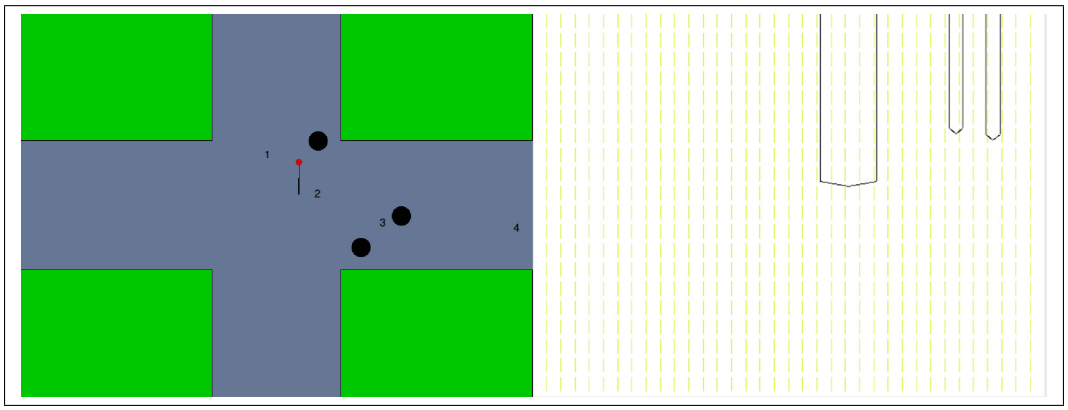

Figure 4.41: Two Bodies Remain

Similarly to the previous right turn example in Section **??**, the bodies follow a similar pattern as they are completing similar plans in Figure **??**

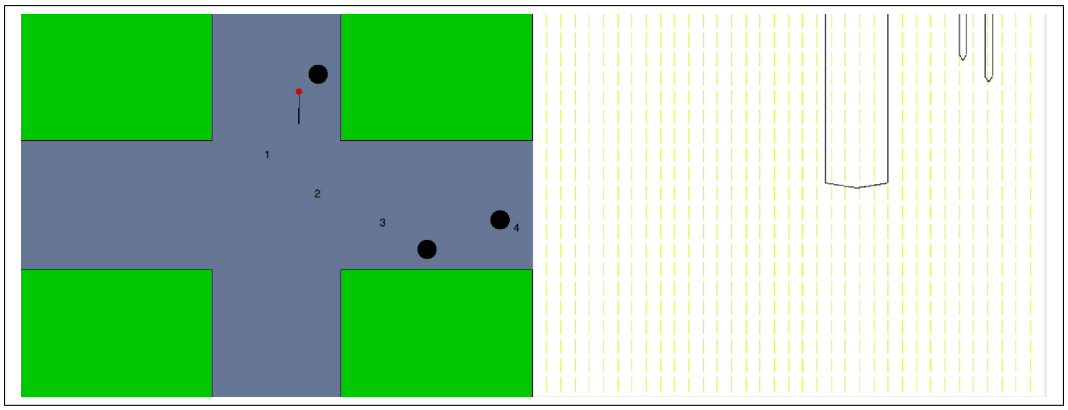

Figure 4.42: One Plan Remains

By the end all bodies are matched to a plan, as the final body remains static after merging into traffic in Figure **??** and **??**. The number of bodies does not affect the ability to recognize plans

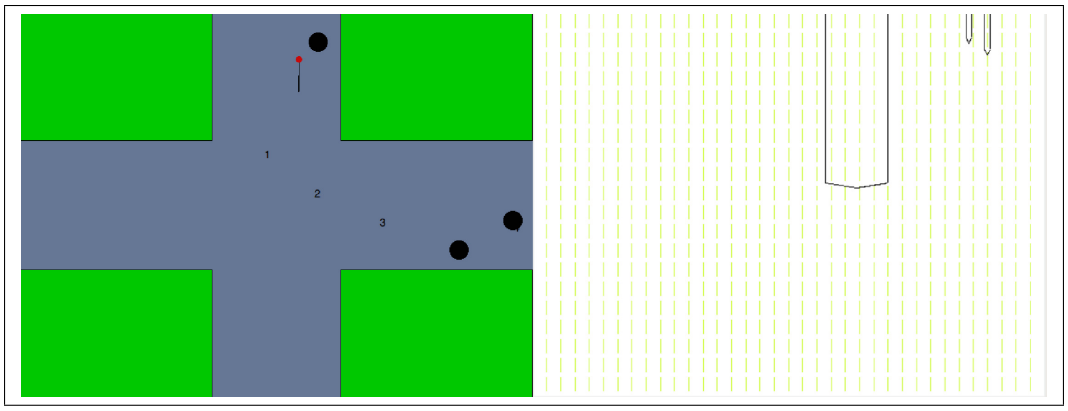

Figure 4.43: Final Plan is Recognized

as we design plans to reach a final state as soon as possible, whether it is a fail or success state. In this example we have shown that even with minor occlusion happening we can still recognize what is happening in the majority of cases.

4.3.6 Limits of Recognition

In this section we discuss the limitations of the system. In all the above examples we have shown how we can define plan recognition programs that define how a body performs a behaviour in relation to the observer. We use the relations based on dynamic depth profiles defined by [**?**, **?**]. Where this system is limited is on defining behaviours based on relations between two depth peaks. We will look at an example of two bodies approaching each other and how the ambiguity created by this behaviour does not allow us to use this definition effectively to define complex behaviours. The full definitions of the successor state axioms between two bodies can be found in Appendix **??**.

Approaching is defined as, when the angular distance between to bodies is decreasing from a previous situation to the current situation. Figure **??** shows an example of this. The angular distance, ω between the two bodies decreases between the two situations.

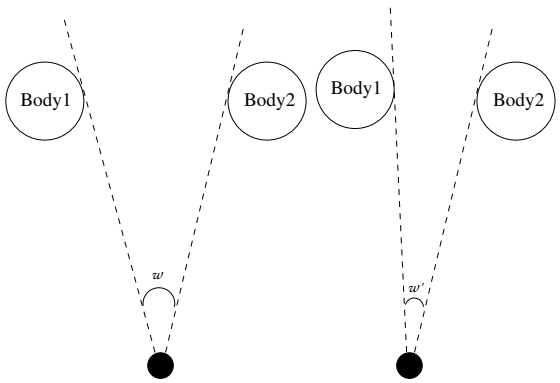

Figure 4.44: Approaching Before and After

Ambiguity happens as it is not clear what action has caused the two bodies to appear to be approaching. In the system developed here, we do not know which action has occurred, we only see the effect of that action on the fluents defined in the system. For this reason the cause of two bodies approaching is unknown, and therefore the true behaviour being performed is impossible to know. Furthermore the Figures below will support this conclusion.

Ambiguity Through Body Movement

In Figure **??** we show an example of how ambiguity is created. In this case **??**A) will represent the previous situation and **??**B) and **??**C) will represent two examples of the current situation where the bodies appear to be approaching each other. In B) the bodies appear to be approaching because Body1 is moving faster than Body2, therefore decreasing the angular distance between the two bodies. It is not possible to gather from the change in angular distance that one body is moving faster than the oher.

In C) the two bodies are both moving equally closer to each other. This causes an equal change in angular distance ω' as was witnessed in B). Because we don't have axioms to define speed or direction of travel it is impossible to tell what is causing the two bodies to be approaching each other. This ambiguity makes it difficult to define behaviours using this relation. Similar examples can be built for the other fluents defined in Appendix **??**.

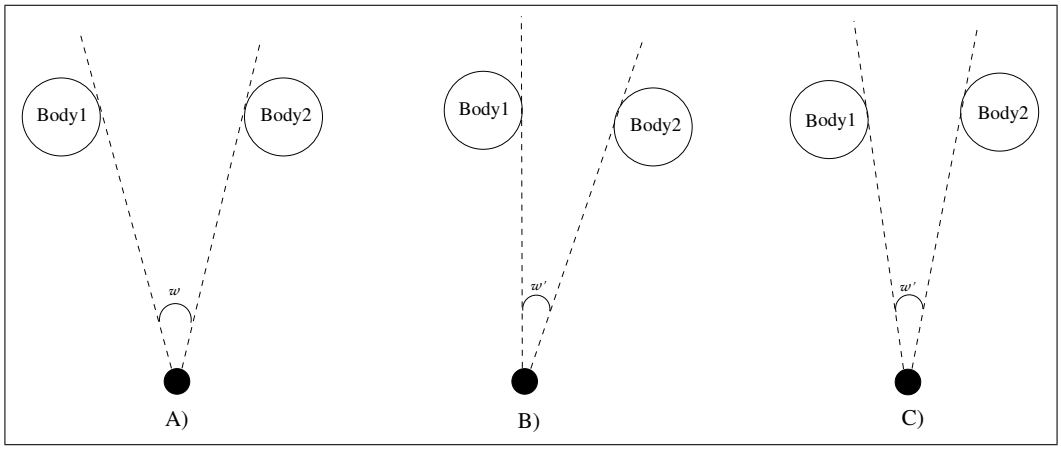

Figure 4.45: Ambiguity Through Body Movement

Important axioms such as speed, direction of travel and relative distance from the observer are not available in the current formalism. But these could be seen as future extensions that would aim to alleviate the ambiguity of these fluents. Knowing these values would allow us to define and eliminate the ambiguity that occurs in **??**B) and **??**C).

Chapter 5

Conclusion

This chapter discusses the major contributions made by this thesis and some possible future research directions.

5.1 Summary

Although planning is a popular research area, there is very little focus on plan recognition. In this thesis we formalized a plan recognition system based on FSA and realized in the Situation Calculus and Golog. The major contributions are as follows

1. We defined 8 additional successor state axioms added as an extension to [**?**] that describe relative direction from the view of an observer. We add to the original set of relations a way to reason about objects based on relative direction. These are first explained in Section **??**

2. We create a software simulator for simulating traffic scenarios as well as mimicking the sense and movement actions of an observer consisting of 9000 lines of C++ code. As well as a plan recognition system based in ConGolog that connects to the simulator consisting of 6800 lines of Prolog code.

3. As discussed in Chapter 3, we describe an informal FSA for plan recognition that receives as input boolean combinations of fluent truth values. This involves a design structure describing FSA that is put forward allowing us recognize different plan recognition programs. Also, we clearly define a structure for input to the FSA.

4. We perform a complexity analysis of the system that supports the design choices we made when designing the FSA defined in Chapter 3. The design choices were made with an effort to limit the number of transitions needed to reach a $Final$ state in the FSA as well as minimize the number of false matches possible during plan recognition..

5. We introduce a modification of the $Trans/Final$ ConGolog interpreter, changing it from a planning system to a plan recognition system. This includes changes that allow it to reason about multiple bodies and multiple plan recognition programs in real time as discussed in Chapter 4. As well as introducing incremental recognition that allows us to match partial plan recognition programs in real time.

6. Finally, we introduce a series of examples showing the capabilities of the plan recognition system developed throughout the thesis as seen in Chapter 4. In turn, this lays the groundwork for future development of plan recognition programs.

5.2 Future Work

Four of the most important directions for future work in a ConGolog plan recognition framework for motion are a larger library of plan recognition programs, addressing the limits discussed in Section **??**, progressing the database for faster computation of fluents used to solve recognition problems, and combining this system with a planning sytem.

Building a large plan library for a planning system is a difficult problem, it is no different for plan recognition. An interesting area of future research would be applying automated learning techniques that have been applied to other domains as discussed in Section **??**. These methods would allow a large plan library to be built quickly, automatically and reduce the number of human caused errors.

It would also be possible to convert between FSA and Golog code. This would allow non-computer programmers to use the techniques put forth in this thesis to create plan recognition programs also. Due to the direct 1 to 1 mapping of FSA to Golog programs this is a very possible software application of the work in this thesis.

As discussed in Section **??**, there are a number of axioms that have ambiguity. These axioms

would be useful in dealing with occlusion. In this thesis occlusion does not affect the system as long as an important event in the plan recognition program is not occluded. Further developing the limits of the system would allow plan recognition programs to be able to reason about occluded events. Axioms such as $direction\_of\_travel$, $speed$, $relative\_distance$ would help to overcome some of these problems. As well as helping to solve the problem of stop and go situations. In this thesis we do not consider cases where bodies are not moving, or are starting and stopping. These type of situations do occur in the real world and the current set of axioms would need extensions to be able to reason about these behaviours.

Next, progressing the initial database. Although in the problems encountered in this thesis, computation time and resources were not a problem, a system running for long periods of time and with large amounts of relations would find that over time, computation time would affect its usability. In the case of Situation Calculus, having a shorter situation term would greatly reduce the amount of regression that needs to be performed for every calculation of fluent values.

Finally, plan recognition can be used as a tool to improve existing planning techniques. Using plan recognition and planning system can limit the search space for what plans a robot is able to execute. Combining the plan recognition system put forth in this thesis with a planning system would allow a user to actively control a robot. A planning system using plan recognition would create a more effective planning system versus traditional planning systems that don't reason about actions of objects around it.

Appendix A

Left Turn Examples

Below is the complete Golog code for the left turn plan recognition program.

```
proc(leftTurn(Body),
   pi([LBD,D,S,L2],
      if(frontof(peakOf(Body,_,_,LBD), L2)  ,
        leftTurn1(Body),
        if(leftof(peakOf(Body,_,_,LBD), L2) v
         leftof_front(peakOf(Body,_,_,LBD), L2),
          leftTurn24(Body),
          if( rightof_front(peakOf(Body,_,_,LBD), L2),
            leftTurn3(Body),
            end_Proc([Body, failure])
          )
        )
      )
   )
).

proc(leftTurn3(Body),
   pi([LBD,D,S,L2],
      if(frontof(peakOf(Body,_,_,LBD), L2) &
```

```
        extending(peakOf(Body,D,S,LBD), L2),
        leftTurn3b(Body,0),
        end_Proc([Body, failure])
      )
    )
).

proc(leftTurn3b(Body,Count),
  pi([LBD,D,S,L2],
    if(leftof_front(peakOf(Body,_,_,LBD), L2) &
        extending(peakOf(Body,D,S,LBD), L2),
      leftTurn3c(Body),
      if(Count < 3,
        ?(CountP is Count + 1) : leftTurn3b(Body,CountP),
        end_Proc([Body, failure])
      )
    )
  )
).

proc(leftTurn3c(Body),
  pi([LBD,D,S,L2],
    if((leftof(peakOf(Body,_,_,LBD), L2) v
        leftof_back(peakOf(Body,_,_,LBD), L2))
          & shrinking(peakOf(Body,D,S,LBD), L2),
      (?(write("Final left turn 3c")) : end_Proc([Body, success])),
      end_Proc([Body, failure])
    )
  )
).
```

```
proc(leftTurn24(Body),
  pi([LBD,D,S,L2],
    if(  (frontof(peakOf(Body,_,_,LBD), L2) v
      leftof_front(peakOf(Body,_,_,LBD), L2))
     & extending(peakOf(Body,_,_,LBD),L2) ,
      leftTurn2(Body,0),
      if(leftof(peakOf(Body,_,_,LBD), L2) &
       shrinking(peakOf(Body,_,_,LBD), L2),
        leftTurn4(Body),
        end_Proc([Body, failure])
      )

    )
  )
).

proc(leftTurn2(Body,Count),
  pi([LBD,D,S,L2],
    if(frontof(peakOf(Body,_,_,LBD),L2) &
      one_peak_static(peakOf(Body,D,S,LBD),L2) ,
      (?(write("Left Turn 2")) : end_Proc([Body, success])),
      if(Count < 3,
        ?(CountP is Count + 1) : leftTurn2(Body,CountP),
        end_Proc([Body, failure])
      )
    )
  )
).
```

```
proc(leftTurn4(Body),
  pi([LBD,D,S,L2],
    if(leftof(peakOf(Body,_,_,LBD),L2) &
        shrinking(peakOf(Body,_,_,LBD),L2),
          (?(write("Left Turn 4")) : end_Proc([Body, success])),
          end_Proc([Body, failure])
    )
  )
).



proc(leftTurn1(Body),
  pi([LBD,D,S,L2],
    if( frontof(peakOf(Body,_,_,LBD), L2) &
       extending(peakOf(Body,_,_,LBD), L2),
      (
      ?(write("Its making a turn in front of you\n")) : leftTurn1b(Body)
      ),
      end_Proc([Body, failure])
    )
  )



).
proc(leftTurn1b(Body),
pi([LBD,D,S,L2],
  if( (rightof_front(peakOf(Body,_,_,LBD), L2) v
     rightof(peakOf(Body,_,_,LBD),L2) )
      & extending(peakOf(Body,_,_,LBD), L2),
    (
```

```
    ?(write("Its Completed the left turn\n")) :
        end_Proc([Body, success])
    ),
    (
    ?(write("Not left turning\n")) :
        end_Proc([Body, failure])
    )
  )
)
).
```

Appendix B

SSA Two Bodies

Below is the SSA for *approaching* used in Section **??**. For the other two body SSA please refer to the source code at `http://www.scs.ryerson.ca/~j2gross`.

Approaching

```
approaching(peakOf(BodyA,_,SizeA,LBD_A), peakOf(BodyB,_,SizeB,LBD_B),
    loc(XR,YR), do(A,S)):-
  A = sense(Profile, loc(XR,YR), T),
  member(peakOf(BodyA,_,SizeA,LBD_A), Profile),
  member(peakOf(BodyB,_,SizeB,LBD_B), Profile),
  location(robot, loc(XR, YR), S),
  ( LBD_B > LBD_A + SizeB,
    % BodyA is between the left border and BodyB
    NewDist is  LBD_B - LBD_A - SizeB ;
    LBD_A > LBD_B + SizeA,
       % BodyB is between the left border and BodyA
    NewDist is LBD_A - LBD_B - SizeA
  ),
  dist(peakOf(BodyA,_,SizeA_Old,LBD_A_Old),
  peakOf(BodyB,_,SizeB_Old,LBD_B_Old), DistBefore, loc(XR,YR), S),
  NewDist < DistBefore,
```

```
    delta(Delta), NewDist > Delta .

approaching(peakOf(BodyA, _, _, LBD_A), peakOf(BodyB, _, _ ,LBD_B),
      loc(XR,YR), do(A,S)):-
    A = endMove(robot, loc(X,Y), loc(XR,YR), T),
    location(BodyA, loc(XA,YA), S), radius(BodyA, RA),
    location(BodyB, loc(XB,YB), S), radius(BodyB, RB),
    location(robot,loc(XR,YR), S),
    facing(Theta, loc(XR,YR),S),
    angleLBD(loc(XA,YA), loc(XB,YB),
         loc(X,Y), RA, RB, Theta, DistBefore),
    angleLBD(loc(XA,YA), loc(XB,YB),
         loc(XR,YR), RA, RB, Theta, DistNow),
    DistNow < DistBefore,
    delta(Delta), DistNow > Delta .

%BodyA moving
approaching(peakOf(BodyA, _, _, LBD_A), peakOf(BodyB, _, _ , LBD_B),
      loc(XR,YR), do(A,S)):-
    A = endMove(BodyA, loc(XA1,YA1), loc(XA2,YA2), T),
    location(robot, loc(XR,YR), S),
    location(BodyA, loc(XA1,YA1), S), radius(BodyA, RA),
    location(BodyB, loc(XB,YB), S), radius(BodyB, RB),
    facing(Theta, loc(XR,YR),S),
    angleLBD( loc(XA1,YA1), loc(XB,YB),
      loc(XR, YR), RA,RB, Theta, DistBefore),
    angleLBD( loc(XA2,YA2), loc(XB,YB),
      loc(XR, YR), RA,RB, Theta, DistNow),
    DistNow < DistBefore,
    delta(Delta), DistNow > Delta .
```

```
%BodyB moving
approaching(peakOf(BodyA, _, _, LBD_A), peakOf(BodyB, _, _ ,LBD_B),
    loc(XR,YR), do(A,S)):-
  A = endMove(BodyB, loc(XB1,YB1), loc(XB2,YB2), T),
  location(robot, loc(XR,YR), S),
  location(BodyA, loc(XA,YA), S), radius(BodyA, RA),
  location(BodyB, loc(XB1,YB1), S), radius(BodyB, RB),
  facing(Theta, loc(XR,YR),S),
  angleLBD( loc(XA,YA), loc(XB1,YB1),
    loc(XR, YR), RA,RB, Theta, DistBefore),
  angleLBD( loc(XA,YA), loc(XB2,YB2),
    loc(XR, YR), RA,RB, Theta, DistNow),
  DistNow < DistBefore,
  delta(Delta), DistNow > Delta .

approaching(peakOf(BodyA,_,_,LBD_A),peakOf(BodyB,_,_,LBD_B),
    loc(XR,YR),do(A,S)) :-
  approaching(peakOf(BodyA,_,SizeA,LBD_A), peakOf(BodyB,_,SizeB,LBD_B),
    loc(XR,YR),S),
  location(robot, loc(XR,YR), S),
  location(BodyA, loc(XA,YA), S),
  location(BodyB, loc(XB,YB), S),
  radius(BodyA, RA), radius(BodyB, RB),
  delta(Delta),
  not ( A = sense(Profile, loc(XR,YR), T),
    member(peakOf(BodyA, _, SA, DA), Profile),
    member(peakOf(BodyB, _, SB, DB), Profile),
    ( DB > DA + SB,
      % BodyA is between the left border and BodyB
```

```
      DistNow is  DB - DA - SB ;
      DA > DB + SA,
          % BodyB is between the left border and BodyA
      DistNow is DA - DB - SA
    ),
    dist(peakOf(BodyA,_,_,LBD_A), peakOf(BodyB,_,_,LBD_B),
     DistBefore, loc(XR,YR),S),
    ( DistNow >= DistBefore ; DistNow < Delta )
),
not ( A = endMove(BodyA, loc(XA1,YA1), loc(XA2,YA2) ,T),
    facing(Theta, loc(XR,YR),S),
    angleLBD( loc(XA2,YA2), loc(XB,YB),
        loc(XR,YR), RA,RB, Theta,DistNow),
    angleLBD( loc(XA1,YA1), loc(XB,YB),
        loc(XR,YR), RA,RB, Theta,DistBefore),
    ( DistNow >= DistBefore ; DistNow < Delta)
),
not ( A = endMove(BodyB, loc(XB1,YB1), loc(XB2,YB2) ,T),
    facing(Theta, loc(XR,YR),S),
    angleLBD( loc(XA,YA), loc(XB2,YB2),
        loc(XR,YR), RA,RB,Theta, DistNow),
    angleLBD( loc(XA,YA), loc(XB1,YB1),
      loc(XR,YR), RA,RB,Theta, DistBefore),
    ( DistNow >= DistBefore ; DistNow < Delta)
),
not (   A = endMove(robot, loc(XR,YR), loc(Xnew,Ynew) ,T),
    facing(Theta, loc(Xnew,Ynew),S),
    angleLBD( loc(XA,YA), loc(XB,YB),
        loc(Xnew,Ynew), RA,RB,Theta, DistNow),
    angleLBD( loc(XA,YA), loc(XB,YB),
```

```
        loc(XR,YR), RA,RB, Theta,DistBefore),
    ( DistNow >= DistBefore ; DistNow < Delta)
).
```

Appendix C

Right Turn Golog code

Below is the complete Golog code for the right turn plan recognition program.

```
/*
Recognizing Right Turns of other vehicles.
Two types:
  1) Turn off of a straight road
    1a)-car starts in front of observer
      and during observer passes car in process of
      right turn (observer in another lane)
    1b)-car starts behind observer
      (same lane)
  2) Right turn merging onto a road
    -observer is facing forwards as it is happening
    (observer doesn't need to slow down as in another lane)

*/

proc(rightTurn(Body),
/*
First determine which of the 3 types of right turns we are
recognizing based on object location
```

```
*/
   ?(write("Right Turn\n")):
  %printSit:
   pi([LBD,D,S,L2],
     if( backof(peakOf(Body,_,_,LBD), L2) v
       rightof_back(peakOf(Body,_,_,LBD), L2),
       %then it could be making this right turn
       (
         ?(write("1B right Turn\n")) :
         rightTurn1b(Body,0)

       ),
       %else it is a different type of right turn
       (if( rightof_front(peakOf(Body,_,_,LBD), L2) ,
         %then
         (
         ?(write("1A2 right Turn\n")) :
         rightTurn1a_or_2(Body,0)
         ),
         %else it is starting to overtake right
         (
         ?(write("Not right Turning\n")) :
         end_Proc([Body, failure])
         )
       ))
     )
   )
).

proc(rightTurn1a_or_2(Body,Count),
```

```
/*
   1a-car starts in front of observer
  and during observer passes car in process of
  right turn (observer in another lane)

  or
  2 Right turn merging onto a road
  -observer is facing forwards as it is happening
  (observer doesn't need to slow down as in another lane)
*/
  ?(write("Right Turn 1A2\n")):
  %printSit:
  pi([LBD,D,S,L2],

  if( rightof(peakOf(Body,_,_,LBD), L2) &
   extending(peakOf(Body,D,S,LBD),L2) & Count < 2,
    %then it could be making this right turn
    (
      ?(write("1A right Turn\n")) :
      rightTurn1a(Body,0)
    ),
    %else
    (
      if( Count > 1 &
        rightof_front(peakOf(Body,_,_,LBD), L2) &
            extending(peakOf(Body,D,S,LBD),L2)   ,
        % then
        (
        ?(write("2 right Turn\n") & write(Body) & nl) :
        rightTurn2(Body,0)
```

```
        ),
        (
          ?(CountN is Count + 1) :
            if(CountN < 6,
            ( %give it another chance
            rightTurn1a_or_2(Body,CountN)
          ),
          %else
          (
           end_Proc([Body, failure])
          )
          )
        )

      )
    )
  )
  )

).


proc(rightTurn2(Body,Count),
  ?(write("Right Turn 2\n")):
  printSit:
  pi([LBD,D,S,L2],

  if( rightof(peakOf(Body,_,_,LBD), L2) &
      one_peak_static(peakOf(Body,D,S,LBD),L2) ,
    %then it could be making this right turn
```

```
(
   ?(write("Completed right turn (joined traffic)\n")) :
   end_Proc([Body, success])
),
%else
(
   if( one_peak_static(peakOf(Body,D,S,LBD),L2) &
       rightof_front(peakOf(Body,_,_,LBD), L2) ,
     % then
     (
       ?(write("Completed right turn (joined traffic)\n")) :
       end_Proc([Body, success])
     ),
     (

       %else
       (
         if(rightof_front(peakOf(Body,_,_,LBD),L2)
           & Count < 2,
         (
           ?(write("2 right Turn\n")):
           ?(CountN is Count + 1) :
           rightTurn2(Body,CountN)
         ), %else
         (
         ?(write("failureed Right Turn 2\n")) :
         end_Proc([Body, failure])
         ))
       )
```

```
        )

      )
    )
  )
  )
).

proc(rightTurn1a(Body,Count),
/*
   1a-car starts in front of observer
  and during observer passes car in process of
  right turn (observer in another lane)


*/
  ?(write("Right Turn 1A\n")):
  %printSit:
  pi([LBD,D,S,L2],

  if( (rightof(peakOf(Body,_,_,LBD), L2) v
       rightof_back(peakOf(Body,_,_,LBD), L2))
      & shrinking(peakOf(Body,D,S,LBD),L2)  ,
    %then it could be making this right turn
    (
      ?(write("1A2 right Turn\n")) :
      rightTurn1a2(Body,0)
    ),
    %else
    (
```

```
        if( rightof(peakOf(Body,_,_,LBD), L2)  &
            extending(peakOf(Body,D,S,LBD),L2)
             & count < 2,
          % then
          (
            ?(write("1A right Turn\n")):
            ?(CountN is Count + 1) :
            rightTurn1a(Body,CountN)
          ),
          (

            %else
            (
             end_Proc([Body, failure])
            )

          )

        )
      )
    )
    )
).

proc(rightTurn1a2(Body,Count),
/*
   1a-car starts in front of observer
  and during observer passes car in process of
  right turn (observer in another lane)
```

```
*/
  ?(write("Right Turn 1A\n")):
  %printSit:
  pi([LBD,D,S,L2],

  if( rightof_back(peakOf(Body,_,_,LBD), L2)  &
      shrinking(peakOf(Body,D,S,LBD),L2) ,
    %then it could be making this right turn
    (
      ?(write("Body Made a Right Turn\n")) :
      end_Proc([Body, success])
    ),
    %else
    (
      if( count < 2,
      % then
        (
        ?(write("1A2 right Turn\n")):
        ?(CountN is Count + 1) :
        rightTurn1a2(Body,CountN)
        ),
        %else
        (
         end_Proc([Body, failure])
        )
      )

    )
  )
```

```
  )

).

proc(rightTurn1b(Body,Count),
/*
  1b-car starts behind observer
  (same lane)
*/
  ?(write("Right Turn 1b\n")):
  pi([LBD,Size,Depth,L2,L3],
    if( (backof(peakOf(Body,_,_,LBD), L2) v
        rightof_back(peakOf(Body,_,_,LBD), L2))
       & shrinking(peakOf(Body,Depth,Size,LBD),L2)   ,
      %then it could possible be overtaking at some point
      (
        ?(write("1B right Turn\n")) :

        rightTurn1b_2(Body,0)

      ),
      %else
      (
        %allows one more chance to slow down and turn
        %then cut off with count feature

          ?(CountN is Count + 1) :
          if(CountN < 2,
          (
          rightTurn1b(Body,CountN)
```

```
        ),
        %else
        (
          end_Proc([Body, failure])
        )
        )
      )
      )
    )

).

proc(rightTurn1b_2(Body,Count),
  ?(write("Right_Turn_1b2\n")) :

  pi([LBD,Depth,Size,L2],
    if( (rightof_back(peakOf(Body,_,_,LBD), L2) &
      shrinking(peakOf(Body,Depth,Size,LBD), L2) ) ,
      %then it could possible be overtaking at some point
      (
        ?(write("1B2_right_Turn\n")) :
        rightTurn1b_3(Body)

      ),
      %else
      (
        %allow one more chance to back of otherwise should
        % and be turned
        %then cut off with count feature
```

```
            ?(CountN is Count + 1) :
            if(CountN < 2,
            (
            rightTurn1b_2(Body,CountN)
          ),
          %else
          (
           end_Proc([Body, failure])
          )
          )
        )
        )
    )

).

proc(rightTurn1b_3(Body),
  ?(write("Right Turn 1b3\n")) :

  pi([LBD, Depth, Size, L2],
    if( (rightof_back(peakOf(Body,_,_,LBD), L2) &
        shrinking(peakOf(Body,Depth,Size,LBD), L2)
        ) ,
      %then it could possible be overtaking at some point
      (
        ?(write("Body Made a Right Turn\n")) :
        end_Proc([Body, success])

      ),
      %else
```

```
        (
          %we assume after a turn it starts to speed
          %up and this is that check
          end_Proc([Body, failure])

        )
        )
    )

).

proc(rightTurn_final(Body),
  ?(write("Made a Right Turn\n"))
).
```

Appendix D

Software Instructions

The purpose of this appendix is to explain the use of the software as well as installation of the platform. The software can be found at the following url: `http://www.scs.ryerson.ca/~j2gross`

The software is made up of 2 main programs that run simultaneously. An environment simulator which simulates the movement of a robot and any number of bodies in 2D space and a Golog(Plan Recognition) framework. Paths of all objects in the simulator are predetermined at time of execution. There is no feedback from Golog(Plan Recognition) that allows the robot to change its path based on what plans are recognized. Although this is not implemented it is the suggestion of the author that this is where this type of plan recognition could be used in an overall scheme of planning.

The idea of predetermined plans is that all objects in the environment are 'human controlled' (in the case of vehicles, have drivers), and the Plan Recognition software provides the ability to recognize the projected path or behaviour being executed by bodies around a robot (whether controlled through software via a planning program, or human controlled). In recognizing these behaviours we would be able to adapt the robot's plan if it is software controlled, or a human could adapt their route based on the information retrieved from the plan recognition software.

The Golog interpreter runs at the same time as the simulator and is able to access data from the simulator in the form of an initial situation, current location of the robot, and sensing data (depth profiles). It uses this information to recognize behaviours of bodies in the simulator, using only information that would be available to the robot, whether human or software controlled. It is assumed that on the robot there is some form on sensing equipment that would provide this

information.

Simulator Instructions

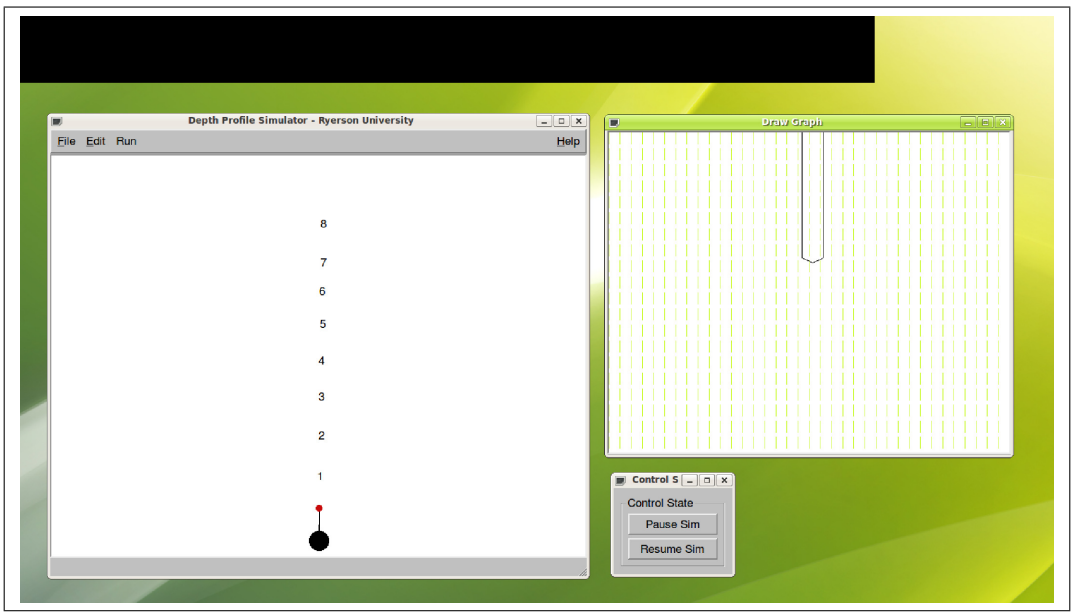

Figure 1: Simulator Screenshot

The environment simulator, Figure **??**, is a GUI that was developed to quickly create the 2D simulations of multiple bodies moving around a robot.

It contains the ability to **Load**, and **Save** created simulations in the **File Menu**.

Every simulation, in order to run, must contain an observer that is represented by a red dot. This can be added to the simulation from the **Edit Menu**→**Add Observer**. The observer has been referred to previously as the robot. At all times during a simulation sensing data is taken from the view of the observer. Additional data is available at all times to outside programs, such as rotation of the observer around its axis and its current location.

Although the simulation can be ran without a body in the simulation it is not really a productive task as no data is really obtained from these simulations. A body can be added to the simulation through the **Edit Menu**→**Add Circle**. Since in this work we are considering all bodies as circular objects, the term circle is used often in the C++ code of the simulator.

After adding bodies and a robot to the simulator it is possible to give them paths through the

idea of a waypoint (WPT). When an object is given a WPT, upon start of the simulation, it will move to the first WPT created in a straight line. Once it has reached the first WPT it will turn and move to the second WPT that was created in a straight line, and continue this motion until it has reached all WPT's that have been created to define its path. To add WPT's to an object, that object must first be selected. An object is selected when it is left-clicked. After an object is selected it can be moved around by using the left mouse button to drag the object around the screen. A WPT can be created by using the middle-mouse button. When clicked a WPT will be created at the cursor's location. By selecting different objects, you can see the WPT's you created and also create more WPT's for that object.

It is also possible to edit some conditions about each object created and the WPT's associated with that object. This is done by right clicking on the object of interest. This will open up a new edit window as seen in Figure **??**.

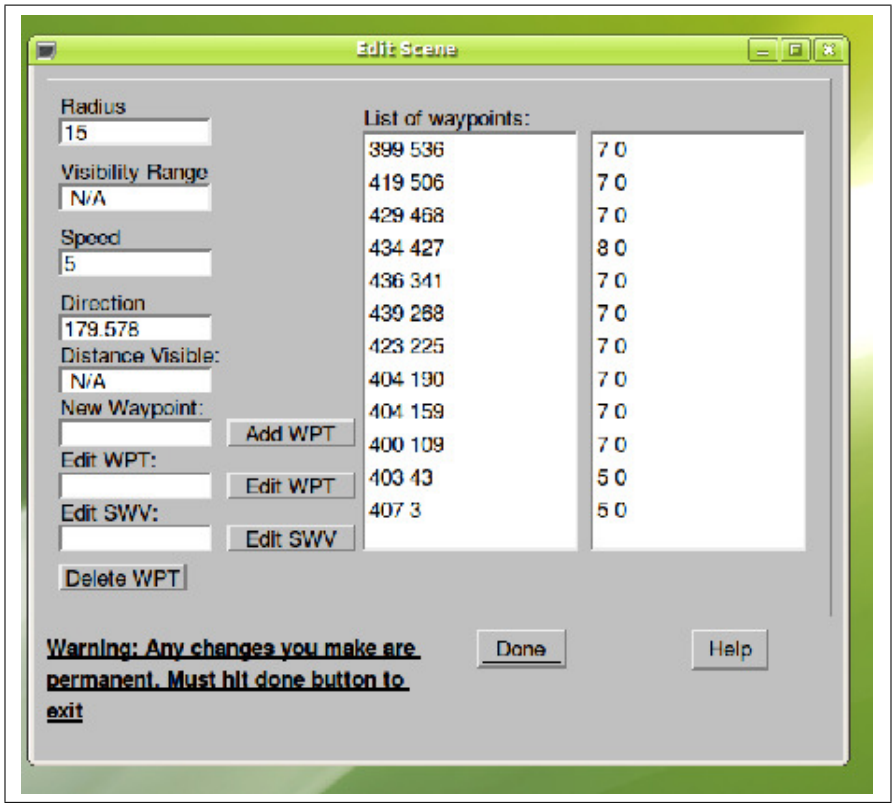

Figure 2: Edit Screen

Editable variables for bodies include: Radius, Speed, and Facing Direction. For the ob-

server/robot it is also possible to edit the Visibility Range (Field of View of the robot's camera) and Distance Visible(How far the robot can see).

If a WPT was added but the location is not exactly where it was expected it can be changed in this window. First select the WPT in the list of WPTs. Then in the Edit WPT text box, enter the X,Y coordinates of the edited WPT in the format "X Y", and push the button Edit WPT. This will change it in the list. It is also possible to add WPT using the New WPT text box. To selete a WPT select the WPT and click the button Delete WPT.

The final variable is the Edit SWV text box. SWV stands for Speed and Wait variables. This means that for each WPT in the left list, there is a corresponding change in the speed of the body and how long it will wait at that WPT in the right list. This can give the allusion of starting and stopping, as well as acceleration. To edit these values follow the same procedure as editing WPTs, except use the format for speed and wait times: Speed WaitTime. A Wait Time informs the body during the simulation to wait at the corresponding WPT for X number of cycles of the simulator. It is important to remember to click the Done button instead of just closing the window in order for the information to be saved to the simulator.

Back to the main window, there are three more menu items from the **Run Menu** that are important to a user. **Run Menu**→**Speed**, allows the user to speed up or slow down the simulation. This increases how far a body will move in one cycle if the simulation is sped up. If it is ran slower, then the body will not move as far each cycle.

Run Menu→**Stop Simulation**, will stop the simulation and restore all objects to their initial position. Finally **Run Menu**→**Full Simulation**, will start the simulation and will continue to run until all objects have reached their final WPT's. If an object reaches its final WPT before all other objects have reached theirs', it will wait there until every object has reached its final destination.

It should be noted that the simulation will not start, until the Golog agent has connected to the simulator. This will be discussed in more detail later. But at least to have a simulation run the command 'initialize' must be executed in Eclipse Prolog, or a similar Prolog environment that has loaded the plan recognition system.

This provides all the basics for creating a simulation to run plan recognition.

Golog Instructions

The Golog framework was built using the Prolog programming language. It requires a Prolog environment such as Eclipse Prolog to run in. The framework uses a set of C functions to connect through sockets to the C++ simulator to gather data. These C functions need to be compiled on the system, as they are called from the Prolog environment to connect to the simulator.

Once the software has been compiled into the Prolog environment by compiling the file "main.ecl". To call the Golog plan recognition system the command is **runMulti([List of Bodies], [List of Plan Recognition Programs]).**

Before using the command **runMulti**, the simulation must be set to run first. In the simulator using the command from the menu, **Run Menu→Full Simulation**, will start the simulation and wait for the Golog system to connect. Then using the **runMulti** command, will start the simulation and recognition system. The two systems will exchange data and return the results as the data is processed.

Simulator Design

All communication between the two programs is done using sockets. We use the idea of a client-server architecture to achieve this communication. The simulator acts as the server in this case. It listens for requests of data from the plan recognition software. When a request is received it will serve up the information that is asked for. We use C++ server sockets to achieve this effect. On the plan recognition side we use C++ client sockets to send requests to the simulator for data. This basic implementation allows for two way communication between the plan recognition software and the simulator. The wrapper classes used in these programs for C++ server/client sockets make it easy for anyone to implement a client/server architecture in their program.

Installation

The software relies on a few free libraries. These are CGAL, [**?**], and QT, [**?**]. The two libraries provide computational geometry libraries and GUI libraries respectively. Installation instructions for these to libraries can be found included with their respective packages.

The software built for this thesis consists of two separate programs. The simulator includes a

Makefile to compile the executable for the system. This software has been tested on an Ubuntu linux system. But the libraries used are cross platform, so it should be able to run on other systems without major changes needed. The Prolog portion of this program was built for the Eclipse Prolog interpreter. Before the main Eclipse Prolog file is loaded in to the Eclipse Prolog interpreter, the C functions for talking to the simulator need to be compiled first. A Makefile is included to compile the related C functions.

Licensing

All software is released under the GPL license. We hope that any derived work from this software, in part or in whole, will follow a similar licensing agreement to keep these ideas open and free for everyone.

Bibliography

[1] P. Santos, *Spatial reasoning and abductive interpretation of sensor data obtained by a mobile robot in a dynamic environment*. Ph.D. Thesis, Imperial, London, 2003.

[2] A. Goultiaeva, "Incremental plan recognition in an agent programming framework," in *Cognitive Robotics Workshop*, pp. 83–90, 2006.

[3] A. Gabaldon, "Programming hierarchical task networks in the situation calculus," in *IN AIPS02 WORKSHOP ON ON-LINE PLANNING AND SCHEDULING*, 2002.

[4] H. Nagel, "Steps towards a cognitive vision system," *Artificial Intelligence Magazine*, vol. 25(2), Summer 2004.

[5] H. Nagel and R. Gerber, "Representation of occurrences for road vehicle traffic," *Artificial Intelligence Journa,*, vol. 172(4-5), March 2008.

[6] A. Miene, A. D. Lattner, U. Visser, and O. Herzog, "Dynamic-preserving qualitative motion description for intelligent vehicles," in *In Proceedings of the IEEE Intelligent Vehicles Symposium (IV 04)*, pp. 642–646, 2004.

[7] A. Lattner, I. Timm, M. Lorenz, and O. Herzog, "Representation of occurrence for road vehicle traffic," in *IEEE International Conference on Integration of Knowledge Intensive Multi-Agent Systems*, pp. 191–196, 2005.

[8] J. D. Gehrke, A. D. Lattner, and O. Herzog, "Qualitative mapping of sensory data for intelligent vehicles," in *Workshop on Agents in Real-Time and Dynamic Environments at the 19th International Joint Conference on Artificial Intelligence*, pp. 51–60, 2005.

[9] A. D. Lattner, J. D. Gehrke, I. J. Timm, and O. Herzog, "A knowledge-based approach to behavior decision in intelligent vehicles," in *IEEE Intelligent Vehicles Symposium, Las Vegas*, pp. 466–471, 2005.

[10] J. Fernyhough, A. G. Cohn, and D. C. Hogg, "Constructing qualitative event models automatically from video input," *Image and Vision Computing*, vol. 18, pp. 81–103, 2000.

[11] M. Sridhar, A. G. Cohn, and D. C. Hogg, "Learning functional object-categories from a relational spatio-temporal representation," in *Proceeding of the 2008 conference on ECAI 2008: 18th European Conference on Artificial Intelligence*, (Amsterdam, The Netherlands, The Netherlands), pp. 606–610, IOS Press, 2008.

[12] N. Van de Weghe, A. G. Cohn, P. D. Maeyer, and F. Witlox, "Representing moving objects in computer-based expert systems: the overtake event example," *Expert Syst. Appl.*, vol. 29, pp. 977–983, November 2005.

[13] M. Soutchanski and P. Santos, "Reasoning about dynamic depth profiles," in *Proceeding of the 2008 conference on ECAI 2008*, (Amsterdam, The Netherlands, The Netherlands), pp. 30–34, IOS Press, 2008.

[14] P. Santos, "Reasoning about depth and motion from an observer's viewpoint," *Spatial Cognition and Computation*, vol. 7(2), pp. 133–178, 2007.

[15] M. dos Santos, R. de Brito, H.-H. Park, and P. Santos, "Logic-based interpretation of geometrically observable changes occurring in dynamic scenes," *Applied Intelligence*, vol. 31, pp. 161–179, 2009. 10.1007/s10489-008-0120-4.

[16] M. dos Santos and P. Santos, "A path semantics for image sequence interpretation," in *VII SBAI/II IEEE-Latin American Robotics Symposium*, 2005.

[17] D.-T. Lee, *Proximity and reachability in the plane.* PhD thesis, Champaign, IL, USA, 1978. AAI7913526.

[18] J. Gross, "Cps040 undergraduate thesis," 2008.

[19] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf, *Computational Geometry, Algorithms and Applications Second Revised Edition*. Springer, 2000.

[20] J. McCarthy, "Situations, actions and causal laws," tech. rep., Stanford University, 1963.

[21] R. Reiter, *Knowlege in action: logical foundations for specifying and implementing dynamical systems*. Cambridge, MA, USA: MIT Press, 2001.

[22] H. J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. B. Scherl, "Golog: A logic programming language for dynamic domains," *Journal of Logic Programming*, vol. 31, 1997.

[23] G. de Giacomo, Y. Lespérance, and H. J. Levesque, "Congolog, a concurrent programming language based on the situation calculus," *Artif. Intell.*, vol. 121, no. 1-2, pp. 109–169, 2000.

[24] D. V. Pynadath and M. P. Wellman, "Accounting for context in plan recognition, with application to traffic monitoring," in *In Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, pp. 472–481, Morgan Kaufmann, 1995.

[25] "Computational geometry algorithms library, v3.3.1, http://www.cgal.org, last checked : April 8, 2011."

[26] "Qt trolltech, v3.3.8b, http://trolltech.com/products/qt, last checked : April 8, 2011."