A Situation Calculus Based Approach For Deliberative Control In Dynamic Environments

Mehul Bhatt
La Trobe University, Australia
mbhatt@cs.latrobe.edu.au

Supervisor: Dr. Richard Hall
rhall@cs.latrobe.edu.au

Abstract

Decision making in a realistic environment necessitates a formal representation of its dynamics, a specification of actions and their effects. Existing reactive approaches work in a hardwired stimulus-response manner thereby not utilising complex *goal oriented planning*. Reasoning about action & change, excluded from reactive systems, instead relies on a formal approach to represent knowledge pertaining to the *causal laws* of the domain. Such knowledge is imperative for goal oriented planning in even the most simplest of domains. The aim of this project is to overcome the limitations of a reactive system through the use of a high-level deliberative behavioral approach. The control mechanism is based on *situation calculus*, which is a first order language specifically designed for representing dynamically changing worlds. Our implementation involves writing programs in the high-level cognitive robotics language *'Indigolog'* to control agents in a simulated soccer environment. Within the context of Indigolog, we emphasize the suitability of high-level deliberative control, based on a sophisticated *logic of action*, in *dynamic environment* such as the soccer domain.

Acknowledgments

I would like to thank my thesis supervisor, Dr. Richard Hall, for his constant feedback throughout the course of this thesis. His ideas, especially concerning research methodology, have had an immense effect on my work. I am thankful to Dr. Maurice Pagnucco (KSG@UNSW), whose feedback and suggestions have heavily influenced this work. Also, I would like to thank Dr. Wenny Rahayu for employing me as a research assistant during the course of this work.

Contents

1 Introduction

A significant effort in Artificial Intelligence has been devoted to studying the representation of domain knowledge pertaining to dynamic environments. Research in this regard mainly focuses on developing formalisms for representing knowledge about such systems and reasoning about it, generally referred to as ***Reasoning about Action and Change*** **(RAC).** Indeed, the real world being a dynamic place, all attempts to model any but its simplest features must take ***change*** seriously. Certain fundamental issues (that make up the core representational problems) from a reasoning viewpoint must be addressed by every formalism. For example, consider the following problems:

- **The *Frame* problem**: How do we reason about those aspects of the state that remain unchanged as a result of performing an action ? If we had a number of actions to be performed in sequence, we would have quite a number of conditions to write down that certain actions do not change the state in some way. Precisely, with m actions and n values (representing the state), we might have to write down mn such conditions. This problem, first addressed in [1], is considered to be fundamental from an RAC viewpoint.

- **The *Ramification* problem**: The ramification problem names the challenge to accommodate actions whose execution causes indirect effects [2]. These effects, not formally accounted for in the respective action specification, are consequences of general laws describing dependencies between components of the world description.

- **The *Qualification* problem**: In formal theories of reasoning about actions, the qualification problem denotes the problem to account for the many conditions, which albeit unlikely to occur in the general case, may prevent the successful execution of an action [3]. Note that given a certain action representation formalism, these preconditions will not be a part of formal action precondition list.

Many approaches based on mathematically rigorous formalisms for reasoning about action and change have been proposed and analyzed. Addressing problems, in their full generality, pertaining to dynamic domains is a non trivial task; Apart from addressing the fundamental problems, a general account of dynamical systems must accommodate the following range of phenomena [4]:

- The causal laws relating actions to their effects.
- The conditions under which actions can be performed.
- Exogenous events.
- Probabilistic action occurrences and effects.
- Utility theory: Deciding what to do and when to do it.

- Complex actions and procedures.
- Discrete and continuous time.
- Concurrency.
- Continuous processes.
- Unreliable sensors and effectors.
- Planning a course of actions.
- Monitoring the execution of a course of actions and recognizing and recovering from failed actions.

Recently, the field of *Cognitive robotics* has emerged with the aim of providing practical tools for reasoning and/or planning in the real world. The broader research agenda within the community is being centered around issues pertaining to action and their effects. The field of cognitive robotics has, as its long term objectives, the provision of a uniform theoretical and implementation framework for robotic or software agents that reason, act and perceive in changing, incompletely known, unpredictable environments. The cognitive robotics idea of a intelligent system, and the metaphor of intelligent individuals that are situated into dynamic environments and that can interact with each other, updating their mutual beliefs, is being regarded as the new model of symbolic cognition [5].

As stated above, most of the research in cognitive robotics aims at unifying theoretical and implementation frameworks for intelligent agent design. One approach to meet this end involves use of logic based KRR formalisms as the underlying theory in order to develop high-level agent languages. For instance, *Situation calculus*, which is a sophisticated logic of action, is the underlying KRR formalism for various high-level action languages such as GOLOG and its extensions conGolog, Indigolog etc. It is a first order language specifically designed for representing dynamically changing worlds. In addition to allowing for the representation of actions and their effects, it also facilitates a parsimonious solution to the frame problem through the specification of the so called *frame axioms* [6] . These are axioms that specify the action *invariants* of the domain, namely, those aspects of the state that remain unchanged as a result of performing an action. The point being made here is that such a merger between theory and practice is highly advantageous as the resulting agent framework tends to be based on mathematically rigorous principles rather than an ad-hoc implementation effort.

The rest of the introductory section is organised as follows: We briefly present some theoretical issues pertaining to agents in sub-section 1.1. Specifically, we look at the two popular models of agent cognition, namely the possible world model and the belief-desire-intention model. In sub-section 1.2, we cover issues pertaining to construction of agent systems that satisfy their respective theoretical properties. We slowly intend to move the discussion from theory towards practice. However, in this introductory section, we refrain from getting too much into practical details. These have been covered in greater detail in other parts of the thesis. Some of the research questions addressed in this thesis have been pointed out in sub-section 1.3. A complete organisation of the rest of the thesis has been presented in sub-section 1.4.

1.1 Agent Cognition Models

An agent cognition model can be defined as a formal approach for the specification of the notion of agency. Agent theorists develop formalisms for representing the properties of agents, and using these formalisms, develop models that capture desirable properties of agents. Agent theorists address questions such as: How are we to conceptualize agents ? What properties should agents have and How are we to formally represent and reason about these properties.

1.1.1 Possible Worlds Model

The possible worlds model for logics of knowledge and belief is based on the that an agents belief could be characterized as a set of possible worlds [7]. The idea of the possible worlds model is that when an agent has incomplete information, when he does not know the value of some propositions or statements, he considers all possible values as if there were parallel worlds, one for each value. Each world represents one state of affairs considered possible, given what the agent knows. The term *epistemic alternatives* has been coined to describe the worlds possible given an agents beliefs. Something true in all our agents epistemic alternatives can be said to be believed by the agent. There exist two well-known problems associated with this model [8] - that of knowing all valid formulae and that of knowledge/belief being closed under logical consequence - together, these constitute the well-known *logical omniscience* problem. It has been widely argued that this problem makes the possible worlds model unsuitable for resource bounded agents - and any real system is resource bounded [9].

1.1.2 Belief Desire Intention (BDI) Model

The Belief-Desire-Intention architecture draws its inspiration from philosophical theories of reasoning and planning [10]. It views the system as a rational agent having certain mental attitudes of Belief, Desire and Intention. These respectively represent the informational, motivational and deliberative states of the agent. From a philosophical point of view, beliefs represent the knowledge about the world. Desires are equivalent to goals whereas intentions are the chosen course of action adopted to accomplish the desires. From a computational viewpoint, beliefs are some way of representing the state of the world, which generally takes the form of expressions in the language of predicate calculus. Intentions are plans to which an agent has committed itself for achieving a desire are represented a stack of partially instantiated plans. At any point during the reasoning process, the BDI agent has access to the so called belief-accessible, desire-accessible and intention-accessible worlds and a event queue consisting of triggering (internal & external) events in response to which the BDI agent forms intentions.

Plans, as mental attitudes that guide the agent in its reasoning process, play a central part within the BDI framework. Plans are basically a part of the belief set of the agent and they capture the procedural information on how to achieve certain desires/goals given a set of contextual conditions. In this sense, plans can be looked upon as context-sensitive, event-invoked recipes that allow hierarchical decomposition of goals as well as the execution of actions [11].

1.2 Agent Behavioral Models

An agent cognition model establishes the theoretical properties of agents. Agent behavioral models consider the issues related to the construction of computer systems that satisfy the properties specified the agent cognition models. Researchers in this area are primarily concerned with the problem of constructing agent systems that will satisfy the properties specified by agent theorists. Limiting the scope of our discussion to this thesis, we present a widely accepted classification of various agent behavioural approaches in the following sub-sections.

1.2.1 Reactive

There has been wide disagreement on what the term 'Reactive' means when applied to an agent. The American Heritage dictionary defines reactive as "Tending to be responsive or to react to a stimulus". In AI, a common definition of reactive is "responding quickly and appropriately to changes in the environment". Wooldridge defines a reactive agent to be one that does not include any kind of central symbolic world model, and does not use complex symbolic reasoning [8] . The key to reactive agents is that they work in a hard-wired stimulus response manner. Certain sensory information always results in a specific action being taken. Reactive behavioral approaches have the following advantages [12]:

- High responsiveness
- Simple to both program and understand
- The system is deterministic and completely predictable
- They require little support infrastructure.

Reactive systems do have a couple of serious drawbacks as well. Firstly, every possible situation the agent might find itself in must be accounted for in the representation scheme. This places a huge burden of responsibility on the designer as they must have allowed for every eventuality. Related to this is the fact that for complex environments the range of possible situations can be vast making the design of the system very difficult, if not impossible. Another major drawback is that reactive systems have trouble forming all but the simplest long term plans. Reactive agents have no internal model of the world and so are incapable of reasoning about it in any abstract way. To reiterate, the agent simply receives input and react to these through simple rules.

The subsumption architecture [13], is arguably one of the best known reactive agent architectures. It was developed at MIT by Rodney Brooks - one of the most vocal and influential critics of the symbolic (deliberative) approach to agency [14]. Brooks [8] has propounded three keys ideas in support of the reactive approach to agency:

1. Intelligent behavior can be generated *without* explicit representations of the kind that symbolic AI proposes.

2. Intelligent behavior can be generated *without* explicit abstract reasoning of the kind that symbolic AI proposes.

3. Intelligence is a emergent property of certain complex systems.

These ideas indicate the main characteristics of reactive architectures - absence of a explicit world model, no goal-oriented long term planning and presence of a huge number of situation-dependent, locally interacting stimulus-action pairs leading to emergent behavior.

1.2.2 Deliberative

Contrary to a reactive agent, a deliberative agent has an internal model of the world and uses its model to reason about the effects of the actions in order to select actions that it predicts will achieve its goals. An agents internal model of the environment must provide certain basic functionality. In order to reason about the consequences of actions, the model must predict how the actions will affect the external state. The model must also be able to derive information about the external state from sensor output. In addition to the model, the deliberative agent needs an estimate of the current external state. It is from this estimated external state that the agent does projections to infer the consequences of potential actions. Given such a model, reasoning about the state of the world and planing are modeled as updates to the agents internal state. The result of the agents deliberation process is a plan to accomplish the task. The deliberative agent also needs to maintain a representation of the plan to be able to issue the chosen action execution commands to the control module. The representation of the plan also allows the agent to further elaborate and revise the plan as new information is gathered and more computation is done.

A system such as first order logic based on situation calculus can be used in order to formulate the model and generate plans. Such a system has the distinct advantage of being capable of planning its course of actions based on a well-defined reasoning formalism. However, due to the use of logic based inference, a deliberative system cannot make the real time guarantees usually required in a dynamic, multi-agent setting. Another drawback of a deliberative approach is that they require constant maintenance of the current state of the world. In fast moving real-time & dynamic environments, this can be a major issue. Also, consistency of the system must be maintained often involving updating any inferences made based on knowledge that may change. However, it must be noted that deliberative approaches have been at the heart of research in AI right from its inception.

1.2.3 Hybrid

Many researchers have argued the case for hybrid-systems, which attempt to combine deliberative and reactive approaches. Obviously, such hybrid system ought to have the advantages of both deliberative as well as reactive systems. Another argument put forth by proponents of hybrid systems is that real life agents need to adopt a situation dependent deliberative or reactive approach to solve problems. For example, a reactive system can be used for time-critical behaviors whereas long term planning can be achieved via deliberative reasoning. One approach towards hybrid systems is to build an agent out of two or more subsystems: a deliberative one, containing a symbolic world model, which develops plans and makes decisions in the way proposed by mainstream symbolic AI; and a reactive one, which is capable of reacting to events that occur in the environment without engaging in complex reasoning. Often, the reactive component is given some kind of a precedence over the deliberative one so that the agent can provide a rapid response to important environmental events [8].

1.3 Problem Definition and Research Agenda

Before we realise the implementation of an agent, it is important to precisely formulate its associated properties. The representation of knowledge and the reasoning approach to be used by the agent in order to make inferences must be specified formally. In this research, an equal importance shall be assigned to theory and practice. Beginning with the theoretical underpinnings of agents, we aim to investigate issues pertaining to representation, architecture and finally implementation.

A major *drawback of reactive agents* is that they are incapable of long-term goal oriented planning. It may be argued that a reactive approach is based on a completely different paradigm, which is devoid of a representation scheme conducive to long-term goal oriented planning. However, this seems to be very restrictive considering the fact that most real world problems involve making informed decisions based on a *causal & functional representation* of it. The aim of cognitive robotics is to provide a theoretical and computational account of exactly how it is that *deliberation can lead to action* **[4]**. Given this, it is tempting to suggest that cognitive robotics implies a deliberative approach to reasoning. Our work primarily investigates a deliberative behavioral approach. However, we do not ignore the possibility that some tasks might be better suited to more than one approach (or a particular one) and one aim of our research is to identify instances where this is applicable.

An *agent formalism* is a mathematical specification of the properties specified by agent theorists. It also includes a definition of the semantics of the reasoning methods that will be employed by the agent to accomplish tasks in its world. We look at various formalisms primarily with a aim of providing essential background. However, a brief comparison shall be provided so as to maintain completeness. Note that the comparisons shall be solely based on practical insights gained from using languages based on the said formalisms.

As stated before, *deliberative behavior* necessitates a complete specification of the causal laws of the domain. There are many different formalisms, based on various agent theoretical models, that could be utilized for the purposes of this specification. In this research, we consider the use of a *situation calculus* based language called *Indigolog*. Indigolog primarily provides high-level deliberative control facilities for dynamic environments. However, it is also possible to utilise its language constructs for implementing event-driven reactive behavior. The interpreter for the language automatically maintains an explicit model of the systems environment and capabilities, which can be queried and reasoned with at runtime [6]. Indigolog accounts for most of the phenomena (pertaining to dynamical systems) cited before - specification of causal laws, preconditions of actions, exogenous actions, complex actions & procedures, planning a course of actions, sensing, on-line plan generation interleaved with sensing etc. We believe such an account is necessary in order for a formalism to be utilised for decision making in the real world.

The central theme of our research is to overcome the limitations of reactive control by employing a high-level deliberative control mechanism that is based on mathematically rigorous formalisms for reasoning about action and change. The control mechanism will be applied in a realistic domain, the RoboCup Soccer Simulator, so as to effectively evaluate the suitability of the high-level deliberative approach as well as the underlying formalism. Within the context of our deliberative approach, we demonstrate a methodology for high-level agent control in the soccer domain whilst maintaining coherency between agent theory and practice.

1.4 Organisation of Thesis

- **Section 2** presents various languages that can be used to write agent programs. The languages covered, STRIPS, AgentSpeak(L), Indigolog, vary in terms of simplicity, expressibility and formalisms on which they are based. We present a formal introduction to the specification of each of the languages. A brief comparison, based on experiments using each of the languages, is also presented so as to indicate the advantages/disadvantages of each.

- **Section 3** is a detailed introduction to the Indigolog framework for incremental planning. It builds upon the language specification introduced in **2.4** and is aimed at demonstrating the mapping of the theory, which Indigolog is based on, into practice. A practical walk-through of the complete specification of a Indigolog program is also presented.

- The domain that will be used to experiment our investigated theories (and their respective implementations) is very crucial. This research focuses on (simulated) dynamic environments. As such, it is important that the selected domain satisfy properties associated with such environments. Moreover, the simulated environment should be flexible to use so as not to deviate our efforts from the primary research agenda. **Section 4** details one such environment, the *RoboCup Soccer Simulator*, which provides a simulated soccer environment and user-defined control of the soccer agents in it. It also covers our motivation for using the said simulator.

- In **Section 5**, we propose the agent architecture to be utilized in the soccer domain. A brief explanation of each of the components within the architecture is provided and the reader is referred to relevant parts of the thesis for a more thorough treatment. This section attempts to relate the various components that make up the soccer agent architecture used in this work.

- **Section 6** is a detailed account of the methodology adopted for implementing the soccer agents. Beginning with implementing simple soccer playing rules to complex deliberative strategies, this section presents a comprehensive treatment of the techniques to write Indigolog programs and utilize the available languages features.

- **Section 7** presents a brief evaluation of Indigolog. The discussion mainly concerns the language features and issues pertaining to the interpreter.

- In **Section 8**, **9**, and **10**, we cover the (low-level) implementation specific aspects of our soccer agent architecture. Section 8 briefly covers the low-level primitive skills repository on which high-level behavior of the soccer agents is based. Section 9 discusses the implementation of the component that interfaces the high-level prolog based component with the low-level C++ based skills library. In Section 10, we discuss our approach for combining the high-level reasoning component with the low-level skills component.

- **Section 11** discusses our solution to one of the problems, pertaining to the Indigolog interpreter, pointed out in Section 7.
- We conclude in **Section 12** with a brief discussion of related and future work.
- **Section 13** lists some of the software that was utilized or *developed* during the course of this thesis.

2 Agent Languages: A Comparative Study

Agent languages are programming languages that may embody the various principles proposed by theorists. They are software systems for programming and experimenting with agents. Those working in the area of agent languages address such questions as: How are we to program agents ? What are the right primitives for this task ? How are we to effectively compile or execute agent programs ?. In this section, we look at some popular action languages that have vastly different theoretical underpinnings. A brief comparison is also in place.

2.1 STRIPS

STRIPS (Stanford Research Institute Problem Solver) is a pioneering planning program developed around 1970 at SRI international. Although back then, the authors introduced it as a problem solving system, contemporarily it is more commonly referred to as a planning system. STRIPS, the representation language, derives from work on a mobile robot called SHAKEY at SRI International in the late 1960's.

STRIPS assumes that the world being world being modelled satisfies the following: [15]

- Only one action can occur at a time
- Actions are effectively instantaneous
- Nothing changes except as the result of planned actions

STRIPS operators are specified by pre-conditions and post-conditions. The preconditions are sets of atomic formulas of the language that need to hold before the operator can be applied. Operator post-conditions come in two parts consisting of a add-list and a delete-list which are sets of atomic formulas that need to be added to and deleted from the world model respectively. Intuitively, the delete list represents those properties of the world state that cease to exist after the operator is applied and the add list represents those properties of the world state that will hold after the operator is applied. An operator takes the world model database of some state and transforms it into a database representing a successor state. The main benefit of this way of representing and reasoning about plans is that *frame problem* can be completely avoided as the operator will change what it needs to in the database and leave the rest of it unaffected [15].

So more precisely, a strips problem is characterized by the tuple $<\mathrm{DB}_0$, Operators, Goal$>$ where,

1. DB_0 represents the world model which is a set of ground atomic formulas, similar to a database of facts.

2. Goal is a list of atoms whose variables are understood existentially.

3. Operators is a list of operators of the form <Act, Pre, Add, Del> where Act is the name of the operator and Pre, Add and Del are lists of atoms specifying the precondition, add-list and the del-list respectively for the operator.

Given the above formulation, a solution to the planning problem is a set of operators that can be applied in sequence starting with a initial state without violating any pre-conditions and which results in a world model that satisfies the goal formula [15]:

1. solution as a sequence $<\mathrm{Act}_1\theta_1, ..., \mathrm{Act}_n\theta_n>$ where Act_i is a operator of the form $<\mathrm{Acts}_i, \mathrm{Pre}_i, \mathrm{Add}_i, \mathrm{Del}_i>$ and θ_i is a substitution of constants for the variables in that operator

2. the sequence satisfies the following:

- for all $1 \leq i \leq n$, $\mathrm{DB}_i = \mathrm{DB}_{i-1} + \mathrm{Add}_i\theta_i - \mathrm{Del}_i\theta_i$;
- for all $1 \leq i \leq n$, $\mathrm{Pre}_i\theta_i \subseteq \mathrm{DB}_{i-1}$;
- for some θ, $\mathrm{Goal}\theta \subseteq \mathrm{DB}_n$.

2.2 AgentSpeak(L)

AgentSpeak(L) is a first-order language that realises BDI agents [11]. The behavior of the agents, encompassing its interaction with the environment and other agents, is guided by the programs written in AgentSpeak(L). The current state of the agent can be viewed as its current belief state whereas states which the agent wants to bring about based on its external or internal stimuli can be viewed as desires. The adoption of programs to satisfy such stimuli can be viewed as intentions. This is also consistent with the BDI theory which defines intentions as the chosen course of action execution to accomplish a desire. Note that belief, desires and intentions of the agent are not explicitly represented as modal formulas in the language. Instead, it is the designer who ascribes the notions to the agent written in AgentSpeak(L). It has been argued in [16] that such a perspective of taking a simple specification language as the execution model of a agent and then ascribing notions of beliefs, desires & intentions from an external viewpoint has a better chance of unifying theory and practice.

The alphabet of the formal language consists of variables, constants, functions symbols, predicate symbols, connectives, quantifiers and punctuation symbols. In addition to the usual first-order connectives, ! (for achievement), ?(for test). ;(for sequencing) and $\leftarrow$(for implication). Following elements make up the core part part of the language:

Let $\vec{t}$ be a vector of first order terms.

- **Base beliefs**: If b is a predicate symbol, and $\vec{t}$, then b($\vec{t}$) is a belief atom. If b($\vec{t}$) and c($\vec{t}$) are belief atoms, b($\vec{t}$) $\wedge$ b($\vec{t}$) are belief atoms. A belief atom or its negation is referred to as a *belief literal.* A belief atom that is ground is referred to as a *base belief.*
- **Goals**: If g is a predicate symbol, and $\vec{t}$, then !g($\vec{t}$) and ?g($\vec{t}$) are *goals.* Acquisition of new goals leads to *triggering of events.*
- **Triggering Events**: When a agent acquires new goals or notices a change in its environment, it may trigger additions and/or deletions to its goals or beliefs. These events are referred to as *triggering events.* Addition is denoted by the operator '+' whereas deletion by '-'. Formally, if b($\vec{t}$) is a belief atom, !g($\vec{t}$) and ?g($\vec{t}$) are goals, then +b($\vec{t}$), -b($\vec{t}$), +!g($\vec{t}$), -!g($\vec{t}$), +?g($\vec{t}$) and -?g($\vec{t}$) are *triggering events.*
- **Actions**: If 'a' is a action symbol and $\vec{t}$, then a($\vec{t}$) is an action.
- **Plans**: If e is a triggering event, b_1,...,b_m are belief literals[1], and h_1,...h_n are goals or actions, then e **:** $b_1 \wedge \ldots \wedge b_m \leftarrow h_1 \wedge \ldots \wedge h_n$ is a *plan.* The expression to the left of the arrow is the *head* of the plan whereas that to the right is referred to as its *body.* The expression to the right of the colon in the head is referred to as the *context.* The expression *true* denotes a empty body.

2.3 Situation calculus and Golog

Situation calculus [1] is a formalism specifically designed for representing dynamically changing domains. The calculus can be used to represent and reason about actions and their effects. All changes to the world are the result of named *actions.* A possible world history which is simply a sequence of actions, is represented by a first order term called a *situation.* The constant S_0 is used to denote the *initial situation*, namely the situation in which no actions have yet occurred. There is a distinguished binary function symbol *do*; *do(α, s)* denotes the successor situation to s resulting from performing the action α. Actions may be parameterised. For example, *put(x, y)* might stand for the action of putting object x on object y, in which case *do(put(A, B), s)* denotes the situation resulting from placing A on B when the world is in situation *s*. Notice that in the situation calculus, actions are denoted by first order terms, and situations (world histories) are also first order terms. For example, *do(putdown(A), do(walk(L), do(pickup(A), S_0)))* is a situation denoting the world history consisting the sequence of actions *[pickup(A), walk(L), putdown(A)].*

[1]Note that by definition, these will always be ground.

Before we introduce the GOLOG programming language, some terminology/notation used in the definitions **[6]** follows:

- **Fluents**: In the context of situation calculus, a fluent is a situation specific property of the domain.
- **Functional fluents**: Functions whose denotations vary from situation to situation are called functional fluents.
- **Relational fluents**: Relations whose truth values vary from situation to situation are called relational fluents.
- **do(α, s)**: This is a situation calculus term denoting the situation resulting from performing action α in situation s.define
- **Do(δ, s, s')**: Intuitively, Do(δ, s, s'), where δ is a complex action expression, holds whenever s' is a terminating situation of an execution of complex action δ starting in situation s.
- **Positive effect axiom**: ${\gamma_{\mathrm{F}}}^{+}(\vec{x}$, a, s) is a formula describing under what conditions doing the action a in situation leads to the fluent F to become true in the successor situation do(a, s).

  $\texttt{Poss(a, s)} \wedge {\gamma_{\mathrm{F}}}^{+}(\vec{x}\texttt{, a, s)} \supset \texttt{F(}\vec{x}\texttt{, do(a, s)).}$

- **Negative effect axiom**: Similar to the positive effect axiom, $\gamma_{\mathrm{F}}-(\vec{x}$, a, s) describes the conditions under which performing action a in situation s results in F becoming false in the successor situation s.

  $\texttt{Poss(a, s)} \wedge \gamma_{\mathrm{F}}-(\vec{x}\texttt{, a, s)} \supset \neg\texttt{F(}\vec{x}\texttt{, do(a, s)).}$

A solution to the frame problem (from a situation calculus & golog point of view) rests on the so called completeness assumption which is that the above positive and negative effect axioms completely characterize all the conditions under which action a can lead to the fluent F becoming true (respectively, false) in the successor situation. In effect, the axioms describe all the causal laws affecting the truth values of the fluent F.

2.3.1 Golog

GOLOG, which is based on McCarthy's situation calculus [1], is a high-level language for implementing dynamical systems such as applications in robotics, intelligent software agents, discrete event simulation etc. It is based on a extended version of situation calculus, which is a first order language for representing dynamically evolving domains. It has situation calculus semantics and an interpreter that executes actions in real or simulated settings. Its interpreter is a theorem prover that performs arbitrary first order reasoning by maintaining an explicit representation of the dynamic world being modeled. This is done on the basis of user supplied axioms about the preconditions and effects of actions and the initial state of the world. The following subsections present what constitutes the *golog programming language*:

- **Primitive actions**:

  $\texttt{Do(a, s, s') = Poss(a[s], s)} \wedge \texttt{s'= do(a[s], s).}$

The notation a[s] means the result of restoring the situation arguments to any ***functional fluents*** mentioned by the action term a. For example, if the action term is read(favorite_book(John)) and favorite_book is a functional fluent then a[s] is read(favorite_book(John, s)).

- **Test actions**:

  $\texttt{Do(}\varphi\texttt{?, s, s') = }\varphi\texttt{[s]} \wedge \texttt{s = s'.}$

Here, φ is a pseudo-fluent expression (not a situation calculus formula) which stands for a formula in the language of the situation calculus but with all situation arguments suppressed. As before, φ[s] denotes the situation calculus formula obtained from $\varphi$by restoring situation variable s as the suppressed situation argument for all fluent names (relational & functional) mentioned in φ. Examples: If φ is

$$(\forall x).\texttt{ontable(x)} \wedge \neg\texttt{on(x, A).}$$

then φ[s] stands for

$$(\forall x).\texttt{ontable(x, s)} \wedge \neg\texttt{on(x, A, s).}$$

If φ is

$$(\exists x)\texttt{on(x, favorite\_block(Mary)).}$$

then φ[s] stands for

$$(\exists x)\texttt{on(x, favorite\_block(Mary, s), s).}$$

- **Sequence**:

  $\texttt{Do([}\delta_1\texttt{;}\delta_2\texttt{], s, s') = (}\exists \texttt{s}^*\texttt{). Do(}\delta_1\texttt{, s, s}^*\texttt{)} \wedge \texttt{Do(}\delta_2\texttt{, s}^*\texttt{, s').}$

- **Nondeterministic choice of two actions**

  $\texttt{Do((}\delta_1\texttt{|}\delta_2\texttt{), s, s') = Do(}\delta_1\texttt{, s, s')} \vee \texttt{Do(}\delta_2\texttt{, s, s').}$

- **Nondeterministic choice of action arguments**:

  $\texttt{Do((}\Pi \texttt{x)}\delta\texttt{(x). s. s') = (}\exists \texttt{x) Do(}\delta\texttt{(x), s, s').}$

- **Nondeterministic iteration**:

$$\texttt{Do(}\delta^*\texttt{, s, s') =}$$
$$(\forall \texttt{P). \{(}\forall \texttt{s}_1\texttt{)P(s}_1\texttt{, s}_1\texttt{)} \wedge (\forall \texttt{s}_1\texttt{, s}_2\texttt{, s}_3\texttt{)[P(s}_1\texttt{, s}_2\texttt{)} \wedge \texttt{Do(}\delta\texttt{, s}_2\texttt{, s}_3\texttt{)}$$
$$\supset \texttt{P(s}_1\texttt{, s}_3\texttt{)]\}} \supset \texttt{P(s, s').}$$

In other words, doing action δ zero or more times takes you from s to s' iff (s, s') is in every set (and therefore, the smallest set) such that:

1. (s_1, s_1) is in the set for all situations s_1.
2. Whenever (s_1, s_2) is in the set, and doing δ in the situation s_2 takes you to situation s_3, then (s_1, s_3) is in the set.

Note that the above definition of nondeterministic iteration utilizes the standard second order way of expressing this set. This is because transitive closure is not first-order definable, and nondeterministic iteration appeals to this closure.

Complex actions in Golog are defined using some extralogical symbols (e.g., while, if etc.) which act as abbreviations for logical expressions in the language of situation calculus. The extralogical symbols should be thought of as macros which expand into genuine formulas of the situation calculus. Note that the complex actions may be nondeterministic, that is, they may have several different executions terminating in different situations. The language has been proposed in [6]. A similar but more detailed account can also be found in [4].

2.4 Indigolog

Indigolog is a extension of GOLOG with added functionality for sensing (getting perceptual input) and exogenous actions (actions occurring in the environment and not executed by the agent). It allows the programmer control over on-line and offline execution, and in the on-line case, allows sensing information to affect subsequent computation/planning [17]. Contrary to the completely offline execution offered by Golog, Indigolog (Incremental Deterministic Golog) provides a more realistic on-line incremental execution. The following explanation from [18] should help make the distinction clear:

Let the program execution be specified in terms of two predicates, Final and Trans. Final(δ, s) holds if program δ can legally terminate in situation s: Trans(δ_1, s_1, δ_2, s_2) holds if one step of δ_1 in situation s_1 leads to a situation s_2 with remaining δ_2 to be executed.

- For off-line execution, we look for a sequence of Trans steps leading to a final termination whereas
- For a on-line incremental execution, we look for any single action A such that program entails Trans(δ, s, δ_1, do(A, s)), commit to it and repeat the search with the remaining program.

Indigolog supports complex agents that are:

- able to do reasoning and planning
- able to react to exogenous events

- able to monitor plan execution and sense the environment
- both reactive & proactive
- written using very high level language constructs

An Indigolog program consists of two main parts: A *domain theory* and *behavior specification.* The domain theory specifies the domain dependent primitive actions and tests of predicates whereas the behavior specification specifies its dynamics - the causal laws of the domain. The meaning for each of these will be made precise in sub-section 3.2.

2.5 Reactive Action Packages (RAP)

A robot acting in the real world must use flexible plans. This is because actions will sometimes fail to produce their desired effects and unexpected events will sometimes demand the robot shift its attention from one task to another. A plan is usually construed as a list of primitive robot actions to be executed one after another but in a complex domain a plan must be structured to cope effectively with the myriad unpredictable details it will encounter during execution. However, adding structure to a plan involves more than augmenting the primitive plan representation; it requires adopting a situation-driven model of interaction with the world.

Situation-driven execution assumes that a plan consists of tasks with three major components: a satisfaction test, a window of activity, and a set of execution methods that are appropriate in different circumstances. Execution of such a plan proceeds by selecting an unsatisfied task and choosing a method to achieve it based on the current world state. A task may be executed as many times as necessary to keep it satisfied while it is active. Many different sequences of task execution may be possible with a plan, including concurrent execution of some tasks, that satisfy certain temporal constraints [19]. The plan may be ***sketchy,*** in the sense that it might include tasks/goals which may not be directly executed by the agent (IE., tasks are not primitive actions). Presence of sketchy plans is a characteristic feature of complex, dynamic domains . ***Tasks/Goals*** and ***Methods*** are the two basic components of a situation driven execution model. A brief description for the two follows:

- ***Tasks***: A task can be best thought of as a goal to be achieved.The goal may be a state to be achieved, a state to be maintained or a action to carry out. A task consists of two parts: An index to use with the current situation for selecting a method and a test for determining when the task should be considered for execution.
- ***Methods***: A method is a set of actions that will achieve the task in a given situation. It is a prescription for changing the world situation so that a given task becomes satisfied. Each method may have certain contextual

(situation dependent) conditions associated with it. The effect of this is that a task may have more than one associated accomplishment methods with each of them being applicable in different situations.

The RAP system proposes a plan and task representation based on program-like reactive action packages. Within the system, execution monitoring becomes an intrinsic part of the execution algorithm, and the need for separate re-planning on failure disappears. Rap's are more than just programs that run at execution time, however, they are also hierarchical building blocks for plan construction. The RAP representation is structured to make a task's expected behavior evident for use in planning as well as in execution. The RAP execution system described includes a sensor memory, representation language and interpreter.

2.6 Comparison

The comparisons here are based on experiments involving design of a delivery agent using each of the action languages. All related source code can be found in the appendices. Also, note that RAP has not been included in the comparative analysis because of time constraints. For STRIPS, a simple means-end planner based on goal regression [20] has been implemented from scratch whereas the AgentSpeak(L) planner is a modified version of an existing implementation based along the lines of [16]. The Indigolog interpreter, publicly available from the Cognitive Robotics Group at University of Toronto, has been used without modification.

All the three approaches, STRIPS, AgentSpeak(L) and Indigolog, considered herein have their own strengths and weaknesses. At one extreme is the STRIPS implementation, as used here, which offers absolutely no reactivity whereas at the other is the BDI implementation that is a highly reactive. INDIGOLOG offers various degrees of reactive and deliberative control. STRIPS simply deals with reaching a desired state from a given initial state in a offline manner by chaining through a set of plan operators. Contrarily, AgentSpeak(L) and Indigolog follow a incremental approach wherein action execution is interleaved with environment monitoring as well as sensing (in the Indigolog case).

In general, STRIPS is a very simple way to represent actions and their effects. It has no notion of exogenous actions and the STRIPS planner is completely offline. Without incorporating domain specific heuristics and/or constraints into the planner, it is not possible to come up with planner that works well on large problems. The simple means-ends planner, as used for the purposes of our comparison, degrades severely on anything but toy problems inspite of the constraint that all action variables be ground during the planning process. The AgentSpeak(L) and INDIGOLOG approaches definitely rank high in this regard as a lot of application dependent control is used to guide the planner towards achieving the goal. For example: in AgentSpeak(L), plans (being event-invoked) help the planner to constrain the search to those plans whose head matches the triggering event. Similarly, Indigolog allows the hard-coding of complex actions out of primitive ones and various other high level constructs which makes up the procedural part of the agent. What remains is specifying the dynamics of the system consisting of a domain theory and behavior specification.

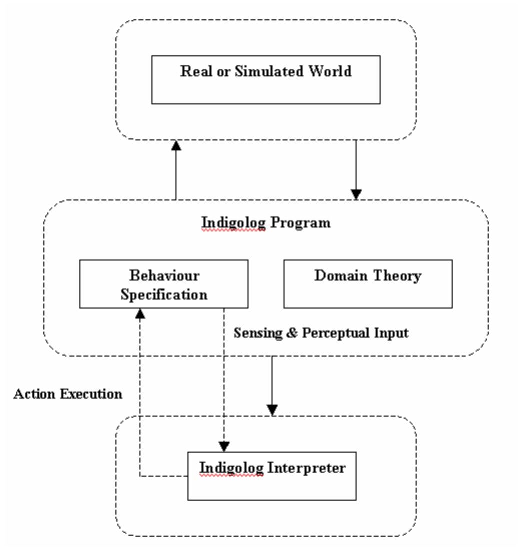

Figure 1: Generic Framework in Indigolog for Deliberative Control

3 Deliberative Control Using Indigolog

Golog looks a lot like a standard procedural programming language. However, the interpreter for this language uses the declarative specification to reason about what the effects of various possible actions would be. As explained before, Indigolog was developed to facilitate incremental execution in which sensing can be used to affect subsequent planning. Later in section, we shall demonstrate through a practical example the specification of a Indigolog program. In the following sections, we discuss the Indigolog framework - The interaction between the planner/interpreter, the Indigolog program & the real or simulated world in which the agent exists. We also present a practical walk-through demonstrating the complete specification of a Indigolog program. Fig. 1 shown above shall be used for referential purposes. Also, the delivery agent code used for comparative purposes in sub-section 2.6 has been used for the walk-through.

3.1 The Framework

The Indigolog framework illustrated in Fig. 1 conceptually represents three components - The language interpreter, the program and the agents environment, which may be a real or simulated. The interpreter works out the next action to be executed based on the domain theory and behavior specification represented by the Indigolog program. However, it is left up to the program to define the semantics of actual execution. Action execution may be defined as sending a command to a robot working in the real world or a simulated agent in a artificial world. Likewise, sensing (as and when it is necessary) and exogenous action monitoring (at every discrete time-step) is done indirectly via the program. The program is free define the actual meaning of these operations.

3.2 Indigolog Program Specification - A practical walk-through

Indigolog's high-level programs contain primitive actions and tests of predicates that are domain dependent and a interpreter for such programs must reason about these. In effect, we need to provide the domain specific details for our programs. As mentioned before, these come in two parts, *the domain theory & behavior specification*, each of which are discussed in the subsections that follow. The aim here is to illustrate, through a practical walk-through, the steps involved in the complete specification of a Indigolog program.

3.2.1 Domain Theory

- **Primitive Actions**: Every action that could be ever executed must be defined using a primitive actions clause. It takes the form; prim_action(α) where α is a ground atom/symbol denoting the action name. This forms a **SET A** of primitive actions that can be executed in the agents environment.

  ```
  prim_action(turn_left).
  prim_action(go_straight).
  ```

- **Exogenous Actions**: These are the actions/events occurring in the system and not executed by the agent. The definition of such actions takes a similar form as primitive actions; exog_action(α) where α is again a ground atom/symbol denoting the name of the exogenous action. This forms a **SET E** of the exogenous actions that may act upon the environment/agent.

  ```
  exog_action(button_on_rcx_pushed).
  exog_action(delivery_requested(From, To)).
  ```

- **Primitive Fluents**: These characterize the state of the system, IE: the agent and its environment. Their definition takes the form prim_fluent(β) where β is a symbol denoting the name of the fluent. It may even be a arbitrary prolog term with variables. This forms a **SET F** of fluents which completely characterize the state of the system.

  ```
  prim_fluent(motion).
  prim_fluent(holding_delivery_for(Loc)) :-
      way_station(Loc).
  ```

- **Initial State**: The initial state definition takes the form initially(β, value) where $\beta \epsilon$ F and value is any symbol or predicate.

```
initially(motion, stopped).
initially(current_location(station1), 1).
initially(button_on_rcx_pushed, 0).
```

3.2.2 Behavior Specification

To complete the description of a Indigolog program, one must now encode the ***behavior specification*** for it - the action preconditions, causal laws of the domain (effects of actions), complex actions/condition (if any) and the main control procedure. what follows is a brief description of each of theses components:

Let α and β be the symbols denoting the action name and the fluent name respectively such that $\alpha \epsilon \mathbf{A} \bigvee \alpha \epsilon \mathbf{E}$ and $\beta \epsilon \mathbf{F}$.

- **Preconditions**: As the name suggests, these specify the action preconditions. They take the form poss(α, γ) where γ is a condition composed of any combination of the binary '**and**' or '**or**' operators and the unary '**not**' operator. Alternatively, the symbol '**true**' can be used as the condition argument to imply that it is always possible to execute the action.

```
poss(request_delivery(_, _), true).
poss(straight, and(at_station = 1, motion = stopped)).
```

- **Causal Laws**: They specify the so called '**effects of actions**'. They take the form causes_val(α, β, v, γ). It could be read as : "Action α causes fluent β to take the value denoted by v iff the condition γ is satisfied where γ is as defined before with the added provision that it could even consist of complex conditions.

```
causes_val(arrive_at_station, motion, stopped, true).
causes_val(dropoff_load, deliveries_held, Update,
               and(To = current_location,
                 and(member([From, To], deliveries_held),
                   delete(deliveries_held, [[From, To]], Update)))).
```

- **Complex actions/conditions**: The complex actions/conditions are of the form, proc(actionName, body) where actionName may be either a symbol or a predicate with arguments. The body may consist of primitive actions, the non-deterministic choice operator and if..else constructs. This makes our complex actions non-deterministic, in the sense that they may have several different executions terminating in different situations.

```
%%Complex Action
```

```
proc(head_to_next_station(P, C, N),
       pi(action, %%non-deterministic choice of action arguments
           [if(way(P, C, N, action), action,
              debug('Could not find the way ->', action))]
)).
%%Complex Condition
proc(any_pickups(Pending_Deliveries, Current_Loc),
       member([Current_Loc, _], Pending_Deliveries)).
```

- **Main control procedure**: Although technically this is no different from any other complex action/condition, its just that we pass this main proc to the the Indigolog interpreter to evaluate the program and in this sense, we call it the main control procedure. The body of the proc here consists of a special control structure called ***prioritized_ interrupts*** which consists of a series of interrupts in decreasing order of priority. Each interrupt can be though of as a condition-action pair. This takes the following form:

```
proc(program_name, prioritized_interrupts(
         [interrupt(γ1, [δ1]),
                ⋮
          interrupt(γn, [δn])
         ])).
```

Here, γ_i will be tested only when all other γ_k, such that $k < i$, have been tested and evaluated to be false in which case the corresponding sequence of steps are executed. The program fails if none of conditions evaluate to be true.

```
proc(control, prioritized_interrupts(
        [
         interrupt(motion = moving,
                  [
                   debug('motion is moving, heading to', heading_to),
                   wait
                  ]),
         interrupt(and(pending_deliveries = [], deliveries_held = []), wait),
         interrupt(at_station = 1,
                  [
                   if(any_pickups(pending_deliveries, current_location),
                        pickup_load, %% primitive action
                        debug('No Pickup at', current_location)),
                   if(any_dropoffs(deliveries_held, current_location),
                        dropoff_load,
                        debug('No Droppoff At', current_location)),
                   head_to_next_station(prev_location, %%complex action
                        current_location, heading_to),
                   if(trajec_served(),
                                re_schedule, %% internal action
                                debug('Continuing delivery...', nl))
                          ])
                ])
        ).
```

- **The 'execute' clause**: This clause is invoked by the interpreter when there is an action to be executed. The application can decide what it means to execute a particular action. Along with the action name, a History of actions performed so far is also passed for debugging purposes.

```
execute(Action, History, SensingResult) :-
    write('Executing action: '), write(Action), nl,
    actionNum(Action, N),
    sendRcxActionNumber(N, SensingNumber),
    translateSensorValue(Action, SensingNumber, SensingResult),
    write('Sensing result: '), write(SensingResult), nl.
```

- **The 'exog_occurs' clause**: This clause, which returns a list of exogenous actions occurred since the last time it was called must be defined for all application programs. In case there are no exogenous actions, a null list may be returned. A further restriction that applies is that this clause must never fail. The following is a prolog code snippet defining such a clause:

```
exog_occurs(ExogList) :-
    checkRcxExog(RcxExogList),  %% Check exog at RCX.
    (RcxExogList == [] -> true;
        (write('Rcx exogenous action: '), write(RcxExogList), nl)),
    checkOtherExog(OtherExogList), %% Check keyboard exog.
    (OtherExogList == [] -> true;
        (write('Other exogenous action: '), write(OtherExogList), nl)),
    append(RcxExogList, OtherExogList, ExogList).
```

- **Initialization clause**: This will be called by the interpreter upon startup. All initialization tasks like setting up of sources of exogenous actions etc can be handled here. For example:

```
initialize :-
        initializeRcx,
        initializeExog.
```

- **Finalization clause**: Any cleanup can be performed here.

```
finalize :-
        finalizeRcx,
        finalizeExog.
```

Both, the initialization and finalization clauses are called only once, during program startup and termination respectively.

4 The Robocup Soccer Simulator: A testbed for our Agent Programs

In 1995, Kitano et al. [21] proposed the first Robot World Cup Soccer Games and conferences to take place in 1997. The aim of Robocup is to present a new standard problem for AI and Robotics, somewhat jokingly referred to as the life of AI after Deep Blue [2][22]. Robocup differs from previous research in AI by focusing on a distributed solution (in our context, often referred to as Multi-agent system or Distributed AI) rather than focusing on a centralized solution, and by challenging researchers from not only AI-related fields, but also researcher in the areas of robotics, sociology, real-time mission critical systems etc. Currently, ROBOCUP includes four robotic soccer competitions (simulation, small-sized, mid-size & legged) and two disaster rescue competitions (simulation & real-robot). For our purposes, we focus on the robocup simulation league[3]

4.1 Domain characteristics

In our opinion, the RoboSoccer domain provides a perfect test bed for the action theories introduced in the previous sub-sections. A few characteristics inherent in the domain are cited here to justify the claim.

- ***The Environment is non-deterministic***: There are potentially many different ways in which the environment may evolve.
- ***The System itself is non-deterministic***: At any given time, there are potentially many different actions or procedures the system can execute.
- ***Multiple Objectives***: At any given instant, there are potentially many different objectives the system is asked to accomplished.
- ***Hidden state***: Each *Agent* only has a partial view of its environment.
- ***Noisy Sensors and Actuators***: The *Agents* do not perceive the world exactly as it is, nor can they affect the world exactly as intended.
- ***Asynchronous***: The perception and actions cycles are asynchronous, prohibiting the traditional AI paradigm of using perceptual input to trigger actions.
- ***Limited Communication***: Communication opportunities are limited; the agents must make and implement their decisions in real-time.

[2]see http://www.chess.ibm.com for details on deep blue.

[3]Henceforth, the terms Robocup and Robocup simulation league might be used interchangeably.

These domain characteristics [23] (*dynamic, real-time, distributed, multi-agent*) combine together to make Robocup a realistic and challenging domain for conducting our experiments. The simulation league is based on the soccer simulator briefed in the following section.

4.2 The Soccer Simulator

The simulator is the system comprising of several sub-systems which collectively provide the environment to setup a simulated soccer match. Each player is represented as an agent program in a arbitrary language. Here, I briefly enumerate the core subsystems within the simulator. Details can be found in [22].

- ***Soccer Server***: This is the heart of the simulator. It executes as a server process and communicates with each of its clients (players) via UDP/IP sockets. This enables representation of client programs in any arbitrary language on any system with UDP/IP facilities, with each client controlling the movement of one individual player. The soccer server provides a virtual field and simulates all movements of a ball and players. The server also provides the players with sensory information relating to position of the ball, goals & other players etc. One important point to note here is that the server provides a highly realistic environment for the simulated game; it is a real time system working with discrete time intervals wherein each cycle has a specified duration and actions that need to be executed in a cycle, must arrive to the server during the right interval.

- ***Soccer Monitor***: The monitor is a visualization tool that allows people to see what is happening within the server during a game. The information shown on the monitor includes the score, team names and the positions of all the players and the ball. It must be noted however that the monitor is not required to run a game on the server. Also, multiple monitors may be connected to the server at the same time if it is desired that the same game be shown at different terminals.

- ***Logplayer***: When running the server, certain options may be used that will cause the server to store/record all the details of the match onto the hard drive. The logplayer, combined with the monitor, is a tool that can be used to replay a game using the stored data. The logplayer comes in very handy to debugging purposes, team analysis and discovering the strong or weak points of a team. It is very flexible in the sense that it is equipped with play, fast forward, rewind & pause buttons. Also, it is possible jump to a particular cycle in the game (for eg., if one only wants to see the goals).

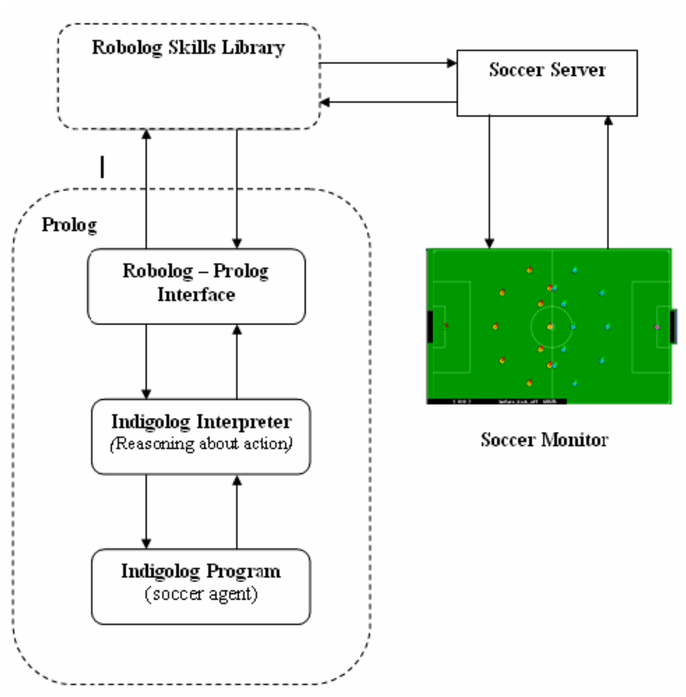

Figure 2: Soccer Agent Architecture

Figure 2. Soccer Agent Architecture

5 Soccer Agent Architecture

5.1 Design Goals

Our robocup soccer agent architecture (see figure 2) is a first attempt at utilizing high-level deliberative reasoning in the soccer domain. Most of the design goals have been set keeping in mind future enhancement of the system so as to be able to demonstrate the capabilities of Indigolog based reasoning in simulated soccer arena. Following are the main goals on which our design is based:

- **Establish inter-component boundaries**: Notice that our system is composed of many different sub-systems which stand out on their own. Combining the skills library (C++ based), the Indigolog Framework (Prolog based) and the robocup soccer simulator (a complex system in its own right) has been a challenging task. It is important to avoid coupling of each of the components by establishing standards to interface each of the communicating components.

- **Facilitate extensibility**: Extensibility is the most important criteria in our design. The current architecture facilitates easy extension/enrichment of each of the components without affecting the rest of the system. For example, the skills library may be easily extended and/or replaced with a new one. Likewise, more complex strategies may be added by extending the Indigolog program to make use of the enhanced primitive skills component.

Every run of the simulation involves 22 different instances of the agent, each working out its own plan, the soccer simulator and the soccer visualisation tool. It is very important that the system be easy to experiment and debug. Although no special measure have been taken to ensure this, it follows naturally from the design if the above two goals are achieved.

5.2 Main Components

Figure 2 above shows the overall system architecture used in our work. Discussion pertaining to each of the components shall be (or has already been) treated in greater detail in appropriate places. Here, we provided a brief overview of each of the components of our system. A more thorough treatment of each can be found in the cross-referenced sections.

- **Soccer Agent**: This represents the highest level of abstraction in our system. The soccer agents are represented by a high-level Indigolog program, whose domain theory consists of the primitive actions and fluents specific to the soccer domain and whose behavior specification would consist of soccer playing rules. We would like to re-iterate here that developing a soccer agent using Indigolog is conceptually similar to writing a program in any other language with the difference that the basic statements of the language are domain dependent actions that can be performed by the agent and that conditions are based on fluents, the agents beliefs about the world. More details regarding the development of the soccer agents can be found starting **Section 6.1** onwards.

- **Indigolog Interpreter**: This is the most important component within the soccer agent architecture. It is representative of our deliberative approach to reasoning applied in the soccer domain. Details can be found in **Section 3**.

- **Foreign Language Interface**: The foreign language interface does the job of connecting the high level reasoning component with the low level primitive skills library. More precisely, it makes available the primitive skills that are written in C++ to a prolog program. Although it might appear that implementing the interface involves development at both ends, both prolog and C++, that is not true. SWI-Prolog facilitates a very flexible mechanism to support such functionality, which only involves development on the foreign language[4] side of the interface. Details can be found in **Section 9.1**.

- **Primitive Skills Library**: The primitive skill library is the repository consisting of low-level skills relevant to the soccer domain. It encapsulates the complex protocol which need to be implemented in order to communicate with the soccer server and implements the basic behavioral blocks that lie at the very foundation of our system. Further details can be found in **Section 8**.

- **The Simulated Soccer World**: As discussed in **Section 4**, this primarily consists of two main components - the soccer server and the soccer monitor. The server maintains the soccer world model and makes the same available to client programs connected to it. Note that most of the rules enforced by the server are user configurable. The monitor in turn provides a visual display of the soccer game.

6 Soccer Agents: Implementation Methodology

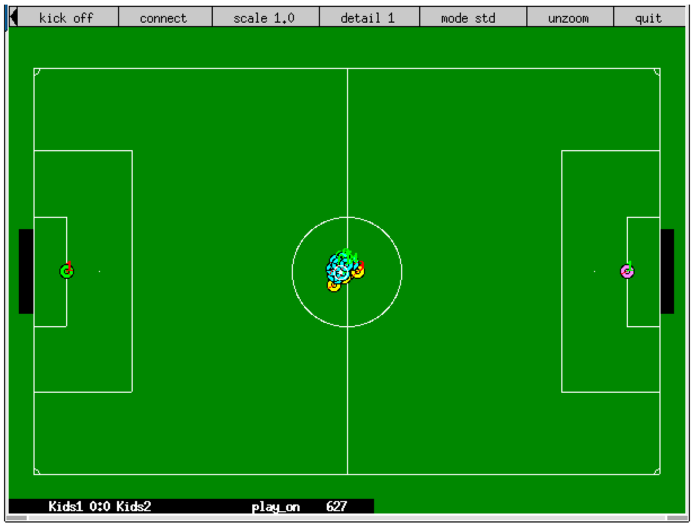

Illustration 1: Kid Soccer like behavior emerging out of minimally deliberative control

In this section, we begin the discussion pertaining to the implementation of soccer agents. Our use of Indigolog for agent control in the soccer domain is new. As such, we present our implementation using a demonstrative approach. Such an approach facilitates our aim to establish a methodology for the construction

[4]As of now, only C/C++ is supported as a foreign language.

of high-level deliberative behavior within the context of Indigolog and the soccer domain.

6.1 The 'Hello World' Stage - Minimal Deliberation

One of the earliest experiments involved writing agent programs with minimal deliberative capabilities. All the agents essentially adopt the same rule, shown below in **RULE 1**, for playing soccer. Note that emphasized words represent domain specific primitives or conditions.

```
    if game_not_started
       wait[5]
    else
    {
       if ball_moving
          wait
       else
          gotoStaticBall
       if ball_kickable
          kick_and_run
    }
Rule 1: A simple decision rule.
```

```
proc(minimalDeliberation, prioritized_interrupts(
        [
        interrupt(play_mode = before_kick_off, wait),
        interrupt(and(play_mode = play_on, ballKickable),
                             [
                              debug('Ball is Kickable'),
                              kick_and_run
                             ]),
        interrupt(play_mode = play_on,
                             [
                              if(ballMoving,
                              [
                               debug('Ball Moving: True'),
                               wait
                              ],
                              [
                               debug('Ball Moving: False'),
```

[5]**Wait** is a Indigolog keyword which puts the agent program in a passive mode. The *wait* is non-deterministic in the sense that further execution completely depends on the happening of a exogenous action/event that is causally related to some property/fluent of the current situation.

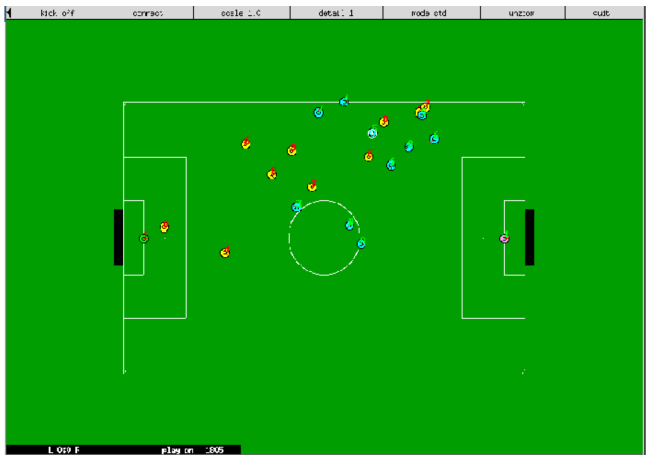

Figure 3: Agents maintain proximity from each other

```
                                        gotoStaticBall
                                      ])
                                    ])
     ])
   ).
```

Code Listing 1: A Minimally Deliberative control procedure.

The code snippet above realizes **Rule 1**. It is minimally deliberative because it does not make use of any complex reasoning and uses a very small subset of the available primitive actions. As such, the behavior it results in too is very simple and fails to achieve soccer playing goals. The resultant behavior is shown in illustration 1. As can be seen, all the players quickly[6] converge near the ball and fight for it with the effect of no player ever getting the ball inspite of being in kicking range. Note that the agent implicitly assumes knowledge of the ball-position and its validity. For brevity, this has been purposely built into the 'gotoStaticBall' primitive as the intent here is to demonstrate the behavior arising out of minimal deliberative control.

6.2 Prevent Crowding of Players

The minimally deliberative rule was based on a few complex conditions and primitive actions. It did not make use of its sensing capability which is indeed essential for any reasonably complex behavior. Since it is our aim to make soccer agents increasingly exhibit deliberative behavior, the next task at hand is to prevent the crowding of players that results from minimal deliberative control 6.1. For this purpose, it seems appropriate to make use of a domain

[6]Precisely, 627 simulation steps.

fluent that captures the number of team mates that are closer to the ball than the particular agent under consideration. Lets consider the following procedural rule:

```
    if game_not_started
       wait
    else
    {
       scanFieldWithBody
       if ball_kickable
          kickStraight[7]
       else if myTeamWithBall
       {
          senNumberTeamCloseToBall  %% sensing action causally
                                    %% related to app. fluent
          if(numTeamMatesCloseToBall < someThreshold)
             getBall
       }
       else
          getBall
    }
Rule 2 : A simple decision rule.
```

```
proc(preventCrowdingControl, prioritized_interrupts(
        [
        interrupt(play_mode = before_kick_off,
                                   [
                                    scanFieldWithBody,
                                    wait
                                   ]),
        interrupt(and(ballPositionValid(X), X = false),
                                   [
                                    scanFieldWithBody
                                   ]),
        interrupt(and(play_mode = play_on,
                      ballKickable),
                                   [
                                    kickStraight
                                   ]),
        interrupt(and(play_mode = play_on,
```

[7]Note that this primitive involves kicking straight which is obviously not the same as kicking towards the goal.

```
                              myTeamWithBall),
                                         [
                                          senNumTeamMatesCloseToBall,
                                          if(numTeamMatesCloseToBall < 3,
                                                 getBall,
                                                 debug('****RUNFREE****'))
                                         ]),
              interrupt(play_mode = play_on,
                                         [
                                          getBall
                                         ])
              ])
          ).
```

Code Listing 2: Adding sensing capability to Minimal Deliberation.

The above code snippet realizes **Rule 2**. Note that a new complex condition that checks for the validy of the ball position has been added. This is different from **Rule 1** which implicitly incorporated the condition into one of its primitives. As such, this modification fails to have any behavioral impact to the result produced by **Rule 2**. Also, note the use of the sensing action '**senNumTeamMatesCloseToBall**' prior to using the fluent **'numTeamMatesCloseToBall'** in the conditional construct. Invoking the sensing action prior to using its linked fluent in such a manner has at least two advantages:

1. Every time the fluent is used in a conditional construct, the most recent/accurate state of affairs (pertaining the fluent) will be taken into account.

2. Repeated invocations of the sensing action are not necessary to keep the fluent value updated. This can save a lot time execution time.

The result produced by this rule can bee see in illustration 2. The agents consistently maintain proximity to each other while 2-3 of them from each team attempt to get the ball and kick it straight. The resulting behavior can be primarily attributed to the sensing capability that was incorporated into **Rule 2**.

6.3 Extending the Domain Theory and Behavior Specification

Rule 1 and 2 primarily served a demonstrative purpose. The intent there was to exhibit the high-level nature of an Indigolog program for our soccer domain. We start the next experiment with the following two goals in mind:

1. **Soccer Playing Behavior**: Encode control rules such that a reasonably good soccer playing behavior emerges. We consider (at least at this

point) that the players should exhibit basic collaborative capabilities. For example: The players should adopt situation specific attacking or defensive strategies. Passing the ball between team mates or opportunistically kicking directly towards the opposition goal are essential primitives that might lead to such a behavior.

2. **Stability**: The above mentioned behavior should persist for most part of the simulation. Although short bursts of undesirable behavior, such as crowding, cant be avoided, it is necessary that the team quickly recover from such situations.

The perception & action related capabilities of each agent are limited. Besides, there is a high level of noise in the system. The state of the world, as believed by the agent, rarely has a one-to-one correspondence with the actual state of affairs. As such, we find it appropriate at this stage to mostly make use of the agents sensing capabilities and let almost all planning to be contingent on complex conditions involving the sensed fluents. The control procedure resulting out of **Rule 2** shall be the basis of further extension. The extension involves further enrichment of both the domain theory and behavior specification.

6.3.1 Extending the Primitive Skill Set

The following primitives will be utilised in the current extension:

- **Kicking to Goal**: Rule 2 involved a primitive for kicking straight. Kicking at a particular angle is actually available as a general primitive. Here, we use a specialisation of it, called *kickToGoal*, that involves kicking at an angle that matches the center of the opponents goalpost.

- **Dribbling the ball**: Dribbling being a continuous action cannot be directly implemented using the Indigolog language. As such, we make use of what we call a '*oneStepDribble*'. oneStepDribble is also defined to be a sensing action, associated with a fluent that keeps track of the result of the oneStepDribble act. Depending upon the sensed result after performing the action, the oneStepDribble act may or may not be repeated again. Apart from the sensed result, further dribbling is also contingent on the satisfiability of certain situation specific conditions, for example: It may no longer be safe to continue to dribble because of the presence of too many opponents around.

- **Go to/near a particular position on the field**: A general primitive for going to a particular position on the field is available. For now, we use four specializations of it, viz - goNearBall, gotoOpponent(Opp), goNearGoal(ourGoalPost) & goNearGoal(theirGoalPost).

6.3.2 Adding Sensing Actions and Associated Fluents

It is imperative that the agent possess the most accurate knowledge of the important domain fluents that would assist it in making strategic decisions. As stated earlier, sensing will be extensively used in the current extension to meet this end. The following sensing actions along with their associated fluents will be utilized:

- Sense the number of teammates/opponents near to a given goal post.
- Sense the number of teammates/opponents around the agent.
- Sense the number of teammates/opponents close to the ball.

6.3.3 Complex Actions and Conditions

The complex actions we make use of in this extension mostly do the job of gluing together of sensing actions and actions that depend on sensed fluents. Note that other high-level features supported by Indigolog, such as non-deterministic choice points etc, have not been utilized.

- **Go to the closest Opponent**

```
proc(gotoClosestOpp,
        [
         senClosestOpp,
         ?(X = closestOpp),
         goNearOpp(X)
        ]
).
```

- **Pass ball to teammate which the agent believes to be closest to the oppositions goal post.**

```
proc(passToClosestToGoal,
        [
         senClosestMateToOppGoal,
         ?(X = closestMateToOppGoal),
         passTo(X)
        ]
).
```

- **Pass ball to the closest team mate.**

```
proc(passToClosestMate,
        [
         senClosestMate,
         ?(X = closestMate),
         passTo(X)
        ]
).
```

The following self-explanatory complex conditions have been used to complement the above complex actions.

- **Is defense in place**?

```
proc(isDefenceInPlace,
        or(and(swiBallInTheirHalf,
                numMatesNearOurGoal >= 2),
           and(not(swiBallInTheirHalf),
                numMatesNearOurGoal >= 4))).
```

- **Am I very close to opposition goal**?

```
proc(veryCloseToGoal,
        and(swiBallInTheirHalf,
                or(swiDirectKickCrit, numOppNearTheirGoal =< 2))).
```

6.3.4 Adding More Control Rules

Finally, we are ready to design the procedural part of the agent; the control rules that would govern the ultimate soccer playing behavior. Limiting our scope to the previously mentioned aim, we consider the following three rules as possible extensions to **Rule 2**[8]:

```
interrupt(and(play_mode = play_on,
             and(amIDribbling = true, ballKickable)),
                [
                 senNumOppNearGoal(theirGoalPost),
                 if(veryCloseToGoal,
                    [
                     kickToGoal
                    ],
                    [
                     oneStepDribble
                    ])
          ])
```

- **Rule 3 - When to Dribble**: The agent has the liberty to dribble the ball towards any other point (the opposition goal post for now) only when it is safe to do so. Moreover, the ball should be kickable in the first place. We regard any situation in which the agent is not surrounded (in a certain circular region around the agent) by even one opponent as a safe situation to dribble the ball. Because dribbling is a continuous action, its execution needs constant monitoring. Also, at any given stage, if a direct kick to goal becomes feasible, the respective action must be executed, in effect resulting in termination of the continuous dribbling act. Note that

[8]From now on, we refrain from illustrating the associated pseudocode.

such a formulation of dribbling is more manipulative rather than natural. In reality, there is no built-in notion of a continuous action within the formalism.

- **Rule 4 - Going to defensive/offensive positions**: Any position within a certain area of the agent's or his opponents goalpost will be considered as a defensive or offensive position respectively. The notion of a defense/offense being in place is situation dependent and will be subject to certain complex conditions pertaining to determining which team holds the ball, Is the ball in the agents half or the opponents half etc.
- **Rule 5 - When to pass and who to pass**: An extremely simple passing rule will be utilised for the purposes of this experiment. If the agent holds the ball and there is an opponent very close-by, the agent considers passing the ball to the closest mate provided the mate is well away from the agent to avoid passing back by the receiving agent. This is necessary as the receiving agent (upon reception) does not know that the ball was received as a part of a passing act.

6.4 Implementing a Defensive Goalie

Notice that until now, we have only concentrated on controlling the players. The aim was to make the soccer agents get the ball up to the opponents goal post in a strategic manner. As our primary intent has been satisfied, we now focus on programming the goalie. To begin with, we intend to implement a goalie with completely defensive traits. Our defensive goalie's foremost concern shall be to keep the ball as far away from the goal post as possible. In addition to any previously mentioned primitive actions that may be required, the following makes up the skill set of the goalie:

- **goalieCatch**: This action is successful if the ball is in the goalie's catchable area.
- **gotoHomePosition**: Go to the center of the goal post.
- **getBehindBall**: Go to a co-ordinate position lesser than that of the ball's current location. This primitive takes care of issues pertaining to maintaining boundary regulations.
- **goalieKick**: We currently use a very crude form of a goalieKick whereby the goalie simply kicks the ball towards the opponents goal post with full power. Ideally, a goalie's kick should be based various strategic conditions pertaining to the position of teammates as well as opponents in order for the goalie's teammates to retain control of the ball after the kick is executed.

```
proc(goalieControl, prioritized_interrupts(
        [
        interrupt(play_mode = before_kick_off,
                                 [
                                  scanFieldWithBody,
                                  wait
                                 ]),
        interrupt(and(play_mode = play_on,
                        ballCatchable),
                                 [
                                  goalieCatch,
                                  goalieKick
                                 ]),
        interrupt(and(play_mode = play_on,
                        ballKickable),
                                 [
                                  goalieKick
                                 ]),
        interrupt(and(play_mode = play_on,
                       and(ballPositionValid(X), X = true)),
                                 [
                                  if(safeGoingHome,
                                   gotoHomePosition,
                                  [
                                   if(isBallBehind,
                                    getBallBehind
                                    faceBall)
                                  ]),
                                 ]),
        interrupt(play_mode = play_on,
                                 [
                                  scanFieldWithBody
                                 ]),
        interrupt(true,
                                 [
                                  wait
                                 ])
        ])
    ).
```

Code Listing 3: Defensive Goalie Control.

Currently, the goalies follows a simple control rule as depicted above in code listing 3: Every time it catches the ball from the catchable area on the field, it almost instantaneously kicks it away from the goal post. When the ball is not catchable, it quickly moves onto its home position if the ball is pretty far off or else tries to move to a co-ordinate (only the X-coordinate is important here) position lower than that of the ball. Note that like other players, the goalie has to constantly scan the field in order to utilise the most recent state of affairs.

7 Evaluating Indigolog

- **Expressiveness**: Indigolog features all the high-level constructs of imperative languages. For example, **IF**...**ELSE** statements, **WHILE** loops, boolean operators **AND**, **OR**, **NOT**. Besides the fact that iteration in Indigolog is non-deterministic, it also supports non-deterministic selection of actions & action arguments using the Π operator and a special construct for prioritized execution. However, the difference between Indigolog and conventional imperative languages is that the basic statements in Indigolog are domain dependent actions that are performed by the agent and conditions are based on fluents that reflect the agents belief about the world. Indigolog scores very high in this regard as it supports all the features deemed essential of a language that ought to provide high-level control of deliberative agents.

- **Efficiency**: Currently (at least in the public domain), only prolog based experimental versions of the Indigolog interpreter are available. For Indigolog to be used in a real world (and real-time) application, efficient implementations based on production languages (such as C++) must be developed. The prolog based interpretor used in this work severely restricts the relative capabilities of the soccer agents when compared to other teams participating in RoboCup.

- **Agent Behavioral Models**: The use of Indigolog in this work may wrongly suggest that it is only possible to implement deliberative behavior models using it. Our experience with implementing Soccer agents using Indigolog suggests that both deliberative as well as reactive behavioral models are supported. For example, it is possible that in addition to the main control procedure, all complex procedures may consist of prioritized interrupts. We believe that such a interrupt driven execution driven by exogenous actions resembles a reactive program.

- **Representing Static Aspects**: Fluents may be used to denote the state specific properties of the domain. However, there must also be a consistent mechanism to characterise the static aspects of the domain. For example, the predicate way(station_A, station_B) may be used to denote the fact that there exists a way between station_A and station_B. This is true in all situations that may arise and therefore is not a candidate for definition as a primitive fluent in the program. Currently, such definitions are defined directly in the language of predicate calculus and reasoned upon using the first-order capabilities of Prolog. However, this has the affect that we couple the use of Indigolog with that of prolog. Clearly, this is not desirable and there needs to be a standard approach for such representational issues.

- **Code clarity**: The high-level constructs supported in Indigolog facilitate easy understanding. Currently, since the language is implemented using prolog, some familiarity with prolog is also necessary for a in-depth understanding of the program. Our experience suggests that it is even possible for someone with minimal understanding of Indigolog/Golog semantics to understand most of the high-level Indigolog code. This is necessary as clarity of code goes a long way in attracting new developers towards the language.

8 Primitive Skills Library

The primitive skills library in use, Robolog, is a initiative of the AI research group at University Koblenz [24]. Robolog is both the name of the soccer team hosted by the group in the simulation league of RoboCup as well as the name of a primitive skills library implementing a subset of the interface provided by the soccer server.

The Robolog skills library is just a small part, albeit a crucial one, of a much larger effort aimed at combining logical and procedural techniques for the specification and implementation of multi-agent systems [25]. Indeed, the high level programming of agents requires the careful implementation of low-level skills as well as the possibility of abstracting quantitative sensor data onto a qualitative level, such that more general qualitative reasoning becomes feasible.

For our purposes, the skills library represents the procedural component aimed at facilitating high-level qualitative reasoning involving perception and action execution. As the name implies, the library provides a set of domain specific implementation of skills/primitive actions out of which complex high-level behavior may be constructed . Apart from that, it also provides primitives facilitating perception and sensing of the environment. The library is implemented in C++ heavily utilises the low-level skills of the CMUnited-99 simulator team [25].

9 Interfacing SWI-Prolog and C

9.1 SWI-Prolog Foreign Language Interface

One of the main reasons behind the use of SWI-Prolog for our system is the powerful foreign language interface it offers. It provides a very flexible and efficient interface to the C language [26]. Arbitrary low level functionality can be implemented in the C language and be made easily accessible to a prolog program via its foreign language interface. This mechanism has been used to achieve all low level initialization and communication with the soccer server. What follows is a brief introduction to the concept of a foreign predicate and the current setup involving the Robolog library, SWI-Foreign interface & Indigolog. This information will be helpful for future efforts aimed at making use of the system and possibly extending it in different ways.

9.2 Foreign Predicates

A foreign predicate is a C-function with same number of arguments as the predicate represented. C-functions are provided to analyse the passes terms, convert them to primitive C-types as well as to instantiate arguments using unification. Every C-function intended to serve as a predicate must be registered with the system. Registration basically involves exporting a foreign name for the predicate under consideration (so that the same may be used from prolog) and other low chores such as specifying the predicate arity etc. Details can be found in [26]. The exported set of predicates is what constitutes yet another layer in the system.

Currently, every registered predicate is simply a wrapper over one of the primitive skills provided by the Robolog library (Section 8). This might obviously not hold in future versions as the interface is extended to include ancillary predicates[9]. The advantage of such a layered design is two-fold; It ensures a clear separation of various levels of abstraction as well as helps provide improved maintainability and extensibility.

Extensions to the low-level capability of the system would primarily consist of complementing the primitive skills library. Following are a few guidelines for doing the same:

- **Extending Robolog skill set**: The skill library does not implement the complete interface provide by the RoboCup Soccer Server. Also, because Robocup is a on-going initiative, there are frequent additions to server capabilities/interface. Extending the skill set would simply involve implementing the required code (following the soccer server protocol) so that

[9]For example: Utilities implemented directly in C++.

the same can be made available to Indigolog soccer agents in the higher level layers of the system[10].

- **Providing Wrappers in the Foreign Interface**: For each of the skills added to the skill set, there has to be a registered foreign predicate in the SWI-foreign interface.

9.3 Communication between Prolog and C

SWI-Prolog facilitates two-way, nested communication between C and prolog: i.e. Prolog can call C, C can call prolog etc up to a arbitrary depth. There is no limit to the amount of nesting in the communication. Currently, all the communication is prolog initiated and the only time a C - to - Prolog communication occurs is during system startup when the embedded prolog engine (Section 10) needs to be initialised. At all other times, all communication is initiated by prolog and basically involves queries to get the world state or send action execution commands.

10 Embedding SWI-Prolog as a Logical Engine

Prolog is generally used as a part of some other larger application so as to make use of its reasoning capabilities. This is referred to as ***'embedding'*** prolog in that application. This section discusses some of the issues related to *'embedding'* with regard to our system, its need in the current framework and how it affects the overall design.

10.1 The Need for Embedding

The primary purpose of prolog is to provide automated reasoning capabilities at a level of abstraction not supported by other imperative languages generally used for production purposes. Embedding is not strictly necessary for our system. The other option would have been to to use prolog directly and make use of dynamic linking to connect to the foreign interface. However, (as it was correctly hypothesized) embedding greatly enhanced the overall development and testing process besides facilitating easy distribution for demonstration purposes.

10.2 Using PLLD

Plld is a utility that may be used to link a combination of C and Prolog files into a standard standalone executable . plld basically automates the creation of a single image consisting of some application program, the prolog kernel and a prolog program. Note that all the three components are statically linked to create a self-complete executable.

[10]Obviously, because ROBOLOG skills layer is completely independent of other layers, its usage will not be restricted only to Indigolog Soccer Agents.

10.3 The Result of embedding

From a implementation viewpoint, our system can be thought of as consisting of two main components; One consisting of the low-level primitive skills library & the associated SWI foreign language interface, both implemented in C++ , and the other consisting of the prolog reasoning engine & the agent program implemented in prolog. For purposes of embedding, we simply use a third stub program that acts as a glue between the two components and feed it in to the plld utility. The result is a statically linked composite program that makes up our system.

11 Program Preprocessor

Typographical errors and/or omissions in the Indigolog program specification can go unnoticed because of the unavailability of program pre-processing facilities. For exam, a primitive action given by '*prim_ action(gotoStaticBall)*' and erroneously used as '*gotStaticBall*' (or anything else) elsewhere in the program may simply cause the Indigolog interpreter to simply fail on the respective action execution and formulate an alternative plan. This will result in undesirable behavior that is hard to track down or probably identify in the first place in our fast moving soccer environment. A similar argument applies to other aspects of the program such as presence of initial values for all fluents, preconditions of actions etc.

A program pre-processing module has been implemented for each of the Indigolog, BDI and STRIPS programs. This module can successful detect & report syntactic errors (by checking for ambiguities between definitions and actual usage or references elsewhere in the program), missing axioms etc. The following are some of the checks made by the pre-processing module:

- All defined actions have a associated action number;
- All actions have associated add lists;
- All actions have del lists;
- All fluents used in the add/del lists are a subset of those actually defined;
- All possible actions are previously defined;
- All actions have valid pro-conditions;
- All triggering events have associated plans;
- Plans defined have a valid body;
- All fluents have a initial value etc.

Note that some of the above checks may be language specific.

12 Related and Future Work

12.1 Related Work

There is (at least) one related approach that employs logic for controlling agents in the robocup domain, namely the Robolog team at University Koblenz [25]. The Robolog Koblenz team focuses on a declarative approach for controlling agents. Agents are implemented in a logic-and rule-based manner, thereby directly using the first-order reasoning capabilities of Prolog. State machines, represented using state charts, are employed to specify the procedural aspects of the agent.

Note that this approach does not utilize any formal approach for reasoning about action. Behavior is completely based on pre-defined scripts and decision rules that are implemented in Prolog. As such, explicit planning and extrapolation of future behavior, which is so important from a artificial intelligence viewpoint, is indeed not possible using this approach. Our approach is more formal, by making use of the extended situation calculus based formalism for representing the dynamics of soccer domain. Execution using our approach is defined as a real-time planning task that is based on a well-defined action formalism.

12.2 Conclusion and Future Work

Our research was based on two main objectives: **(a)**. *To overcome the limitations posed by reactive approaches to agent control* and **(b)**. *To achieve the first objective by using a mathematically rigorous action representation formalism.* We achieved these objective by employing a deliberative control mechanism comprising of implementing agent programs in the high-level cognitive robotics language, Indigolog. Indigolog is similar to other imperative languages with the difference that the basic statements of the language consist of domain dependent actions and tests of predicates. The program consists of a domain theory and behavior specification that is used by its interpreter to reason about the effects of actions and make decisions in the real or simulated world. In order to perform reasoning, the interpreter itself uses the situation calculus, which is a first order language (with some second order features) for representing knowledge pertaining to dynamically changing domains.

The *event-driven* behavior of reactive agents is not suitable for long-term *goal oriented planning*. Because of the absence of a symbolic world model & knowledge of the causal laws of the domain, reactive agents cannot be utilised in realistic, dynamic environments such as the robocup soccer simulator. Decision making in such environments must utilise domain knowledge, as exemplified by our use of Indigolog, so as to reason about *action and their effects*. Moreover, such a knowledge representation scheme must be based on a sophisticated *logic* *of action*[11] as has been done in this work.

Our work represents a first attempt at utilizing high-level reasoning, based on a formal theory of action, for controlling agents in the soccer domain. The Indigolog based domain theory and behaviour specification currently being used are not extensive. Moreover, the complete set of available primitive skills and sensory capabilities provided by the soccer simulator have not been utilised. The programs need to be further extended in order to be able to compete in the RoboCup tournament. Future work in this regard could involve further refinement and/or extension of the strategies being used by the soccer agents. Also, completely new strategies for collaborative game play may be implemented.

The primitive skills library itself may be extended to account for more primitive skills out of which high-level behaviour may be constructed. High level reasoning may be complemented by utilizing *machine learning* capabilities at the lower primitive skills level. Such a hybrid approach seems interesting enough to be given due consideration as it represents a merger between symbolic and non-symbolic approaches to artificial intelligence.

[11]Formalisations based on approaches other than logic are indeed possible. However, discussion pertaining to them is beyond the scope of this research.

13 Software Used

- **Indigolog Interpreter/Planner**: Publicly available from http://www.cs.toronto.edu/cogrobo/Legolog
- **STRIPS Planner**: A simple planner based on the STRIPS formalism implemented using goal-regression. Customised version as used in this work *available upon request.*
- **AgentSpeak(L) Planner**: *Available upon request.*
- **Delivery Agent Code (STRIPS)**: *Available upon request.*
- **Delivery Agent Code (BDI)**: *Available upon request.*
- **Delivery Agent Code (Indigolog)**: *Available upon request.*
- **Soccer Agent Code**: *Available upon request.*
- **Language Pre-Processor**: *Available upon request.*
- **SWI-Prolog**: Freely available from http://www.swi-prolog.org. Provides a very flexible foreign language interface.
- **Robolog Library**: The Robolog Primitive skills Library publicly available from http://www.robolog.org. However, a modified version as used in this work is *available upon request.*
- **Robocup Soccer Server**: Available on sourceforge.
- **Soccer Monitor**: Available at sourceforge.
- **LogPlayer**: Available at sourceforge. Very useful for debugging purposes.
- **Robolog2Flash**: A tool to create Macromedia Flash (SWF) files from version 3 soccer logfiles. Freely available from http://www.robolog.org.
- **LyX**: A front-end to latex based on the 'What You See Is What You Mean' philosophy. Available on source-forge.
- **Pybliographic**: Manage your bibliography, In Style. Freely available on SourceForge.

References

[1] John McCarthy and Patrick J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969. reprinted in McC90. 1, 2.3, 2.3.1

[2] Michael Thielscher. Ramification and causality. *Artificial Intelligence*, 89(1-2):317–364, 1997. 1

[3] Michael Thielscher. Causality and the qualification problem. In Luigia Carlucci Aiello, Jon Doyle, and Stuart Shapiro, editors, *KR'96: Principles of Knowledge Representation and Reasoning*, pages 51–62. Morgan Kaufmann, San Francisco, California, 1996. 1

[4] Raymond Reiter. *Knowledge in action: Logical foundations for describing and implementing dynamical systems*. MIT Press, 2001. 1, 1.3, 2.3.1

[5] Paolo Torroni. Logics and Multi-agent systems: Towards a new symbolic model of cognition. *Electronic Notes on theoritical computer science*, 70(5), 2002. 1

[6] Hector J. Levesque, Raymond Reiter, Yves Lesperance, Fangzhen Lin, and Richard B. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1-3):59–83, 1997. 1, 1.3, 2.3, 2.3.1

[7] Jaakko Hintikka. *Knowledge and Belief.* Cornell University Press: Ithaca, NY., 1962. 1.1.1

[8] Michael Wooldridge and Nicholas R. Jennings. Intelligent agents: Theory and practice. HTTP://www.doc.mmu.ac.uk/STAFF/mike/ker95/ker95-html.h (Hypertext version of Knowledge Engineering Review paper), 1994. 1.1.1, 1.2.1, 1.2.3

[9] Michael Wooldridge and Nicholas R. Jennings. Agent theories, architectures, and languages: A survey. volume 890, pages 1–32. 1994. 1.1.1

[10] Michael Bratman. Intention, plans, and practical reason. Harvard University Press, 1987. 1.1.2

[11] Anand Rao and M. P. Georgeff. Modeling rational agents within a BDI-architecture. In J. Allen, R. Fikes, and E. Sandewall, editors, *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning*. Morgan Kaufmann Publishers, San Mateo, CA, 1991. 1.1.2, 2.2

[12] M. Namee and B. Cunningham. A proposal for an agent architecture for proactive persistent non player characters, 2001. 1.2.1

[13] Rodney Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2(1):14–23, 1986. 1.2.1

[14] Gernhard Weiss, editor. *Multiagent systems: A Modern approach to distributed Artificial Intelligence*. MIT Press, 1999. 1.2.1

[15] Ronald Brachman and Hector Levesque. *Knowledge Representation and Reasoning*. In Press. 2.1, 2.1, 2.1

[16] Anand Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In Rudy van Hoe, editor, *Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, Eindhoven, The Netherlands, 1996. 2.2, 2.6

[17] Giuseppe De Giacomo, Hector J. Levesque, and Sebastian Sardina. Incremental execution of guarded theories. *Computational Logic*, 2(4):495–525, 2001. 2.4

[18] H. Levesque and M. Pagnucco. Legolog: Inexpensive experiments in cognitive robotics, 2000. 2.4

[19] James Firby. Adaptive execution in complex dynamic worlds. Technical Report RR-672, 1989. 2.5

[20] Ivan Bratko. *PROLOG programming for Artificial Intelligence*. Adison Wesley, s econd edition, 1990. 2.6

[21] Hiroaki Kitano, Minoru Asada, Itsuki Noda Yasuo Kuniyashi, and Eiichi Osawa. Robocup: The Robot World Cup Initiative. 1995. pages 19-24. 4

[22] Mao Chen, Ehsan Foroughi, and et al. The robocup soccer server manual, ver 7.07 and later. 4, 4.2

[23] Peter Stone. Multiagent competitions and research: Lessons from robocup and tac. 4.1

[24] Robolog. Robolog simulation league at University Koblenz. http://www.uni-koblenz.de/ag-ki/ROBOCUP/ABOUT/about.html. 8

[25] Jan Murray, Oliver Obst, and Frieder Stolzenburg. Towards a logical approach for soccer agents engineering. In Peter Stone, Tucker Balch, and Gerhard Kraetzschmar, editors, *RoboCup 2000: Robot Soccer World Cup IV*, pages 199–208. Springer, Berlin, Heidelberg, New York, 2001. 8, 8, 12.1

[26] Jan Wielemaker. SWI-Prolog Reference Manual. http://www.swi-prolog.org, March 2003. 9.1, 9.2