Formalizing Properties of Agents

Richard Goodwin

May 1993

CMU-CS-93-159

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

There is a wide gulf between the formal logics used by logicians to describe agents and the informal vocabulary used by people who actually build robotic or software agents. In an effort to help bridge the gap, this report applies techniques borrowed from the field of formal software methods to develop a common vocabulary. Terms useful for discussing agents are given formal definitions. A framework for describing agents, tasks and environments is developed using the Z specification language. The terms successful, capable, reactive, reflexive, perceptive, predictive, interpretive, rational and sound are then defined in terms of this framework. In addition, a hierarchy for characterizing tasks is given. The aim of this report is to develop a precise vocabulary for discussing and comparing agents.

This research is supported in part by a Natural Sciences and Engineering Research Council of Canada 1967 Science and Engineering Scholarship. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Natural Sciences and Engineering Research Council of Canada.

Keywords: agents, tasks, environments, robotics, formalisms, successful, capable, perceptive, reactive, reflexive, predictive, interpretive, rational, sound

Introduction

In the artificial intelligence community, it has become common to talk about agents that perform tasks and to describe such agents in terms of characteristics that would allow them to be successful. Common terms include: successful, capable, perceptive and reactive. Each of these terms has an intuitive meaning that allows for informal discussion of the suitability of an agent for a task. The problem with these informal definitions is that they are often ambiguous and incomplete. What is needed are more precise definitions that will help to establish a common, uniform vocabulary for talking about agents, environments and tasks.

This paper formally defines some useful terms for discussing agents. First, a general framework for describing agents, tasks and environments is developed using the Z specification language [Spivey, 1989]. This framework is useful for analyzing agents and communicating ideas, but is not intended to suggest representations or algorithms for implementing agents. Through developing the framework, a number of key issues are identified. Agent properties are then defined in terms of this general framework.

Overview

An agent is an entity created to perform some task or set of tasks. Any property of an agent must therefore be defined in terms of the task and the environment in which the task is to be performed. The most basic question that can be asked is whether the agent achieves the task or not. Is the agent successful? Other properties concern the suitability of various components of the agent and can be defined in terms of success.

The operational problem is how to determine if an agent is successful. The approach adopted here is to take the lead of the behavioural psychologists and make observable behaviour the only criteria allowed for determining success [Skinner, 1974]. Such an approach allows us to avoid having to make inferences about the agent's beliefs, intentions or motives. Using a behaviourist approach does limit the types of tasks that can be considered. Any task where success depends on inferring or interpreting the agent's internal state are ruled out. Tasks of this type would have to be modified so that any internal state of interest is manifested as external behaviour. For example, suppose the task were to go outside and determine if it is raining. Even if we observed the agent going outside and getting wet, there is no way to know whether the agent believes that it is raining. To make the task operational, it could be modified to have the agent open its umbrella, if it is raining. To determine whether the agent is successful, we only have to observe its behaviour. If it opens the umbrella at the appropriate times, it is successful. We don't have to ascribe any belief about rain to the agent.

Since success is the fundamental property for characterizing agents, some appropriate scale is needed for determining success. The simplest scale that can be used is binary: An agent is either successful or not. A binary scale for success leads to binary definitions of other properties. For instance, an agent is either capable or not and there is no way to compare relative capability of agents. The scale for measuring success can be refined to give a numeric value to the quality of task achievement. In economics, such a numeric measure of quality corresponds to a utility function defined over the possible task completions [Raiffa, 1968]. Further, the likelihood of possible initial conditions and external influences can be taken into account to give a measure of the expected success of the agent. Refining the measure of success allows corresponding refinements in agent properties defined in terms of success.

Although a behaviourist approach is being adopted for measuring success, it is sometimes useful to consider the internal organization of the agent and the extent to which that organization leads to behaviour that results in success. A particularly interesting class of agents are those that maintain some internal model of the world and reason about the results of their actions. Such *deliberative* agents attempt to choose actions that are more likely to lead to success based on their world models. Additional properties for a deliberative agent relate to how accurate the agent's models are and whether the agent behaves consistently with its models.

Informal Definitions

The informal definitions given in Figures 1 and 2 are derived from the set of characteristics that Mitchell believes are necessary for a successful agent [Mitchell, 1990] [1]. These informal definitions provide the intuitive meaning for the agent properties that will be more precisely defined in the rest of this paper.

Issues

Before delving into the details of the definitions, we present a set of issues that any specification in this area must address. Then, with these issues in mind, we will present the approach that we have taken.

[1]Contains references to *Reactive, perceptive and correct*. Some additional terms were described through personal communication.

Successful :	An agent is successful to the extent that it accomplishes the specified task in the given environment.
Capable :	An agent is capable if it possesses the effectors needed to accomplish the task.
Perceptive :	An agent is perceptive if it can distinguish salient characteristics of the world that would allow it to use its effectors to achieve the task.
Reactive :	An agent is reactive if it is able to respond sufficiently quickly to events in the world to allow it to be successful.
Reflexive :	An agent is reflexive if it behaves in a stimulus-response fashion.

Figure 1: **General Agent Properties**

Predictive :	An agent is predictive if its model of how the world works is sufficiently accurate to allow it to correctly predict how it can achieve the task.
Interpretive :	An agent is interpretive if can correctly interpret its sensor readings.
Rational :	An agent is rational if it chooses to perform commands that it predicts will achieve its goals.
Sound :	An agent is sound if it is predictive, interpretive and rational.

Figure 2: **Deliberative Agent Properties**

Any formal definition of agent properties must include a framework for describing an agent, a task and an environment. To say that an agent is successful at a particular task in a particular environment, some method is needed for specifying the agent, the task and the environment. The framework for these descriptions must be sufficiently detailed to allow the distinction between successful and unsuccessful agents to be made. At the same time, the framework must be general enough to permit a wide range of agents, tasks and environments to be specified. In designing the framework, the range of possible tasks and environments needs to be considered. The kinds of tasks can include tasks of achievement, tasks of maintenance and tasks with deadlines. The environment can be static or dynamic, can include other agents and can be non-deterministic. In addition, we also have to consider how to represent the interaction of the agent with the environment over time.

One aim of this specification is to be able to compare different agents performing the same task in same environment. Clearly, to compare agents, we must distinguish between the agent and the environment. For a physical agent, such as a robot, the distinction between the agent and the environment is reasonably clear, although, this is not always the case. What happens if a part of the robot breaks off? Is this broken piece still part of the agent? Furthermore, is the resulting robot the same "agent" as before? For software agents, the distinction between the environment and the agent may be harder to make. Fortunately, this question is mostly of academic interest. In general, where to draw the line between the agent and the environment will be clear. The environment is everything present before the agent was built and the agent is everything that was added.

Of more concern is whether and where to make distinctions within the agent, since such distinctions impact the definitions of agent properties more directly. One extreme possibility is to regard the agent as a black box. This is sufficient for defining some properties of an agent, but not others. In particular, there is no way of describing the suitability of the agent's effectors or sensors for a task separately from the entire agent. The distinctions that should be made are those that are useful for comparing and designing agents. Distinctions should also be sufficiently abstract as to be able to apply to a wide range of agents.

Given distinctions between the environment and the agent, and between the sub-parts of the agent, a method is needed to describe how these components interact. The agent affects the environment through its actions and is affected by changes in the environment. Sub-parts of the agent interact either physically or by passing information. These interactions should be characterized to the level of detail necessary to make useful comparisons between behaviour.

When performing a task, the interactions between the agent and the environment occur over a period of time. So, in addition to representing the form of the interaction, the evolution of the interaction over time must also be represented. Time can either be modeled as a continuous quantity or as discrete intervals. Of course, as the size of each time interval shrinks, a discrete representation better approximates the continuous quantity.

Finally, we need to consider whether our model of the environment will include non-determinism. We could adopt the Newtonian view that the world is deterministic, if only we know its state in infinite detail. Any apparent non-determinism is the result of our limited information about the current state and our limited ability to compute future consequences.

Neglecting computational limits, the environment may still be non-deterministic if we allow for other agents with free will. Either we can distinguish other agents from the environment and model them as non-deterministic entities or leave them as part of the environment and include non-determinism in the environment. If our aim were to define properties of co-operating agents, then the distinction between other agents and the environment is needed. Since this is not our current objective, the distinction between other agents and the environment will not be made in order to simplify the framework.

Framework

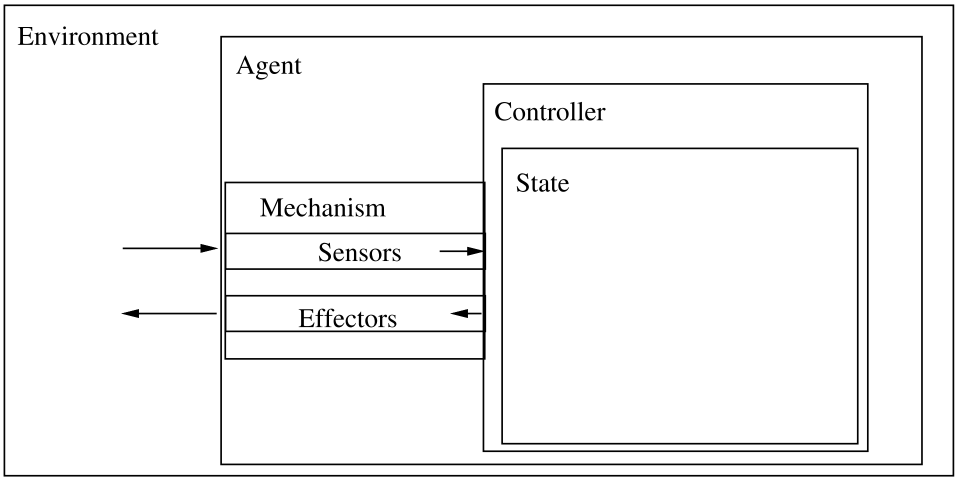

Figure 3: **Agent System Overview**

The general framework adopted is shown in figure 3. We make the necessary distinction between the agent and the environment in which it is situated. The environment will include non-determinism and time will be measured in discrete intervals. Within the agent, we distinguish the mechanism from the controller. The mechanism consists of the sensors and effectors that allow the agent to interact with the environment. The controller is, conceptually at least, the component that accepts input from the sensors and controls the effectors. The controller may maintain some state (memory). The details of the nature of the environment, the structure and implementation of the agent and the agent's control have been abstracted away.

The model is intended to apply to systems as diverse as autonomous robots, knowbots [2] and thermostats. The agent may have a physical presence or just exist within the memory of a computer. The control component could be implemented in software or hardware. By refining this abstract model, particular classes of agents can be specified.

This formal specification first defines the basic data types: an environment, a task and an agent. These components are then used to define the general agent system and a deliberative agent system. Binary definitions of the agent properties are given in terms of these agent systems. Following this, a utility function is defined over possible task completions. This allows the agent properties to be refined and stated as partial orders. When a probability distribution over the possible initial states is added, expected performance of an agent can be defined. Using expected performance, agent properties are further refined.

The formal aspects of our definitions are presented using the Z specification language [Spivey, 1989]. A short introduction to the language is given in appendix A. The specification consists of a prose description and Z code. These two components complement each other. The prose explains the specification in an intuitive way and gives the intentional meaning of the Z code. The Z code provides precise definitions in a language with well defined semantics. Developing the specification is a process of creating and refining both the text and the Z code. The Z code is created based on the written description. The process of writing the Z code often identifies inconsistencies and omissions in the original text. In addition, the Z code can be put through a syntax and type checker to identify more subtle errors. Updating the prose to conform to the revised Z code often highlights assumptions and implications that can be used to further refine the prose and the Z code.

It is intended that the basic idea behind each definition presented in this report should be understandable from the text alone, though not in full detail.

[2] A knowbot is an agent that exists as a program running on a computer or computer network. It performs tasks such as information retrieval from databases.

Primitive Data Types

In this section, the primitive data types used to describe agents, environments and tasks are presented and discussed. These data types are used to describe the state of each component and to describe interactions between components. In addition, we define the term *Chronicle* which is used to represent how components of the agent and environment change over time.

External Descriptions

The externally observable state consists of the state of the environment and the externally observable state of the agent. Each is described by a primitive data type.

$[WORLD, CONFIGURATION]$

A WORLD is a complete, instantaneous description of the environment. It includes the state of the environment as well the instantaneous rates of change of that state. It is like a very detailed snapshot of the environment.

A CONFIGURATION refers both to the composition of the agent's "body" and its current position. For instance, a CONFIGURATION specifies how many arms the agent has and the position of each arm. Like a WORLD, a CONFIGURATION is an instantaneous description that includes the instantaneous rates of change of the state.

A complete description of the external state of a system requires both a WORLD and a CONFIGURATION. An ExternalState is such a description. It is interesting to note that not all combinations of WORLDs and CONFIGURATIONs may be possible. The physical laws of the environment may restrict which pairs are legal. For example, two things cannot be in the same place at the same time. Each environment will determine which ExternalStates are possible.

$$
\begin{array}{|l}
\hline
\textit{ExternalState} \\
\textit{world} : \textit{WORLD} \\
\textit{configuration} : \textit{CONFIGURATION} \\
\hline
\end{array}
$$

It is impossible to completely specify the external state of the system in exact detail. From a practical standpoint, some of the details must necessarily be omitted. It is also desirable to abstract away irrelevant details. An abstract description of the external state can be represented by the set of ExternalStates that are consistent with the partial description. A predicate such as *On(A,B)* is equivalent to the set of

ExternalStates where the predicate holds. An ExternalDescription is defined to be a set of ExternalStates representing an abstract description of the external state.

$ExternalDescription == \mathbb{P}\ ExternalState$

Internal State

The controller within the agent may maintain some information in its memory. A STATE is defined to be a complete description of the internal state of the agent's controller. It represents the contents of the agent's memory. Note that since it is a primitive data type, nothing is said about the structure, capacity or organization of this memory.

$[STATE]$

Commands and Interactions

Interactions between the agent's controller and its effectors and between the agent's effectors and the environment are mediated by COMMANDs and INTERACTIONS, respectively.

$[COMMAND, INTERACTION]$

A COMMAND is a signal that the agent's control component can send to the agent's mechanism. The results of sending a particular COMMAND signal will depend on the CONFIGURATION of the Mechanism and WORLD in which it is sent. A COMMAND used in this sense is not the same as what is meant by an action in classical Artificial Intelligence planning. A COMMAND is not a high level action like *putOn(A,B)*. It is more like a setting of the switches that control the agent. A single configuration of the switches may cause the robot to begin moving forward and open its gripper.

To get a better idea of what a COMMAND is, imagine yourself as the controller inside the agent. In front of you is a control panel with some buttons. You control the agent by pushing combinations of these buttons. Each COMMAND corresponds to one combination of buttons you could push.

An INTERACTION is the influence the Mechanism exerts on the environment as the result of acting on a particular COMMAND signal. The exact INTERACTION depends on the Mechanism, the COMMAND signal and the CONFIGURATION of the Mechanism.

The advantage of having separate COMMANDs and INTERACTIONS is that it allows for the distinction between the signals sent to the mechanism and the effects the mechanism has on the environment. A COMMAND that turns on the arm lifting motor will produce a different INTERACTION depending whether the motor fuse is burnt out or not. There can also be more than one way of achieving the same INTERACTION. I can push a block with my right hand or my left hand. The effect on the block is the same, as long as the force applied is the same.

Perception

An agent receives information about the state of the environment through its sensors. A PERCEPT is a signal that the agent's sensors pass to the agent's control component. It is the input counterpart to a COMMAND. The PERCEPT sent can be influenced both by the WORLD and the CONFIGURATION of the agent's mechanism. For instance, the output of a camera will depend on the state of the world and the direction the camera is pointed in. Again imagine yourself as the controller inside the agent. The control panel has some lights on it indicating the current value of the sensor readings. These lights provide your only access to the information about the outside world. Each of the possible patterns of lights corresponds to a single PERCEPT.

$[PERCEPT]$

A PERCEPT encodes the output of all the sensors in a single value. When designing a particular agent, it may be helpful to distinguish the output from each sensor and to identify the possible values. A complete specification of sensor output would then consist of the value of the output for each sensor. When considering agents in the abstract, it is desirable to hide details about the number of sensors and the values they can return. Combining the individual values into a single composite value does not reduce the amount of information obtained from the sensors. The reason for having a single composite value is to make definitions dealing with sensor output simpler.

Time and Chronicles

In this specification, Time is measured in discrete intervals. For convenience, we mapped each interval to an integer, to keep track of the sequencing. A discrete time representation was chosen since it results in cleaner property definitions with little loss in generality. This specification does not depend on the size of the time interval. Nowhere is the time quanta specified. By making the time quanta sufficiently short, continuous time can be approximated to any desired granularity.

$Time == \mathbb{Z}$

The evolution of an entity, either a component or an interaction, over an interval of time, can be represented using a Chronicle [McDermott, 1982]. In our representation, a Chronicle consists of a series of samples of the entity. Each sample represents the value of the entity at the start of a time interval. Chronicles are well suited for modeling entities that change only at discrete times. A good example of such a entity would be the output of a controller implemented using a micro-processor. The controller's outputs would only change synchronously with the micro-processor's clock. For a continuously varying entity, a chronicle is only an approximate representation in the same way that the samples on an audio CD only approximate the original music. The critical factor in such an approximation is the sampling rate. The entity must be sampled often enough to prevent any required information from being lost.

Formally, a Chronicle is defined to be a partial function that maps times to corresponding values. All Chronicles have definite start times. In addition, we define FiniteChronicles to have finite end times as well. The definitions of Chronicle and FiniteChronicle are parameterized by the type of entity, Q. A particular kind of Chronicle is defined by specifying the type of entity. For example, a Chronicle of external states would be "Chronicle[ExternalState]".

$$Chronicle[Q] == \{c : Time \mathrel{+\!\!\!\rightarrow} Q \mid \exists\, t_0 : Time \bullet \operatorname{dom} c = \{t_1 : Time \mid t_1 \geq t_0\}\}$$

$$FiniteChronicle[Q] == \{c : Time \mathrel{+\!\!\!\!\!\rightarrow\!\!\!\rightarrow} Q \mid \exists\, t1, t2 : Time \bullet \operatorname{dom} c = t1 \mathinner{..} t2\}$$

Here we define six auxiliary functions for dealing with chronicles. They will be used to make formulas involving chronicles more concise and readable. The name of each function is meant to suggest its semantics. The timesOf function returns the set of times covered by the chronicle, while the startOf function returns the earliest time. For inductive definitions, it is convenient to be able to refer to the times covered by the chronicle, after the initial state. The restTimesOf function returns the times covered by the chronicle, minus the start time. It is also useful to be able to refer to the range of values covered in the chronicle. The valuesOf function returns the set of values for the quantity in the chronicle and the firstOf function returns the initial value. Finally, the prefixesOf function returns all chronicles which are prefixes of the given chronicle.

$$
\begin{array}{l}
[Q] \\
\hline
timesOf : Chronicle[Q] \longrightarrow \mathbb{P}\ Time \\
startOf : Chronicle[Q] \longrightarrow Time \\
restTimesOf : Chronicle[Q] \longrightarrow \mathbb{P}\ Time \\
valuesOf : Chronicle[Q] \longrightarrow \mathbb{P}\ Q \\
firstOf : Chronicle[Q] \longrightarrow Q \\
prefixesOf : Chronicle[Q] \longrightarrow \mathbb{P}\ FiniteChronicle[Q] \\
\hline
\forall\, c : Chronicle[Q] \bullet \\
\quad timesOf(c) = \mathrm{dom}\, c \land \\
\quad startOf(c) = min(\mathrm{dom}\, c) \land \\
\quad restTimesOf(c) = timesOf(c) \setminus \{startOf(c)\} \land \\
\quad valuesOf(c) = \mathrm{ran}\, c \land \\
\quad firstOf(c) = c(startOf(c)) \land \\
\quad prefixesOf(c) = \{prefix : FiniteChronicle[Q] \mid \\
\qquad\qquad startOf(prefix) = startOf(c) \land prefix \subseteq c\}
\end{array}
$$

Environment

Every agent operates in an environment. To describe an environment, we must characterize the valid states and how they change over time. In general, the state of an environment is dynamic, changing of its own accord and in response to interaction with the agent. In a deterministic environment, knowing the exact state, the interaction with the agent and the "laws of nature" allows the immediate future to be predicted exactly. In a non-deterministic environment, the best that can be done is to predict a set of possible future states. Any model of the environment must be able to account for both deterministic and non-deterministic state changes.

In this framework, changes in the state of the environment will be represented as a Chronicle of the external state. To characterize how the environment changes, we need only be able to predict the next state given the current state and the current interaction with the agent. This can be done if the Markov property holds. Such is the case if the state description of the environment captures all relevant the features. For example, to predict the flight of a ball thrown into the air, the state description needs to include the velocity of the ball as well as its position. The definition of WORLD and CONFIGURATION include the state and instantaneous rates of change of state so that the Markov property holds for an ExternalState.

Even with the Markov property, it is not possible to predict the next state for a non-deterministic environment. The best that can be done is to predict the set of possible next states and the probability of each one. When considering how to model non-determinism, there are a couple of issues to keep in mind. For one, the model should allow for a probability distribution over successor states. Secondly, the model should also allow the performance of agents to be compared under the same conditions.

The approach we will take is to model non-deterministic environments as deterministic environments with hidden state variables. To get an idea of how this works, imagine that at the beginning of time you ask an all knowing oracle to write down the outcome of all future non-deterministic events. You then include this information in the state of the environment. Of course, there can be no way that an agent could perceive this information. It must remain hidden until each non-deterministic event takes place and the outcome is revealed. There are advantages to modeling non-determinism in this way. It allows agents to be compared under identical conditions. How does each agent do if the coin flip comes up heads? Such an encoding also allows a single probability distribution over initial states to account for the likelihood of each state and the probability of each non-deterministic event. In practical terms, being able to determine the next state rather than a set of possible next states simplifies the specification without loss of generality.

The environment is represented by a schema that embodies the "laws of Nature." The schema includes a description of the valid combinations of WORLDs and CONFIGURATIONs, a function for specifying how things change over time and a partitioning of external states for hiding non-deterministic outcomes. As described previously, not all combinations of WORLDs and CONFIGURATIONs are compatible. The "consistent" set gives the set of legal pairs. The consequence function maps the current ExternalState and the current INTERACTION with the agent to the next ExternalState. The consequence function is defined for all INTERACTIONs and all consistent ExternalStates. The resulting ExternalState is always a consistent ExternalState.

It is not possible for an agent or an observer to determine the value of the hidden state variables. If it were, then the environment would be deterministic. The environment schema divides the set of world states up into sets of indistinguishable states that differ only in the value of hidden state variables. This partitioning will be used to enforce the restriction that hidden state remain hidden. An agent will be restricted to be able to, at best, determine which set of indistinguishable states it is in. A task specification must also not depend on the hidden state, in the same way that it cannot depend on the internal state of the agent.

$$
\begin{array}{|l}
\hline
\textit{Environment} \\
\textit{consistent} : \textit{ExternalDescription} \\
\textit{consequence} : (\textit{ExternalState} \times \textit{INTERACTION}) \rightarrowtail\!\!\!\!\!+ \textit{ExternalState} \\
\textit{indistinguishable} : \mathbb{P}\ \textit{ExternalDescription} \\
\hline
(\mathrm{ran}\ \textit{consequence}) \subseteq \textit{consistent} \\
\mathrm{dom}(\mathrm{dom}\ \textit{consequence}) \subseteq \textit{consistent} \\
\bigcup \textit{indistinguishable} = \textit{consistent} \\
\forall a, b : \textit{indistinguishable} \bullet a \cap b \neq \varnothing \Leftrightarrow a = b \\
\hline
\end{array}
$$

To assist in enforcing the restriction on indistinguishable states, we define a function that takes an environment and returns the set of pairs of indistinguishable states. The returned set consists of all pairs of ExternalStates that differ only in their hidden state.

$$
\begin{array}{|l}
\textit{sameExternalStates} : \textit{Environment} \longrightarrow \mathbb{P}(\textit{ExternalState} \times \textit{ExternalState}) \\
\hline
\forall \textit{env} : \textit{Environment};\ e_1, e_2 : \textit{ExternalState} \bullet \\
\quad (e_1, e_2) \in \textit{sameExternalStates}(\textit{env}) \\
\quad \Leftrightarrow (\exists \textit{ind} : \textit{env}.\textit{indistinguishable} \bullet e_1 \in \textit{ind} \wedge e_2 \in \textit{ind})
\end{array}
$$

External Chronicle

An ExternalChronicle is a Chronicle of the ExternalState. It describes the evolution of the environment and the behaviour of the agent.

$\textit{ExternalChronicle} == \textit{FiniteChronicle}[\textit{ExternalState}]$

In a particular environment, only some Chronicles are possible. To be valid, the sequence of ExternalStates must be consistent and follow the "laws of nature," that is, they must result from repeated application of the consequence function. The Chronicles function returns the valid ExternalChronicles for a particular environment.

$$
\begin{array}{l}
chronicles : Environment \longrightarrow \mathbb{P}\ ExternalChronicle \\
\hline
\forall\ env : Environment \bullet \\
\quad chronicles(env) \\
\quad = \{chronicle : ExternalChronicle \mid \\
\qquad (valuesOf(chronicle) \subseteq env.consistent) \land \\
\qquad (\forall\ t : restTimesOf(chronicle) \bullet \exists\ interaction : INTERACTION \bullet \\
\qquad\quad (env.consequence(chronicle(t-1), interaction) = chronicle(t)))\}
\end{array}
$$

Two Chronicles are indistinguishable if they cover the same time period and each of the external states in the Chronicle are indistinguishable. "sameChronicles" is a function that takes an environment as input and returns the set of pairs of indistinguishable Chronicles.

$$
\begin{array}{l}
sameChronicles : Environment \longrightarrow \mathbb{P}(ExternalChronicle \times ExternalChronicle) \\
\hline
\forall\ c_1, c_2 : ExternalChronicle; env : Environment \bullet \\
\quad (c_1, c_2) \in sameChronicles(env) \\
\quad \Leftrightarrow timesOf(c_1) = timesOf(c_2) \land \\
\qquad (\forall\ t : timesOf(c_1) \bullet \\
\qquad\quad \exists\ ind : env.indistinguishable \bullet c_1(t) \in ind \land c_2(t) \in ind)
\end{array}
$$

Task

A task is a description of what the agent is supposed to achieve in the environment. There are a wide variety of possible tasks. These include classic blocks world problems as well as tasks with time constraints and tasks of maintenance. The method of specifying tasks must be able to encode each of these types of tasks. In addition, the task specification should allow for different types of performance evaluation.

All task specifications require the environment, the set of possible initial conditions and a method for evaluating performance. The task schema provides the basic structure used to define all types of tasks. The schema requires that the initial ExternalDescription be consistent. The method of evaluation is given as a parameter, "E" to allow the schema to be specialized for different types of task evaluation functions. Throughout this specification, the "E" parameter will refer to the method of task evaluation.

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\textit{Task}[E]} \\
\textit{environment} : \textit{Environment} \\
\textit{initialConditions} : \textit{ExternalDescription} \\
\textit{evaluation} : E \\
\hline
\forall\, w : \textit{initialConditions} \bullet w \in \textit{environment}.\textit{consistent} \\
\hline
\end{array}
$$

The simplest form of task evaluation is binary. Either the task is accomplished or it is not. Such an evaluation can be encoded by listing all of the ways of accomplishing the task. An agent is successful if its behaviour, in response to conditions in the environment, is one of the methods of accomplishing the task. Note that we are not suggesting that any implementation would use such a representation. It is only meant to serve as a useful conceptualization.

A BinaryEvaluation is defined to be the set of desired ExternalChronicles that gives all the ways that the task can be accomplished. The BinaryEvaluation schema is used as a parameter to the task schema to define a BinaryTask. The BinaryTask schema imposes further restrictions on the desired Chronicles. The desired Chronicles must be valid Chronicles for the environment. In addition, each Chronicle must start in one of the given initial states. Finally, the task evaluation is not allowed to differentiate between Chronicles that differ only in hidden state.

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\textit{BinaryEvaluation}} \\
\textit{desired} : \mathbb{P}\ \textit{ExternalChronicle} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\textit{BinaryTask}} \\
\textit{Task}[\textit{BinaryEvaluation}] \\
\hline
\textit{evaluation}.\textit{desired} \subseteq \textit{chronicles}(\textit{environment}) \\
\forall\, c : \textit{evaluation}.\textit{desired} \bullet \textit{firstOf}(c) \in \textit{initialConditions} \land \\
\quad (\forall\, c' : \textit{ExternalChronicle} \bullet \\
\qquad (c, c') \in \textit{sameChronicles}(\textit{environment}) \Rightarrow c' \in \textit{evaluation}.\textit{desired}) \\
\hline
\end{array}
$$

The following examples illustrate the generality of the task representation. The examples include tasks of achievement, tasks of maintenance and tasks of information gathering.

Tasks of achievement require the agent to achieve some condition in the world. Classic STRIPS tasks are tasks of achievement where the conditions to be achieved are stated as a prepositional goals [Fikes and Nilsson, 1971]. Such a task is represented by the set of Chronicles where the goals are eventually achieved. The desired set of Chronicles for the blocks world task *On(A,B)* would include all Chronicles that started in one of the initial states and end with block A on block B. Note that this type of task requires an infinite number of desired Chronicles since no time limit is specified. Any Chronicle where the goal is eventually achieved is allowed. Tasks that involve temporal constraints restrict the number of desired Chronicles. The task On(A,B) by 5:00 pm today restricts the desired Chronicles to those that achieve the goal by the deadline.

A task of maintenance is a task where the agent must maintain some condition in the environment. For example, a thermostat agent may have to maintain the temperature of the room between 20 and 22 degrees Celsius. The task is represented by mapping the initial state to the set of Chronicles where the temperature does not vary outside of the allowed limits. It is also possible to add temporal constraints such as "Don't let the temperature vary above 22 degrees for more than 3 minutes at a time."

Tasks of observation require the agent to determine some information about the state of the world and act on it. Since tasks are behavioural descriptions, tasks must be specified in terms of behaviour; they cannot specify anything about the mental state of the agent. For example, the task to "Go outside and see if it is raining" is not operational. There is no way to determine if the agent "knows" whether it is raining. We cannot infer that the agent would know if it was raining even if we observed the agent going outside. Such a task can be made operational by adding actions that depend on the observations. The example could be changed to : "Go outside and open your umbrella if it is raining."

When defining a task, one must be careful when dealing with tasks that don't have a deadline or other constraints that prevent infinite loops. Consider again the blocks world. In most formulations of planning problems for this domain, there is no time limit on how long the agent may take to complete the task. As stated before, such a task results in an infinite set of desired Chronicles. An agent that does nothing will always be in a desired state, but will never achieve the goal. Even if the agent were to act, determining whether it will ever achieve the goal is an undecidable problem [Chapman, 1987]. An analogy can be made with the difference between partial and total correctness for a computer program. A program is partially correct if it never outputs a bad result and is totally correct if it never outputs a bad result, but always outputs a result and stops. Similarly, an agent is partially successful if it never does anything wrong, but is only totally successful if it also eventually completes the task. In this specification, success will mean total success.

Agent

An agent is any entity created to accomplish a task. We distinguish the agent from the environment to enable us to substitute one agent for another in order to compare their performance. Within the agent, we distinguish the mechanism from the controller, again to allow comparisons between different mechanisms and controllers.

Mechanism

An agent's Mechanism determines how the agent can perceive and affect its environment. It provides the agent's only means for interacting with the environment. The mechanism's sensors provide information about the state of the WORLD and the mechanism's effectors provide a means for changing the WORLD. The distinction between sensors and effectors has more to do with the flow of information and influence rather than physical arrangement. A force sensor on the end of an arm may be both the point of contact that affects the environment as well as the supplier of information about the magnitude of the force.

A mechanism is represented by a pair of functions that model its sensors and its effectors. The perceive function models sensors by mapping the current ExternalState to a PERCEPT. The effects function represents the effectors as a mapping from a COMMAND, issued by the agent's controller, and the agent's current CONFIGURATION to an INTERACTION with the environment. The perceive mapping allows for perceptual aliasing and sensor noise. Perceptual aliasing results when more than one distinguishable ExternalState maps onto the same PERCEPT. Sensor noise is the result of non-deterministic mapping of a single world onto multiple PERCEPTs. Noise is modeled by using the hidden world state to map indistinguishable ExternalStates to different PERCEPTs.

$$
\begin{array}{|l}
\hline
\textit{Mechanism} \\
\textit{perceive} : \textit{ExternalState} \rightarrow\!\!\!\!\!+ \textit{PERCEPT} \\
\textit{effects} : (\textit{COMMAND} \times \textit{CONFIGURATION}) \rightarrow\!\!\!\!\!+ \textit{INTERACTION} \\
\hline
\end{array}
$$

The following functions characterize the mechanism's interaction with the controller. The mechPercepts function gives the range of PERCEPTs that the mechanism's sensors can generate. Similarly, the mechCommands function gives the range of COMMANDs that the mechanism can accept. These functions will be used to simplify definitions involving the mechanism.

$$
\begin{array}{|l}
mechPercepts : Mechanism \longrightarrow \mathbb{P}\ PERCEPT \\
mechCommands : Mechanism \longrightarrow \mathbb{P}\ COMMAND \\
\hline
\forall\ Mechanism \bullet \\
\quad mechPercepts(\theta Mechanism) = \mathrm{ran}\ perceive \land \\
\quad mechCommands(\theta Mechanism) = \mathrm{dom}(\mathrm{dom}\ effects)
\end{array}
$$

Controller

In the abstract, the internals of an agent can be modeled as a finite state controller that consists of a control function and a memory. The control function is a mapping from a PERCEPT and current STATE to a COMMAND and next STATE. The exact nature of the control mapping and the information encoded in the state are determined by the agent's architecture and implementation.

The controller schema defines a controller to be a generic partial function where the type of the state is given as a parameter, "S". This allows controllers with different memory organizations to be specified as specializations of a generic controller. Throughout this specification, the "S" parameter will refer to the organization of the agent's memory. As a sanity constraint, the control function is limited to never produce a state that it cannot accept as input.

$$
\begin{array}{|l}
[S] \\
\hline
Controller : \mathbb{P}((PERCEPT \times S) \rightarrow\!\!\!\!\!+ (COMMAND \times S)) \\
\hline
\forall\ ctr : Controller \bullet (\mathrm{ran}(\mathrm{ran}\ ctr)) = (\mathrm{ran}(\mathrm{dom}\ ctr)) \\
\hline
\end{array}
$$

Below, a set of helper functions are defined to extract information about a controller. The ctrStates of a Controller are the set of possible controller STATEs. The ctrCommands of a controller are the set of commands the controller can issue. Finally, the ctrPercepts of a controller are the PERCEPTs that the controller can accept as input.

$$
\begin{array}{|l}
[S] \\
\hline
ctrStates : Controller[S] \longrightarrow \mathbb{P}\ S \\
ctrPercepts : Controller[S] \longrightarrow \mathbb{P}\ PERCEPT \\
ctrCommands : Controller[S] \longrightarrow \mathbb{P}\ COMMAND \\
\hline
\forall\ ctr : Controller[S] \bullet \\
\quad ctrStates(ctr) = \mathrm{ran}(\mathrm{dom}\ ctr) \land \\
\quad ctrPercepts(ctr) = \mathrm{dom}(\mathrm{dom}\ ctr) \land \\
\quad ctrCommands(ctr) = \mathrm{dom}(\mathrm{ran}\ ctr) \\
\hline
\end{array}
$$

A complete agent consists of a mechanism, a controller and an initial internal state. The controller and the mechanism must be matched in terms of PERCEPTs and COMMANDs; That is, the controller must be able to accept the PERCEPTs generated by the mechanism and the mechanism must be able to handle the COMMANDs issued by the controller. The agent schema also includes an initial state for the controller. This initial state encodes any explicit, a priori knowledge that the designer has given the agent. Different types of agents are defined as specializations of this general agent model.

$Agent[S]$

$controller : Controller[S]$
$mechanism : Mechanism$
$initialInternalState : S$

$ctrCommands(controller) \subseteq mechCommands(mechanism)$

$mechPercepts(mechanism) \subseteq ctrPercepts(controller)$

$initialInternalState \in ctrStates(controller)$

Agent System

Putting the parts together, we get an agent system consisting of the agent and the task that the agent is to perform. The agent is parameterized by the organization of its internal state and the task is parameterized by the type of evaluation function.

$AgentSystem[S, E]$

$Agent[S]$
$Task[E]$

Agent Chronicles

Now that the basic framework has been defined, we describe the machinery needed to characterize how the agent and the environment interact and evolve over time. The final result will be a function that takes an agent system and an initial state and returns the chronicle of external state. This function will form the basis for defining successful and in turn other agent properties.

The details of how the system changes with time are somewhat complex, but the basic idea is simple. Given a complete description of the AgentSystem and its state, the state at the next time interval can be determined. The future Chronicle of the system state is generated by repeatedly determining the next state. The part of the agent Chronicle that is of interest is the external state, since it is the observable behaviour of the agent. The ExternalChronicle can be obtained by projecting the external state out of the Chronicle of the system state.

The AgentSystemState schema is a complete, instantaneous description of the agent system. It consists of the AgentSystem augmented by the current internal and external states. In the schema, the internal and external states are restricted to be legal values for the agent and the environment.

$$
\begin{array}{l}
\underline{AgentSystemState[S, E]} \\
AgentSystem[S, E] \\
internalState : S \\
externalState : ExternalState \\
\hline
internalState \in ctrStates(controller) \\
externalState \in environment.consistent
\end{array}
$$

A subset of the AgentSystemStates are possible start states. An AgentSystem-Start is an AgentSystemState with the initial internal state and one of the possible initial external states.

$$
\begin{array}{l}
\underline{AgentSystemStart[S, E]} \\
AgentSystemState[S, E] \\
\hline
internalState = initialInternalState \\
externalState \in initialConditions
\end{array}
$$

Each transition of the system state must be a valid step which depends on the environments consequence function and the agent. The AgentSystemStep transition schema defines a valid step. The agent's control function is used to update the internal state and issue a command. The command is passed to the agent's mechanism and interacts with the environment to produce the next external state. The agent and the task remain unchanged.

$$
\begin{array}{|l}
\multicolumn{1}{l}{AgentSystemStep[S, E]} \\
\Delta AgentSystemState[S, E] \\
\hline
\exists\, cmd : COMMAND \bullet \\
\quad controller(mechanism.perceive(externalState), internalState) \\
\quad = (cmd, internalState') \land \\
\quad environment.consequence(externalState, \\
\qquad mechanism.effects(cmd, externalState.configuration)) \\
\quad = externalState' \\
\\
\theta AgentSystem = \theta AgentSystem'
\end{array}
$$

As time proceeds, the external state and the agent's internal state are updated after every time interval. This is not to say, however, that the controller must go through a decision cycle and generate a new COMMAND after each interval. Likewise, it does not mean that the environment only changes when the controller generates a new command. The controller may take any number of intervals to process the sensor output and calculate the next command to issue. In the mean time, the controller may continue to issue the same COMMAND (push the same buttons) or issue a null command (not push any buttons). What happens will depend on the implementation of the controller. The state of the controller is updated every interval to reflect any progress made in doing computation. Likewise, the state of the environment is updated to reflect any changes. The critical factor is the length of the time interval. It must be short enough that no significant information about the external state is lost.

An AgentChronicle is an infinite Chronicle of the evolution of the AgentSystemState. Each transition in the sequence is a valid step and the first state in the Chronicle must be a valid start state.

$$
\begin{array}{l}
AgentChronicle[S, E] == \{agentChronicle : Chronicle[AgentSystemState[S, E]] \mid \\
\quad \exists\, AgentSystemStart[S, E] \bullet \theta AgentSystemStart = firstOf(agentChronicle) \land \\
\quad (\forall\, t : restTimesOf(agentChronicle) \bullet \\
\qquad (\exists\, AgentSystemState[S, E]; AgentSystemState'[S, E] \bullet \\
\qquad\quad \theta AgentSystemState = agentChronicle(t - 1) \land \\
\qquad\quad \theta AgentSystemState' = agentChronicle(t) \land \\
\qquad\quad AgentSystemStep[S, E]))\}
\end{array}
$$

As noted above, if the system starts in a particular start state, then the AgentChronicle is uniquely determined. The AgentChronicleOf function maps an AgentSystem and an initial ExternalState to an AgentChronicle.

$$
\begin{array}{l}
[S, E] \\
\hline
AgentChronicleOf : (AgentSystem[S, E] \times ExternalState) \longrightarrow AgentChronicle[S, E] \\
\hline
\forall\ AgentSystem[S, E];\ ExternalState;\ agentChronicle : AgentChronicle[S, E] \bullet \\
\quad AgentChronicleOf(\theta AgentSystem, \theta ExternalState) = agentChronicle \Rightarrow \\
\quad\quad (\exists\ AgentSystemStart_1[S, E] \bullet \theta AgentSystemStart_1 = firstOf(agentChronicle) \wedge \\
\quad\quad\quad \theta AgentSystem = \theta AgentSystem_1 \wedge \\
\quad\quad (firstOf(agentChronicle)).externalState = \theta ExternalState)
\end{array}
$$

When examining the behaviour of the agent, we are only interested in changes in the external state. The chronicleProject function maps an AgentChronicle to a chronicle of the external state only. The chronicle project function is defined to be the function composition of the original chronicle and a lambda expression. Remember that a chronicle is itself a partial function from times to values. The lambda expression is a function that projects the ExternalState from the AgentSystemState. When the original chronicle is composed with the lambda expression, the resulting partial function is an ExternalChronicle that maps times to ExternalStates.

$$
\begin{array}{l}
[S, E] \\
\hline
chronicleProject : AgentChronicle[S, E] \longrightarrow ExternalChronicle \\
\hline
\forall\ a : AgentChronicle[S, E];\ ExternalState \bullet \\
\quad chronicleProject(a) = a \mathbin{;} (\lambda\ AgentSystemState[S, E] \bullet \theta ExternalState)
\end{array}
$$

Finally, we create the chronicleFrom function that gives the behaviour of the system when started in a particular initial state. This function is the culmination of the effort to define a framework for describing agents, tasks and environment. It forms the basis for defining successful.

The chronicleFrom function is the functional composition of the AgentChronicleOf and the chronicleProject functions. The AgentChronicleOf function produces the complete agent chronicle and the chronicleProject function projects out the chronicle of the external state.

$$chronicleFrom[S, E] == AgentChronicleOf[S, E] \mathbin{;} chronicleProject[S, E]$$

General Agent Properties

This section defines five general agent properties: successful, capable, perceptive, reactive and reflexive. These properties are relevant for any type of agent

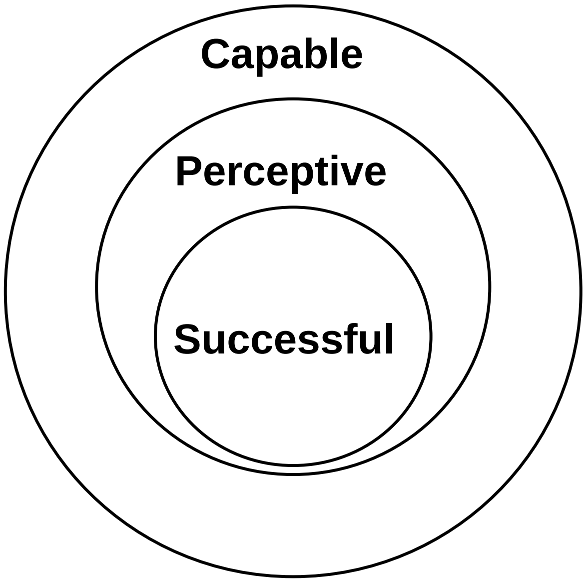

Figure 4: **General Agent Properties**

performing a task with a binary evaluation. The relationship between three of these properties is show in Figure 4. An agent is capable if its effectors are able to accomplish the task. A perceptive agent also possesses the sensors needed to determine how to operate the effectors for achieving the task. A successful agent is a perceptive agent with the right controller.

To assist in our definitions, we define the relation $=_{\text{Task}}$ that holds between two agent systems when the task, and hence, the environment are the same.

$$[S, E]$$

$$\_ =_{\text{Task}} \_ : AgentSystem[S, E] \leftrightarrow AgentSystem[S, E]$$

$$\forall\, AgentSystem_1[S, E];\ AgentSystem_2[S, E] \bullet$$
$$\quad \theta AgentSystem_1 =_{\text{Task}} \theta AgentSystem_2 \Leftrightarrow (\theta Task_1 = \theta Task_2)$$

Successful Agents

A successful agent always achieves its task. Given the concept of an AgentChronicle and the chronicleFrom function defined in the previous section, it is easy to define success for an agent performing a binary task. The chronicleFrom function is used to generate the infinite chronicle of the agent system when started in a particular state. The agent accomplishes the task from this state if some prefix of the infinite chronicle is one of the desired chronicles. The agent is successful if it accomplishes the task from all possible initial states.

$$
\begin{array}{|l}
\hline
\mathit{SuccessfulAgentSystem}[S] \\
\mathit{AgentSystem}[S, \mathit{BinaryEvaluation}] \\
\hline
\forall\, w : \mathit{initialConditions} \bullet \\
\quad \mathit{prefixesOf}(\mathit{chronicleFrom}(\theta \mathit{AgentSystem}, w)) \cap \mathit{evaluation.desired} \neq \emptyset \\
\hline
\end{array}
$$

Capable Agents

Capability reflects the ability of the agent's effectors to achieve a task. One way of demonstrating capability is to show that there is some agent with the same effectors that is successful, although it may not have the same perception or control function as the given agent. Capability ignores the fact that the given agent may not be able to perceive the relevant characteristics of the external state and may not choose the correct COMMANDs.

A CapableAgentSystem is defined as to be AgentSystem for which there exists a second AgentSystem with the same task, initial conditions and effectors (effects) that is successful. That is, the given agent would be successful if only it had the right controller and perception.

$$
\begin{array}{|l}
\hline
\mathit{CapableAgentSystem}[S] \\
\mathit{AgentSystem}[S, \mathit{BinaryEvaluation}] \\
\hline
\exists\, \mathit{SuccessfulAgentSystem}'[S] \bullet \\
\quad \theta \mathit{AgentSystem} =_{\text{Task}} \theta \mathit{SuccessfulAgentSystem}' \wedge \\
\quad \mathit{mechanism.effects} = \mathit{mechanism}'.\mathit{effects} \wedge \\
\quad \mathit{initialInternalState} = \mathit{initialInternalState}' \\
\hline
\end{array}
$$

Perceptive Agent

Perceptiveness measures the agent's ability to distinguish salient features of the environment. What constitutes a salient feature is determined by the task and the agent's mechanism. Different tasks require the agent to respond to different features and events in the world. For example, if the task were to open an umbrella when it is raining, then rain is a salient feature that must be responded to. The agent's mechanism is also important. Different mechanisms can achieve the same effect through different INTERACTIONS. Since an agent's PERCEPTs help determine its COMMANDs (and hence INTERACTIONs with the world), it is necessary to know the mechanism in order to determine which PERCEPTs are salient. For example, if the mechanism is a large tank and the task were to move from one location to another, then sensing bushes and rocks would not be required. If instead the mechanism were smaller and could get tangled in bushes, then sensing them would be salient.

Alternatively, perceptive could be defined to depend only on the task and not on the agent's effectors. An agent would be perceptive if its perceive function supplied the information needed for some effectors to achieve the task. But, what class of effectors should be considered? As illustrated by the tank example above, the class of possible effectors will determine the class of perceptive agents.

A more practical argument for making perceptive depend both on the task and the mechanism has to do with the way in which agents are generally constructed. Agent design, especially in robotics, usually proceeds from the design of the effectors to the design of a perception system. The perception system is selected to provide the information that is needed to control the effector to achieve a particular task or set of tasks. Defining perceptive in terms of the agent's mechanism focuses attention on the appropriateness of the perception system design for the selected effectors.

An agent is perceptive if its sensors provide the information needed to select COMMANDs that accomplish the task. Perceptiveness ignores the fact that the agent's controller may not actually select COMMANDs that achieve the task. The definition of perceptive will again depend on the existence of a successful agent, this time one with the same mechanism. A PerceptiveAgentSystem is an AgentSystem which has the same sensors and effectors as a SuccessfulAgentSystem.

$$
\begin{array}{l}
\underline{\quad PerceptiveAgentSystem[S] \quad} \\
CapableAgentSystem[S] \\
\hline
\exists\, SuccessfulAgentSystem'[S] \bullet \\
\qquad \theta AgentSystem =_{\text{Task}} \theta SuccessfulAgentSystem' \wedge \\
\qquad mechanism = mechanism' \wedge \\
\qquad initialInternalState = initialInternalState'
\end{array}
$$

Reactive Agent

There has been wide disagreement on what the term reactive means when applied to an agent. The American Heritage dictionary defines reactive to be *"Tending to be responsive or to react to a stimulus."*[dict, 1985] In AI, a common definition of reactive is *responding quickly and appropriately to changes in the environment*.

When considering binary task achievement, to achieve the task the agent must respond sufficiently quickly and appropriately to changes in the environment. An agent that is successful is then by definition sufficiently reactive. The definition of reactive will get more interesting when relative task achievement is considered.

$$ReactiveAgentSystem[S] == SuccessfulAgentSystem[S]$$

Reflexive

A term that is often confused with reactive is reflexive. An agent is reflexive if it responds only to immediate stimulus. Such agents are also called stimulus-response agents.

Reflexive agents don't need to maintain any memory. The history of the agent plays no part in determining its actions [Chrisman *et al.*, 1991][3].

To assist in the definition of reflexive, we define a function that determines the minimum number of internal states needed for a behaviourly equivalent agent.

$$
\begin{array}{l}
[S, E] \\
\hline
minCtrStates : (AgentSystem[S, E]) \longrightarrow \mathbb{N} \\
\hline
\forall\, AgentSystem[S, E] \bullet \\
\quad minCtrStates(\theta AgentSystem) \\
\quad = min(\{ n : \mathbb{N} \mid \\
\qquad (\exists\, AgentSystem'[S, E] \bullet \\
\qquad\quad \theta Task = \theta Task' \land \\
\qquad\quad mechanism = mechanism' \land \\
\qquad\quad (\forall\, w : initialConditions \bullet \\
\qquad\qquad chronicleFrom(\theta AgentSystem, w) \\
\qquad\qquad = chronicleFrom(\theta AgentSystem', w)) \land \\
\qquad\quad n = (\#(ctrStates(controller'))))\})
\end{array}
$$

A reflexive agent can be modeled as an agent system that never changes internal state. In this framework, that corresponds to an agent with a single internal state and hence no memory.

$$
\begin{array}{l}
ReflexiveAgentSystem[S, E] \\
\hline
AgentSystem[S, E] \\
\hline
minCtrStates(\theta AgentSystem) = 1
\end{array}
$$

[3] Argues that agents that require less memory are more reactive

Deliberative Agent

Environment

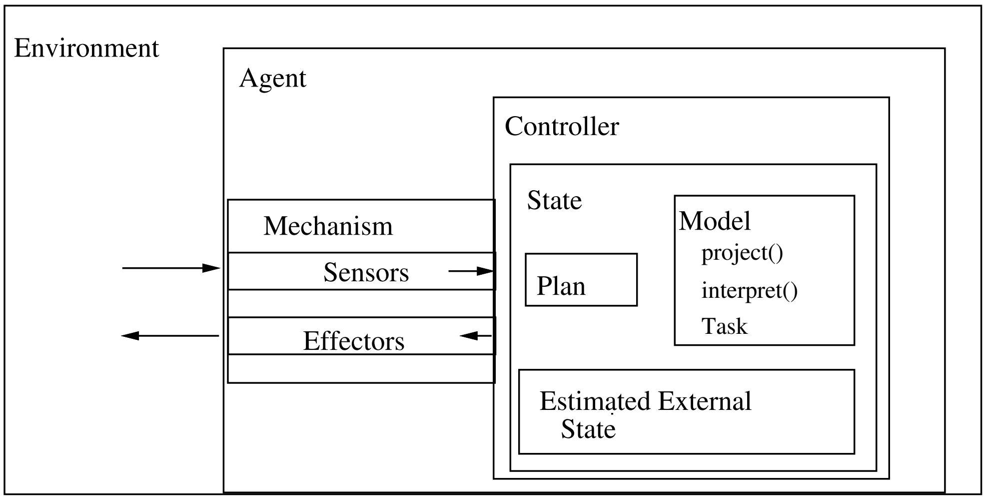

Figure 5: **Deliberative Agent System**

The agent properties defined in the previous section apply to any type of agent. In this section we introduce a type of agent of particular interest to the AI community known as a *deliberative agent*. A deliberative agent has an internal model of the world and uses its model to reason about the effects of COMMANDs in order to select COMMANDs that it predicts will accomplish the task. Figure 5 shows a conceptual model of the organization of a deliberative agent system. (The conceptual model of a generic agent was shown in figure 3)

An agent's internal model of the environment must provide certain basic functionality. In order to reason about the consequences of COMMANDs, the model must predict how COMMANDs will affect the external state. The model must also be able to derive information about the external state from sensor output. The sensor model is also needed to predict which PERCEPTs to expect in predicted future external states. In addition to the model, the 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. The result of an agent's deliberation process is a plan to accomplish the task. The agent needs to maintain a representation of the plan to be able to issue the chosen COMMANDs at the correct time. 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.

The internal state of a deliberative agent has six components; an *interpret* relation, a *project* relation, an *estimatedExternalState*, a *plan*, a *task* and some *workingMemory*. The *interpret* relation is a model of the agent's sensors and indicates which PERCEPTs the agent believes can be generated from a given external state. The *project* relation is a model of how the environment changes in response to a given COMMAND. The estimatedExternalState is the agent's current estimate of the external state. The plan is represented as a controller within the controller. It embodies the agent's intended response to PERCEPTs and the passage of time. The agent plans by extending and modifying this control function [McDermott, 1992].

$$ProjectRelation == ExternalState \times COMMAND \leftrightarrow ExternalState$$

$$InterpretRelation == ExternalState \leftrightarrow PERCEPT$$

$DelibState[E]$

$$\begin{array}{l} estimatedExternalState : ExternalDescription \\ plan : Controller[STATE] \\ interpret : InterpretRelation \\ project : ProjectRelation \\ task : Task[E] \\ workingMemory : STATE \end{array}$$

A DeliberativeAgentSystem is an AgentSystem where the agent has a deliberative internal state. Reasoning about the state of the world and planning are modeled as updates to the agent's internal state. The agent selects the next COMMAND for execution by interpreting its plan. For consistency, the plan in the internal state is restricted to only produce COMMANDs that the mechanism can accept and to accept all PERCEPTs that the mechanism can generate.

$DeliberativeAgentSystem[E]$

$$AgentSystem[DelibState[E], E]$$

$$\begin{array}{l} \forall\, internalState : ctrStates(controller) \bullet \\ \quad ctrCommands(internalState.plan) \subseteq mechCommands(mechanism) \land \\ \quad mechPercepts(mechanism) \subseteq ctrPercepts(internalState.plan) \end{array}$$

Since the agent's world model and task model are stored in its state (memory), the agent can modify them. Learning, in a deliberative agent, is accomplished by modifying its world and task models. However, since our purpose is to specify agent properties in terms of these models, we will restrict our definitions to agents that don't modify their models. This restriction greatly simplifies the definitions, but will have to be relaxed if learning agents are to be defined.

Simple deliberative agents are agents that don't change their world model or task model in the course of completing the task.

$$
\begin{array}{|l}
\hline
\textit{SimpleDeliberativeAgentSystem}[E] \\
\textit{DeliberativeAgentSystem}[E] \\
\hline
\exists\, \textit{project} : \textit{ProjectRelation};\ \textit{interpret} : \textit{InterpretRelation} \bullet \\
\quad (\forall\, \textit{internalState} : \textit{ctrStates}(\textit{controller}) \bullet \\
\qquad \textit{internalState.project} = \textit{project} \land \\
\qquad \textit{internalState.interpret} = \textit{interpret} \land \\
\qquad \textit{internalState.task} = \theta\,\textit{Task}) \\
\hline
\end{array}
$$

To make some of the definitions more readable, a SimpleAgentSystem is defined to be a DeliverativeAgentSystem with a binary task evaluation function.

$$
\begin{array}{|l}
\hline
\textit{SimpleAgentSystem} \\
\textit{SimpleDeliberativeAgentSystem}[\textit{BinaryEvaluation}] \\
\hline
\end{array}
$$

Deliberative Properties

A deliberative agent depends on its model of the world to enable it to accomplish its task. The properties defined for deliberative agents characterize the accuracy and suitability of the model for the task and how well the agent uses its model. Predictiveness characterizes the model's ability to make predictions about the environment. Likewise, interpretiveness characterizes the model's ability to infer information about the state of the world from the agent's sensors. Independent of the accuracy of the model, an agent is rational if it behaves in accordance with its model of the world. A rational agent with a correct model is sound.

Predictive Agent

An agent is predictive if its model of the world allows it to predict the results of its COMMANDs. A correct prediction relation must predict all the possible external

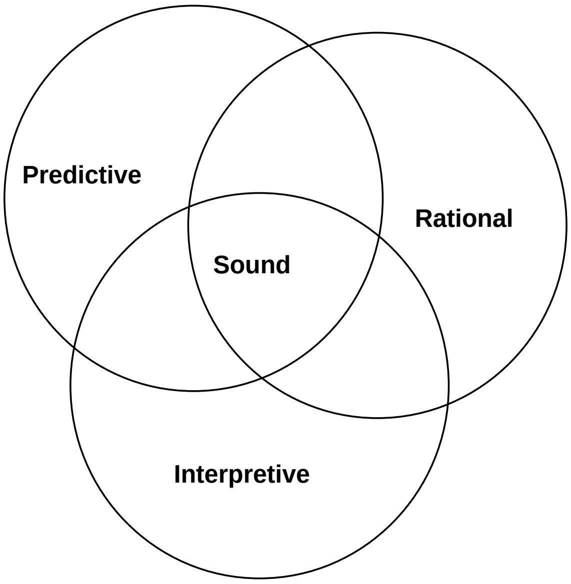

Figure 6: **Deliberative Agent Properties**

states that could result, and not predict impossible states. In this ideal case, the project function is the same as the mechanism's effects function composed with the environment's consequence function.

Given an environment and a mechanism, we can determine the correct projection relation. The definition of this relation is complicated by the fact that we don't want it to depend on hidden state. The projection relation is not required to be clairvoyant. The relation can be created by composing the effects and consequence functions to determine which externalState will result from a given initial externalState and command. The correct prediction relation predicts that any state, indistinguishable from the initial state, can produce any state that is indistinguishable from the final state.

$$
\begin{array}{|l}
correctProject : (Environment \times Mechanism) \rightarrow\!\!\!\!\!+ ProjectRelation \\
\hline
\forall\, env : Environment;\ mechanism : Mechanism \bullet \\
\quad correctProject(env, mechanism) \\
\quad = \{e_1, e_1', e_2, e_2' : env.consistent;\ cmd : COMMAND \mid \\
\qquad env.consequence(e_1, mechanism.effects(cmd, (e_1).configuration)) = e_1' \wedge \\
\qquad (e_1, e_2) \in sameExternalStates(env) \wedge \\
\qquad (e_1', e_2') \in sameExternalStates(env) \\
\qquad \bullet ((e_2, cmd), e_2')\}
\end{array}
$$

A predictive agent system is a SimpleDeliberativeAgentSystem that starts out with a correct project relation.

$$
\begin{array}{|l}
\hline
PredictiveAgentSystem[E] \\
SimpleDeliberativeAgentSystem[E] \\
\hline
initialInternalState.project = correctProject(environment, mechanism) \\
\hline
\end{array}
$$

It is also possible for the agent to have separate models of how the world works and how the agent's mechanism works. These individual models would correspond to the effects and consequence functions. Having separate models would allow the agent to reason about what INTERACTIONs are required to produce a desired result and then to reason about how to produce the INTERACTION. If I want to move a block east, I can decide to apply a force on the west side. I can then think about how I can control one of my hands to apply the correct force. For simplicity, the two models were be composed into a single project relation.

Interpretive Agent

An agent is interpretive if its model of its sensors allows it to correctly interpret the PERCEPTs it receives. The interpretation must be accurate. The relation should include all the possible (ExternalState,PERCEPT) pairs, but not those that are not possible.

As was done for the project function, we will define a function that generates the correct interpret relation. Again this definition is complicated by the fact that it should not depend on hidden state.

$$
\begin{array}{|l}
correctInterpret : (Environment \times Mechanism) \rightarrow\!\!\!\!\!+ \; InterpretRelation \\
\hline
\forall\, env : Environment;\ mechanism : Mechanism \bullet \\
\quad correctInterpret(env, mechanism) \\
\quad = \{e_1, e_2 : env.consistent;\ percept : PERCEPT \mid \\
\quad\quad (e_1, e_2) \in sameExternalStates(env) \\
\quad\quad \bullet\ (e_2, mechanism.perceive(e_1))\}
\end{array}
$$

$$
\begin{array}{|l}
\hline
InterpretiveAgentSystem[E] \\
SimpleDeliberativeAgentSystem[E] \\
\hline
initialInternalState.interpret = correctInterpret(environment, mechanism) \\
\hline
\end{array}
$$

Rational Agent

An agent is rational if it adopts plans that it predicts will succeed over plans predicted not to succeed. A deliberative agent can predict the future by repeatedly applying its project and interpret relations to its current estimate of the external state. The agent can use this ability to predict the future to simulate the execution of its current plan. If such a simulation leads to accomplishing the task, then the plan is predicted to succeed.

To define a rational agent, we need to be able to determine the result of having the agent simulate a plan. The PredictedChroniclesOf function takes a DeliberativeAgentSystem, an initial external state and a plan returning the set of predicted AgentChronicles. Note that the result is a set of Chronicles. There can be multiple possible predicted steps for any given state because the agent's project and interpret are relations and not functions.

The definition of predicted chronicles will parallel the definition of the AgentChronicles. First, a predicted step will be defined. A predicted Chronicle will consist of a sequence of predicted steps. The PredictedChroniclesOf an agent are then defined to be the set of predicted Chronicles that are consistent with the agent's world model.

A predicted step is a single predicted transition in the external and internal states.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad PredictedStep[E] \quad}} \\
\Delta AgentSystemState[DelibState[E], E] \\
\hline
\exists\, cmd : COMMAND;\ state : DelibState[E] \bullet \\
\qquad state = internalState \land \\
\qquad state.plan(state.interpret(externalState), state.workingMemory) \\
\qquad = (cmd, internalState'.workingMemory) \land \\
\qquad ((externalState, cmd), externalState') \in state.project \\
\\
\theta AgentSystem = \theta AgentSystem' \\
\hline
\end{array}
$$

A predicted future of the system is a sequence of AgentSystemStates where each transition between states is a predicted step. In addition, the first AgentState in the Chronicle must be a valid AgentStartState.

$$
\begin{array}{l}
PredictedFuture[E] \\
== \{agentChronicle : AgentChronicle[DelibState[E], E] \mid \\
\quad \exists\, AgentSystemStart[DelibState[E], E] \bullet \\
\quad \theta AgentSystemState = firstOf(agentChronicle) \land \\
\quad (\forall\, t : restTimesOf(agentChronicle) \bullet \\
\qquad (\exists\, AgentSystemState[DelibState[E], E];\ AgentSystemState'[DelibState[E], E] \bullet \\
\qquad\quad \theta AgentSystemState = agentChronicle(t-1) \land \\
\qquad\quad \theta AgentSystemState' = agentChronicle(t) \land \\
\qquad\quad PredictedStep[E]))\}
\end{array}
$$

Given a particular deliberative agent system and an initial world state, it is possible to generate the set of predicted agent Chronicles that are consistent with the initial conditions and the agent's world model. The PredictedChroniclesOf relation maps deliberative agent systems and initial world states to the set of AgentChronicles.

$$
\begin{array}{l}
[E] \\
\hline
PredictedChroniclesOf : (DeliberativeAgentSystem[E] \times Controller \times ExternalState) \\
\qquad \longrightarrow \mathbb{P}\ AgentChronicle[DelibState[E], E] \\
\hline
\forall\ DeliberativeAgentSystem[E]; plan : Controller; ExternalState \bullet \\
\quad PredictedChroniclesOf(\theta DeliberativeAgentSystem, plan, \theta ExternalState) \\
\quad = \{future : PredictedFuture[E] \mid \\
\qquad (\exists\ AgentSystemStart_1[DelibState[E], E] \bullet \\
\qquad\quad \theta AgentSystemStart_1 = firstOf(future) \land \\
\qquad\quad \theta AgentSystem = \theta AgentSystem_1 \land \\
\qquad\quad internalState_1.plan = plan \land \\
\qquad\quad (firstOf(future)).externalState = \theta ExternalState)\}
\end{array}
$$

An agent predicts that it succeeds when started in a particular initial world state if all the predicted chronicles from that initial state lead to success. The predicted success function maps deliberative agent systems to the set of worlds where the agent predicts it will succeed.

$$
\begin{array}{l}
predictedSuccess : SimpleAgentSystem \times Controller \longrightarrow ExternalDescription \\
\hline
\forall\ agentSystem : SimpleAgentSystem; plan : Controller \bullet \\
\quad predictedSuccess(agentSystem, plan) = \\
\qquad \{w : (agentSystem).initialConditions \mid \\
\qquad\quad (\forall\ ac : PredictedChroniclesOf(agentSystem, plan, w) \bullet \\
\qquad\qquad (\exists\ c : prefixesOf(chronicleProject(ac)) \bullet \\
\qquad\qquad\quad c \in agentSystem.initialInternalState.task.evaluation.desired))\}
\end{array}
$$

In the planning process, the agent revises its plan attempting to improve it. Occasionally, the agent will adopt a revised plan. An agent is rational if the revised plan it adopts is predicted to be at least as successful as the old plan. A plan is predicted to be at least as successful as another plan if it is predicted to succeed under a superset of conditions where the second plan is predicted to succeed. This definition of ration is very restrictive. A preferable definition would allow a rational agent to prefer plans where the predicted probability of success or the expected utility is higher. This requires a probability distribution over possible events, which will be defined later.

$$
\begin{array}{l}
\textit{RationalAgentSystem} \\
\hline
\textit{SimpleAgentSystem} \\
\hline
\forall\, \textit{update} : \textit{controller} \bullet \\
\quad (\exists\, \textit{planBefore}, \textit{planAfter} : \textit{Controller} \bullet \\
\qquad \textit{planBefore} = (\textit{second}(\textit{first}(\textit{update}))).\textit{plan} \land \\
\qquad \textit{planAfter} = (\textit{second}(\textit{second}(\textit{update}))).\textit{plan} \land \\
\qquad \textit{predictedSuccess}(\theta \textit{AgentSystem}, \textit{planBefore}) \\
\qquad \subseteq \textit{predictedSuccess}(\theta \textit{AgentSystem}, \textit{planAfter})) \\
\hline
\end{array}
$$

Sound Agent

A sound agent is predictive, interpretive and rational. That is, its project and interpret relations are valid and it selects plans that it predicts will succeed under more conditions.

$$
\begin{array}{l}
\textit{SoundAgentSystem} \\
\hline
\textit{PredictiveAgentSystem}[\textit{BinaryEvaluation}] \\
\textit{InterpretiveAgentSystem}[\textit{BinaryEvaluation}] \\
\textit{RationalAgentSystem} \\
\hline
\end{array}
$$

A sound agent is not necessarily successful. The agent's mechanism may not be capable or perceptive enough to accomplish the task. In such a case, there is no plan that can achieve the task. Even if a successful plan exists, the agent may not be able to generate it. Just because an agent could decide correctly whether a plan will succeed or not, does not mean it can generate a successful plan. The agent's plan generator may not be complete. It is also possible that it may take the agent too long to generate the successful plan. A complete plan generator will eventually generate a successful plan, if one exists, but by the time it has done so, some deadline for the task may have passed.

Utility of Task Achievement

When we talk about tasks and task accomplishment, we recognize that some ways of achieving a task are better than others. There are tradeoffs that must be made in terms of efficiency, resource use and time. One method of taking these tradeoffs into account is to define a utility function over possible task completions. Such a utility function maps a chronicle to a number that indicates the quality of the chronicle on a common scale.

This section refines the definition of a task to include a measure of utility. The utility function is used as the task evaluation function. It is included in the task definition since it is integral to what it means to accomplish a task. Using the utility augmented task definition, agent properties are redefined as partial orders on agent systems. It is then possible, for example, to say that one agent system is more capable than another, for a specific task in a specific environment.

A utility function is a mapping from a chronicle to a utility value.

$$Utility == ExternalChronicle \longrightarrow \mathbb{Z}$$

A UtilityAgentSystem is an agent where the task evaluation function is a utility function.

$$\begin{array}{l}
\textit{UtilityAgentSystem} \\
\hline
\textit{SimpleDeliberativeAgentSystem}[\textit{Utility}] \\
\hline
\end{array}$$

Relative Agent Properties

Given a utility task, the agent properties defined above can be refined to give relative measures of those properties. The definitions below give dominance relations for each property. These relations are only partial. They indicate when one agent system is clearly better than the other. They don't indicate any information in the case where one is sometimes better than the other and sometimes worse.

Relative Success

One agent is at least as successful as another if it always achieves the task with equal or higher utility from each possible initial state.

$$\begin{array}{l}
_ \geq_{\text{Successful}} _ : UtilityAgentSystem \leftrightarrow UtilityAgentSystem \\
\hline
\forall\, s1, s2 : UtilityAgentSystem \bullet \\
\quad s1 \geq_{\text{Successful}} s2 \Leftrightarrow s1 =_{\text{Task}} s2 \wedge \\
\qquad (\forall\, w : s1.initialConditions \bullet \\
\qquad\quad s1.evaluation(chronicleFrom(s1, w)) \\
\qquad\quad \geq s1.evaluation(chronicleFrom(s2, w)))
\end{array}$$

Relative Capability

An agent is at least as capable as another if its mechanism allows it to accomplish the task with equal or higher utility in all possible circumstances. This is made operational by saying that there exists an agent with the same effectors as that of the first agent that is more successful than all of the agents with the same effectors as the second agent.

$$
\begin{array}{l}
\_ \geq_{\text{Capable}} \_ : \mathit{UtilityAgentSystem} \leftrightarrow \mathit{UtilityAgentSystem} \\
\hline
\forall\, s1, s2 : \mathit{UtilityAgentSystem} \bullet \\
\quad s1 \geq_{\text{Capable}} s2 \Leftrightarrow s1 =_{\text{Task}} s2 \land \\
\quad\quad (\exists\, s1' : \mathit{UtilityAgentSystem} \bullet s1' =_{\text{Task}} s1 \land \\
\quad\quad\quad s1.\mathit{mechanism}.\mathit{effects} = s1'.\mathit{mechanism}.\mathit{effects} \land \\
\quad\quad\quad (\forall\, s2' : \mathit{UtilityAgentSystem} \bullet (s2' =_{\text{Task}} s2 \land \\
\quad\quad\quad\quad s2.\mathit{mechanism}.\mathit{effects} = s2'.\mathit{mechanism}.\mathit{effects}) \Rightarrow \\
\quad\quad\quad\quad\quad s1' \geq_{\text{Successful}} s2'))
\end{array}
$$

Relative Perceptiveness

An agent is at least as perceptive as another if its sensors provide information that allows it to control its effectors to achieve the task with equal or higher utility. Since the salient sensor information depends on the effectors, we will only compare agents with the same effectors.

Relative perceptiveness is made operational by saying that there exists an agent with the same mechanism as that of the first agent that is at least as successful as all agents with the same mechanism as the second agent.

$$
\begin{array}{l}
\_ \geq_{\text{Perceptive}} \_ : \mathit{UtilityAgentSystem} \leftrightarrow \mathit{UtilityAgentSystem} \\
\hline
\forall\, s1, s2 : \mathit{UtilityAgentSystem} \bullet \\
\quad s1 \geq_{\text{Perceptive}} s2 \Leftrightarrow \\
\quad\quad s1.\mathit{mechanism}.\mathit{effects} = s2.\mathit{mechanism}.\mathit{effects} \land \\
\quad\quad (\exists\, s1' : \mathit{UtilityAgentSystem} \bullet s1 =_{\text{Task}} s1' \land \\
\quad\quad\quad s1.\mathit{mechanism} = s1'.\mathit{mechanism} \land \\
\quad\quad\quad (\forall\, s2' : \mathit{UtilityAgentSystem} \bullet (s2 =_{\text{Task}} s2' \land \\
\quad\quad\quad\quad s2.\mathit{mechanism} = s2'.\mathit{mechanism}) \Rightarrow \\
\quad\quad\quad\quad\quad s1' \geq_{\text{Successful}} s2'))
\end{array}
$$

Relative Reactivity

Using our definition of reactive, an agent is more reactive than another if it acts appropriately and more quickly to accomplish the task. In the definition below, an agent is at least as reactive as another if it is at least as successful and accumulates utility at the same or faster rate.

$$
\begin{array}{l}
\_ \geq_{\text{Reactive1}} \_ : \mathit{UtilityAgentSystem} \leftrightarrow \mathit{UtilityAgentSystem} \\
\hline
\forall\, s1, s2 : \mathit{UtilityAgentSystem} \bullet \\
\quad s1 \geq_{\text{Reactive1}} s2 \Leftrightarrow ((s1 \geq_{\text{Successful}} s2) \land \\
\quad\quad (\forall\, w : s1.\mathit{initialConditions} \bullet \\
\quad\quad\quad (\forall\, c1 : \mathit{prefixesOf}(\mathit{chronicleFrom}(s1, w)); \\
\quad\quad\quad\quad c2 : \mathit{prefixesOf}(\mathit{chronicleFrom}(s2, w)) \bullet \\
\quad\quad\quad\quad \#c1 = \#c2 \Rightarrow \\
\quad\quad\quad\quad s1.\mathit{evaluation}(c1) \geq s1.\mathit{evaluation}(c2))))
\end{array}
$$

Relative Reflexivity

An agent is at least as reflexive as another if its behaviour matches that of a pure stimulus response system to the same extent. The idea that will be used to define relative reflexivity is that an agent that behaves in a way that requires less internal state must have its actions dictated by a control function that is closer to a pure stimulus-response system. Note that a more reflexive agent does not necessarily have less internal state, it just behaves in a way that requires less state.

A system at least as reflexive as another if it behaves in a way that require less memory. The function minCtrStates is used to determine the amount of memory required for the behaviour of each agent.

$$
\begin{array}{l}
\_ \geq_{\text{Reflexive}} \_ : \mathit{UtilityAgentSystem} \leftrightarrow \mathit{UtilityAgentSystem} \\
\hline
\forall\, s1, s2 : \mathit{UtilityAgentSystem} \bullet \\
\quad s1 \geq_{\text{Reflexive}} s2 \Leftrightarrow s1 =_{\text{Task}} s2 \land \\
\quad\quad (\mathit{minCtrStates}(s1) \leq \mathit{minCtrStates}(s2))
\end{array}
$$

Relative Predictiveness

An agent is at least as predictive as another if its prediction relation is more accurate. Mistakes in the prediction relation can be of two types. The prediction function can omit correct predictions and can include incorrect predictions. An agent is more predictive than another if its correct predictions are a superset of the correct predictions of the second agent and its omissions are a subset of the omissions of the second agent. Only agents with the same environment and effectors can be compared for predictiveness.

The definition of $\geq_{\text{Predictive}}$ makes use of the correctProject function. The correctProject function is used to generate the valid projection relation, pr. The correct projections made by each of the agent's project relations are their projection relations intersected with the correct projection relation. The correct projections of the first agent are required to be a superset of those of the second agent. The project relation of each agent minus the correct projections, pr, gives the set of incorrect predictions. The incorrect predictions of the first agent must be a subset of those made by the second agent.

$$
\begin{array}{l}
\_ \geq_{\text{Predictive}} \_ : \mathit{UtilityAgentSystem} \leftrightarrow \mathit{UtilityAgentSystem} \\
\hline
\forall s1, s2 : \mathit{UtilityAgentSystem} \bullet \\
\quad s1 \geq_{\text{Predictive}} s2 \Leftrightarrow (s1 =_{\text{Task}} s2 \wedge \\
\quad\quad s1.\mathit{mechanism}.\mathit{effects} = s2.\mathit{mechanism}.\mathit{effects} \wedge \\
\quad\quad (\exists\, pr : \mathit{ProjectRelation} \bullet \\
\quad\quad\quad pr = \mathit{correctProject}(s1.\mathit{environment}, s1.\mathit{mechanism}) \wedge \\
\quad\quad\quad (pr \cap s2.\mathit{initialInternalState}.\mathit{project}) \\
\quad\quad\quad\quad \subseteq (pr \cap s1.\mathit{initialInternalState}.\mathit{project}) \wedge \\
\quad\quad\quad (s1.\mathit{initialInternalState}.\mathit{project} \setminus pr) \\
\quad\quad\quad\quad \subseteq (s2.\mathit{initialInternalState}.\mathit{project} \setminus pr)))
\end{array}
$$

Relative Interpretiveness

An agent is at least as interpretive as another if its interpret relation makes the same or fewer mistakes. That is, if the incorrect interpretations included in its interpret relation are a subset of the incorrect interpretations of the other agent and its correct interpretations are a superset of the other agent's correct interpretations.

$$
\begin{array}{l}
\_ \geq_{\text{Interpretive}} \_ : \mathit{UtilityAgentSystem} \leftrightarrow \mathit{UtilityAgentSystem} \\
\hline
\forall s1, s2 : \mathit{UtilityAgentSystem} \bullet \\
\quad s1 \geq_{\text{Interpretive}} s2 \Leftrightarrow (s1 =_{\text{Task}} s2 \wedge \\
\quad\quad s1.\mathit{mechanism}.\mathit{perceive} = s2.\mathit{mechanism}.\mathit{perceive} \wedge \\
\quad\quad (s2.\mathit{initialInternalState}.\mathit{interpret} \cap s1.\mathit{mechanism}.\mathit{perceive}) \subseteq \\
\quad\quad\quad (s1.\mathit{initialInternalState}.\mathit{interpret} \cap s1.\mathit{mechanism}.\mathit{perceive}) \wedge \\
\quad\quad (s1.\mathit{initialInternalState}.\mathit{interpret} \setminus s1.\mathit{mechanism}.\mathit{perceive}) \subseteq \\
\quad\quad\quad (s2.\mathit{initialInternalState}.\mathit{interpret} \setminus s1.\mathit{mechanism}.\mathit{perceive}))
\end{array}
$$

Utility based Rationality

With a utility based measure of success, we can refine the meaning of rational. An agent is rational if it adopts plans with a higher utility over plans with a lower utility. The problem with this definition is that plans typically have a range of predicted utility. Each plan has a set of predicted outcomes and each outcome has a different utility. When we define a probability distribution over predicted outcomes, we can use expected utility for each plan to define rational. For now, a utility based rational agent is one that prefers plans that have at lease as high a minimum predicted utility in all possible starting conditions.

To assist with the definition of a UtilityRationalAgentSystem, we define a function that takes an agent, a plan and an initial external state and returns the minimum predicted utility.

$$
\begin{array}{l}
\mathit{predictedUtility} : \mathit{UtilityAgentSystem} \times \mathit{Controller} \times \mathit{ExternalState} \longrightarrow \mathbb{Z} \\
\hline
\forall\, \mathit{UtilityAgentSystem};\ \mathit{plan} : \mathit{Controller} \bullet \\
\quad (\forall\, w : \mathit{initialConditions} \bullet \\
\quad \mathit{predictedUtility}(\theta \mathit{AgentSystem}, \mathit{plan}, w) = \\
\quad\quad \mathit{min}(\{ \mathit{ac} : \mathit{PredictedChroniclesOf}(\theta \mathit{AgentSystem}, \mathit{plan}, w) \bullet \\
\quad\quad\quad \mathit{initialInternalState}.\mathit{task}.\mathit{evaluation}(\mathit{chronicleProject}(\mathit{ac}))\}))
\end{array}
$$

A UtilityRationalAgentSystem will only update its plan if the new plan has at least the same minimum predicted utility in each possible starting condition.

$$
\begin{array}{l}
\textbf{schema } \mathit{UtilityRationalAgentSystem} \\
\mathit{UtilityAgentSystem} \\
\hline
\forall\, \mathit{update} : \mathit{controller} \bullet \\
\quad (\exists\, \mathit{planBefore}, \mathit{planAfter} : \mathit{Controller} \bullet \\
\quad\quad \mathit{planBefore} = (\mathit{second}(\mathit{first}(\mathit{update}))).\mathit{plan} \wedge \\
\quad\quad \mathit{planAfter} = (\mathit{second}(\mathit{second}(\mathit{update}))).\mathit{plan} \wedge \\
\quad\quad (\forall\, w : \mathit{initialConditions} \bullet \\
\quad\quad\quad \mathit{predictedUtility}(\theta \mathit{AgentSystem}, \mathit{planBefore}, w) \\
\quad\quad\quad \leq \mathit{predictedUtility}(\theta \mathit{AgentSystem}, \mathit{planAfter}, w)))
\end{array}
$$

Relative Soundness

An agent is at least as sound as another if it is at least as predictive and interpretive and it is rational. That is, its project and interpret relations are as good or better and it selects plans that are predicted to have a higher minimum utility.

$$
\begin{array}{l}
\_ \geq_{\text{Sound}} \_ : \mathit{UtilityRationalAgentSystem} \leftrightarrow \mathit{UtilityRationalAgentSystem} \\
\hline
\forall\, s1, s2 : \mathit{UtilityAgentSystem} \bullet \\
\quad s1 \geq_{\text{Sound}} s2 \Leftrightarrow (s1 \geq_{\text{Predictive}} s2 \wedge \\
\qquad s1 \geq_{\text{Interpretive}} s2)
\end{array}
$$

Expected Task Performance

The relations defined in the previous section for agent comparisons are generally inadequate for common use. It is not the case that one agent will always dominate another in terms of its performance. For most agents, there are tradeoffs that are made that allow the agent to perform better in some cases, but not it others. When evaluating performance, the distribution of situations must be taken into account to give the expected performance of the agent.

To evaluate expected performance, a probability distribution over possible initial states is needed. Using a probability distribution, it is possible to determine the expected success. This is done by determining the performance in each possible initial state and weighting this performance by the probability of the initial state. Agent properties that are defined in terms of success can then be refined to take the relative distribution of states into account.

Probability Schemas

For the purposes of this specification, a discrete probability distribution will be represented as a "bag" of the appropriate items. The count of each item in the bag will be proportional to the frequency of that event in the probability distribution. The probability of a particular item is the count of the item divided by the sum of the counts of all the items in the bag.

$$
\begin{array}{l}
[X] \\
\hline
\mathit{Distribution} : \mathbb{P}(\text{bag } X)
\end{array}
$$

The purpose of using a distribution is to weight possible outcomes by their probability. The generic function probability takes an item and a distribution and returns an ordered pair (n,m) representing the probability of the item in the distribution as "n out of m". The generic weight function takes a probability distribution and a function that maps an item to a number. The weight function applies the function it is passed to each item in the Distribution and sums the results, weighting each by the probability of the item in the distribution.

$$
\begin{array}{l}
[X] \\
\hline
probability : (X \times Distribution[X]) \longrightarrow (\mathbb{Z} \times \mathbb{Z}) \\
weight : (X \longrightarrow \mathbb{Z}) \times Distribution[X] \longrightarrow \mathbb{Z} \\
\hline
\forall x : X;\ d : Distribution[X] \bullet probability(x, d) = (count(d)(x), \#d) \\
\forall f : X \longrightarrow \mathbb{Z};\ d : Distribution[X] \bullet \\
\quad weight(f, d) = \sum_{x \in d}(probability(x, d) * f(x))^4
\end{array}
$$

Expected Performance Evaluation

An expected performance task evaluation function is an extension of the utility task evaluation function. Expected performance evaluation depends on the utility function and the initial distribution. The expected performance is calculated by weighting the utility of the performance in each situation by the probability of that situation.

$$
\begin{array}{l}
ExpectedEvaluation \\
\hline
utility : Utility \\
initialDistribution : Distribution[ExternalState] \\
\hline
\end{array}
$$

An ExpectedAgentSystem is a SimpleDeliberativeAgentSystem with an expected evaluation function.

$$
\begin{array}{l}
ExpectedAgentSystem \\
\hline
SimpleDeliberativeAgentSystem[ExpectedEvaluation] \\
\hline
\end{array}
$$

Expected Success

The expected success of an agent is just the performance of an agent for each possible initial state weighted by the probability distribution of the initial state.

First we define a performance function that takes an ExpectedAgentSystem and returns a function that maps ExternalStates to performance.

$$
\begin{array}{l}
performance : ExpectedAgentSystem \longrightarrow (ExternalState \longrightarrow \mathbb{Z}) \\
\hline
\forall ExpectedAgentSystem;\ ExternalState \bullet \\
\quad performance(\theta ExpectedAgentSystem)(\theta ExternalState) = \\
\quad\quad evaluation.utility(chronicleFrom(\theta ExpectedAgentSystem, \theta ExternalState))
\end{array}
$$

[4] Not strictly Z code

Then we define an expected success function that takes an ExpectedAgentSystem and returns the expected performance.

$$
\begin{array}{l}
expectedSuccess : ExpectedAgentSystem \longrightarrow \mathbb{Z} \\
\hline
\forall\, ExpectedAgentSystem \bullet \\
\quad expectedSuccess(\theta ExpectedAgentSystem) = \\
\quad\quad weight(performance(\theta ExpectedAgentSystem), evaluation.initialDistribution)
\end{array}
$$

Expected Capability

Expected capability reflects the ability of the agent's effectors to achieve the task. It is defined to be the best expected performance that could be achieved with the same set of effectors doing the same task. One mechanism is more capable than another, in an expected value sense, if it has a higher expected capability.

$$
\begin{array}{l}
expectedCapability : ExpectedAgentSystem \longrightarrow \mathbb{Z} \\
\hline
\forall\, ExpectedAgentSystem \bullet \\
\quad expectedCapability(\theta ExpectedAgentSystem) = \\
\quad\quad max(\{ExpectedAgentSystem' \mid \\
\quad\quad\quad (\theta ExpectedAgentSystem =_{\text{Task}} \theta ExpectedAgentSystem') \land \\
\quad\quad\quad\quad (mechanism.effects = mechanism'.effects) \\
\quad\quad\quad\quad \bullet\ expectedSuccess(\theta ExpectedAgentSystem')\})
\end{array}
$$

Expected Perceptiveness

Expected perceptiveness reflects both the capability of the agents effectors and the perceptiveness of the agents sensors. The expected perceptiveness of an agent is the maximum expected success of any agent with the same mechanism. With this definition of perceptiveness, the expected perceptiveness will always be less than or equal to the expected capability.

$$
\begin{array}{l}
expectedPerceptiveness : ExpectedAgentSystem \longrightarrow \mathbb{Z} \\
\hline
\forall\, ExpectedAgentSystem \bullet \\
\quad expectedPerceptiveness(\theta ExpectedAgentSystem) = \\
\quad\quad max(\{ExpectedAgentSystem' \mid \\
\quad\quad\quad (\theta ExpectedAgentSystem =_{\text{Task}} \theta ExpectedAgentSystem') \land \\
\quad\quad\quad (mechanism = mechanism') \\
\quad\quad\quad \bullet\ expectedSuccess(\theta ExpectedAgentSystem')\})
\end{array}
$$

General Task Properties

Up to this point, we have considered properties of agents in relation to the task and the environment. In this section, we consider tasks in relation to all possible agents. This allows us to define properties of tasks and to categorize tasks as shown in figure 7.

The categorization of tasks depends heavily on the set of agents that are considered possible. Clearly, the task of going to the moon and returning moon rocks is achievable only if we allow autonomous rockets as possible agents. Changing the set of possible agents, changes the set of tasks that are achievable. Similarly, the set of tasks in other categories also changes as the set of possible agents changes.

Detailed explanations of each category of task follow.

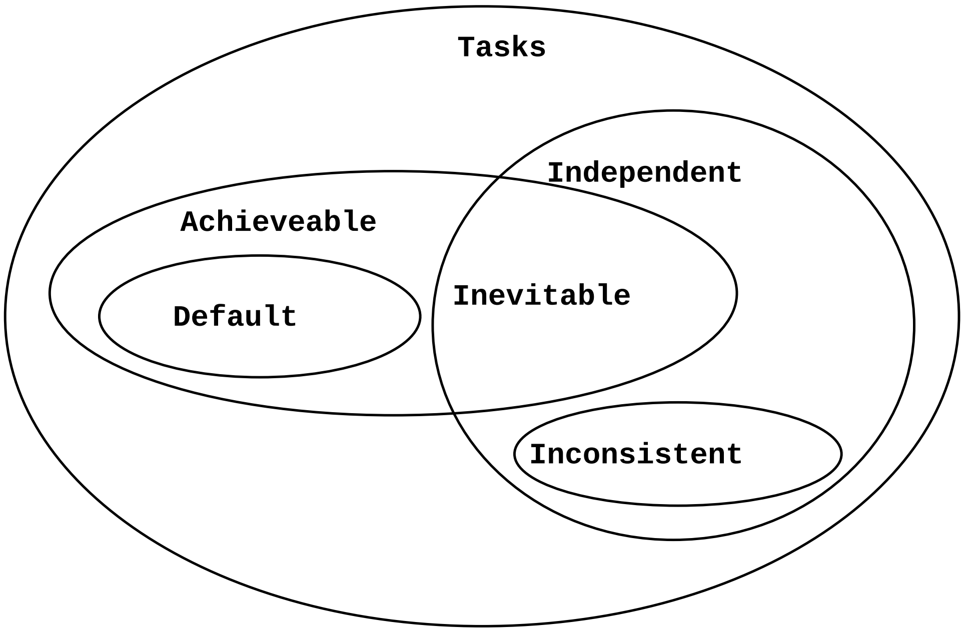

Figure 7: **Task Hierarchy Overview**

Consistent Tasks

It is possible that a task can be internally inconsistent. There may be some conditions under which there is no way of achieving the task. In this framework, an inconsistent task corresponds to a binary task where there is no desired chronicle that begins in one or more of the possible initial states. An inconsistent task cannot be accomplished by any agent. A classic example of such a task is requiring the agent to be in two places at the same time.

$$
\begin{array}{|l}
\hline
\textit{ConsistentTask} \\
\textit{BinaryTask} \\
\hline
\forall\, w : \textit{initialConditions} \bullet \\
\quad \exists\, \textit{chronicle} : \textit{evaluation.desired} \bullet \\
\quad\quad w = \textit{firstOf}(\textit{chronicle}) \\
\hline
\end{array}
$$

Inconsistent tasks are the complement of consistent tasks.

$$\textit{InconsistentTask} \mathrel{\widehat{=}} \textit{BinaryTask} \land \neg\, \textit{ConsistentTask}$$

Achievable Tasks

It is possible that a task is consistent yet still cannot be achieved by any one agent. Suppose the task were to recover the black box recorder from a crashed airplane. The possible agents are walking robots and autonomous submarines. If the airplane crashes on land, then the robot can recover the recorder. If it crashes in the ocean, then the submarine can recover it. The problem is that the submarine cannot travel on land and the robot cannot operate under water. The task is consistent since there is always some way of achieving it. However, it is not achievable by a single agent in the set of possible agents. If a walking submarine were added to the set of possible agents, then the task would be achievable.

The tasks that can be achieved by a single agent are achievable tasks.

$$
\begin{array}{|l}
\hline
\textit{AchievableTask}[S] \\
\textit{ConsistentTask} \\
\hline
\exists\, \textit{SuccessfulAgentSystem}'[S] \bullet \\
\quad \theta\, \textit{Task} = \theta\, \textit{Task}' \\
\hline
\end{array}
$$

Independent Tasks

There are some tasks where the agent has no influence over whether the task is achieved or not. The achievement of these tasks is independent of the actions of any possible agent. For example, consider the task of getting a particular star to go supernova at a particular time. The star either will go supernova or not, independent of what the agent does.

We define an independent task to be a task where if one agent succeeds in a given set of initial conditions, then all agents will succeed, given the same initial conditions.

$$
\begin{array}{l}
\textit{IndependentTask}[S] \\
\hline
\textit{BinaryTask} \\
\hline
\forall\, w : \textit{initialConditions} \bullet \\
\quad (\forall\, \textit{AgentSystem}'[S, \textit{BinaryEvaluation}] \bullet \\
\qquad (\theta \textit{Task} = \theta \textit{Task}') \Rightarrow \\
\qquad (\textit{prefixesOf}(\textit{chronicleFrom}(\theta \textit{AgentSystem}', w)) \cap \textit{evaluation.desired} = \varnothing)) \\
\quad \vee\ (\forall\, \textit{AgentSystem}'[S, \textit{BinaryEvaluation}] \bullet \\
\qquad \theta \textit{Task} = \theta \textit{Task}' \Rightarrow \\
\qquad (\textit{prefixesOf}(\textit{chronicleFrom}(\theta \textit{AgentSystem}', w)) \cap \textit{evaluation.desired} \neq \varnothing))
\end{array}
$$

Inevitably Achievable Tasks

A task will be achieved inevitably if, for all agents, all possible future states of the world are in the set of desirable future states. In this type of situation, the agent can do no wrong. Consider as an example, an agent that is given the task of getting the sun to rise tomorrow morning. No matter what the agent does, the sun will rise. (Although you may not be able to see it behind the clouds.)

An inevitable task is a task that is achievable and is independent of the particular agent.

$$\textit{InevitableTask}[S] \mathrel{\widehat{=}} \textit{IndependentTask}[S] \wedge \textit{AchievableTask}[S]$$

Default Task Achievement

Some tasks are possible to achieve by doing nothing and cannot be achieved if the agent does the wrong thing. These tasks are achieved by default. For example, if we give the agent the task of watering the lawn, and it is raining, there is nothing the agent has to do to complete the task. Note that this is not necessarily the same as an inevitable task. The agent could hold a large umbrella over the lawn and prevent it from getting wet. The agent can perform actions that will not accomplish the task.

The problem that must be addressed in this definition is "What does it mean for an agent to 'do nothing.' " Even when we are doing nothing, we are still breathing and growing hair.

To get around this problem, we designate one of the COMMANDs for each agent to be the "do nothing" or null command. When this COMMAND is sent to the mechanism, the mechanism does its own version of nothing.

$$NullCommand : COMMAND$$

A default task is a task that an agent can achieve by executing only NullCommands. To ensure that there are also ways to not achieve the task, default tasks are restricted not to be independent and thus not inevitable.

$$
\begin{array}{l}
\textbf{DefaultAchievableTask}[S] \\
\hline
AchievableTask[S] \\
\hline
\neg\ IndependentTask[S] \\[4pt]
\exists\ SuccessfulAgentSystem'[S] \bullet \\
\quad \theta Task = \theta Task' \land \\
\quad ctrCommands(controller') = \{NullCommand\}
\end{array}
$$

Future Work

The current specification does not deal with learning. There is no definition of learning. It is suggested that a deliberative agent could learn by becoming more predictive, interpretive and rational. While this is probably correct, the issues in learning have more to do with how well the learner generalizes. How performance of one task can improve performance of related tasks. To do this, some method is needed for talking about related tasks.

A second major direction for future work is to define the deliberative properties; predictive, interpretive and rational, in terms of the expected distribution of external states. It is more important to make correct predictions in situations that are more likely to occur. Such definitions would result in a finer scale for measuring these properties. This would be useful when defining learning agents. The problem in doing this is that the agent selects its actions based on its models. Changing the model could change the actions selected and thus change the expected distribution of external states. Any definition must take this shifting of external state distributions into account.

Another direction for future work would involve applying the framework developed in this specification to an actual agent system. This would involve tailoring the representations of the agent, the task and the environment for the particular system. Abstract definitions would have to be made operational. For instance, the evaluation function for a binary task could not be represented as an infinite set of infinitely detailed Chronicles. Extending this idea, multiple agents could be formalized and compared.

Conclusions

This paper has presented formal definitions for some properties of agents that perform tasks. For a general class of agents we have defined successful, capable, perceptive reactive and reflexive. In addition, predictive, interpretive, rational and sound were defined for deliberative agents. These definitions have been given in terms of a framework for discussing agents, tasks and environments. Through developing this framework, we have identified some important issues that any work in this area should address. These issues include how to represent time, non-determinism and interaction between the agent and the environment.

It is intended that the definitions and framework presented in this paper be useful for analyzing and comparing agents. It is also hoped that this work will help to foster discussion by providing a common and well defined vocabulary for talking about environments, tasks and agents.

Acknowledgments

I wish to thank David Garlan and Reid Simmons who were instrumental in guiding and encouraging the work reported in this paper. I would also like to thank Rich Caruana, Scott Reilly, Lonnie Chrisman, Siegfried Bocionek and David Zabowski for their comments.

A Z primer

This appendix contains a short introduction to the Z specification language. Its aim is to provide sufficient detail about the language to allow someone familiar with standard mathematical notation to be able to understand the specifications given in this report. For a more complete introduction to Z see [Spivey, 1989].

In this explanation, a simple specification for a library will be used as an example. The library will consist of a set of books and a card catalog. The example is very simple, but uses most of the Z notation needed to understand the agent properties specification.

Primitive Data Types

A primitive data type has no structure and no details are given about how the data type might be implemented. For the library, the basic data types needed are :

$[BOOK, CATALOGNUM, STRING]$

Primitive data types are written in all capitals by convention.

Abbreviations

It is often useful to abbreviate type definitions. Below a Bundle is defined to be an abbreviation for a set of books. The *P* before the BOOK is used to indicate the power set of all BOOKs.

$Bundle == \mathbb{P}\ BOOK$

Axiomatic Definitions

Functions and variables can be declared as axiomatic for the definition. The function *titleOf* defined below is a function that maps BOOKs to STRINGs. It is a function since each BOOK is mapped to a single STRING. In an axiomatic definition, the functions and variables are declared above the horizontal line. Below the line, any restrictions imposed on the definitions are given. In this case, the restriction says that the number of elements in the domain of the function is greater than or equal to the number of elements in the range of the function. In other words, the number of BOOKs is greater than or equal to the number of titles. There can be more than one book with the same title, but one book can't have two titles. In the Z notation, **ran** and **dom** are short for range and domain respectively. The "#"

sign indicates a count of the number of items in a set. Functions are indicated with "$\longrightarrow$" and partial functions are indicated with "$\rightarrow\!\!\!\!\!\!+$". Relations are indicated with "$\longleftrightarrow$".

$$
\begin{array}{|l}
titleOf : BOOK \longrightarrow STRING \\
\hline
\#(\mathrm{dom}\ titleOf) \geq \#(\mathrm{ran}\ titleOf)
\end{array}
$$

The variable "childrensBooks" is declared as a globally defined bundle of books.

$$
\begin{array}{|l}
childrensBooks : Bundle
\end{array}
$$

Schemas

Schemas are used to specify composite data types similar to records. The schema name appears at the top of the schema box. As with axiomatic definitions, the type declarations are separated from predicates that restrict them by a horizontal line.

The library schema is composed of a collection, a catalog and a numbering of BOOKs. The collection is a Bundle, or set of books. The catalog is a partial function that maps titles (STRINGs) to CATALOGNUMs. The numberOf partial function assigns CATALOGNUMs to particular books.

Below the horizontal line a number of restrictions on the elements of the library schema are specified. All BOOKs in the collection are included in the range of the partial function numberOf. Also, the title of every BOOK in the collection is in the catalog. Furthermore, for every CATALOGNUM in the catalog, there is a BOOK in the collection. The "$\bullet$" between the quantified variables and the following expression is just a separator.

$$
\begin{array}{|l}
\hline
\textit{Library} \\
collection : Bundle \\
catalog : STRING \rightarrow\!\!\!\!\!\!+\ CATALOGNUM \\
numberOf : BOOK \rightarrow\!\!\!\!\!\!+\ CATALOGNUM \\
\hline
\forall\, b : collection \bullet \\
\qquad (b \in (\mathrm{dom}\ numberOf) \wedge \\
\qquad titleOf(b) \in (\mathrm{dom}\ catalog)) \\
\\
\forall\, c : (\mathrm{ran}\ catalog) \bullet \\
\qquad (\exists\, b : collection \bullet \\
\qquad\qquad numberOf(b) = c) \\
\hline
\end{array}
$$

Schemas can also be refined to produce specializations. Below a children's library is defined to be a library where all the books in the collection are children's books. All components of the library schema and the restrictions placed on them are also part of the childrensLibrary schema.

$$\begin{array}{|l}
\hline
\textit{ChildrensLibrary} \\
\textit{Library} \\
\hline
\textit{collection} \subseteq \textit{childrensBooks} \\
\hline
\end{array}$$

When schemas are used in quantified expressions, the θ operator can be used to refer to the entire schema. The definition of collectionSize below maps a library to a number books in its collection. Within the scope of the $\forall$, the entire library schema can be referred to using $\theta Library$. It is convenient to read this as "$theLibrary$". θ can be used where ever the Library being referred to is obvious from the context. Individual components, like collection can be referenced directly.

$$\begin{array}{|l}
\textit{collectionSize} : \textit{Library} \longrightarrow \mathbb{Z} \\
\hline
\forall\, \textit{Library} \bullet \textit{collectionSize}(\theta \textit{Library}) = \#\textit{collection}
\end{array}$$

References

[Chapman, 1987] D. Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32, 1987.

[Chrisman *et al.*, 1991] Lonnie Chrisman, Rich Caruana, and Wayne Carriker. Intelligent agent design issues: Internal agent state and incomplete perception. In *Symposium on Sensory Aspects of Robotic Intelligence, AAAI Fall Symposium Series*. AAAI, Nov 1991.

[dict, 1985] *The AMERICAN HERITAGE DICTIONARY of the English language*. Houghton-Mifflin, second college edition, 1985.

[Fikes and Nilsson, 1971] Richard E. Fikes and Nils Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 5(2), 1971.

[McDermott, 1982] D. McDermott. A temporal logic for reasonig about processes and plans. *Cognitive Science*, 6:101–155, 1982.

[McDermott, 1992] Drew McDermott. Transformational planning of reactive behavior. Technical Report YALEU/CSD/RR 941, Yale University, December 1992.

[Mitchell, 1990] Tom M. Mitchell. Becoming increasingly reactive. In *Proceedings, Eight National Conference on Artificial Intelligence*, volume 2, pages 1051–1058. AAAI, The MIT Press, July 1990.

[Raiffa, 1968] Howard Raiffa. *Decision Analysis: Introductory Lectures on Choices under Uncertainty*. Addison-Wesley, Reading Mass., 1968.

[Skinner, 1974] B. F. Skinner. *About Behaviorism*. Random House Inc, New York, 1974.

[Spivey, 1989] J. M. Spivey. *The Z Notation, A Reference Manual*. Prentice Hall International Series in Computer Science. Prentice Hall, 66 Wood Lane End, Hamel Hemstead, Hertfordshire HP2 4RG, Endland, 1989.