Designing and Modelling Situated Agents Systematically: Cyberoos'98

Mikhail Prokopenko, Ryszard Kowalczyk, Maria Lee, Wai-Yat Wong

Applied Artificial Intelligence Project
CSIRO Mathematical and Information Sciences
Locked Bag 17, North Ryde, NSW 1670, Australia

e-mail: {mikhail.prokopenko, ryszard.kowalczyk, maria.lee, wai-yat.wong}@cmis.csiro.au

Abstract. The paper describes a hierarchical logic-based framework for intelligent agent architectures and proposes an approach to formalising interactivity in dynamic multi-agent systems. We first attempt to formally define various types of agent architectures, and encapsulate them in an ontology of synthetic worlds and situated agents. Then we evaluate the role of systematic methodology for multi-agent modelling. The approach can specifically assist in mapping logic theories of actions to reactive agent architectures, where ramifications are embedded in situated behaviours. The described hierarchical framework has been mapped to the RoboCup Simulation League domain, resulting in an implementation of the Cyberoos'98 – a soccer team of heterogeneous software agents.

Acknowledgements. The authors are grateful to Dong-Mei Zhang, Ian Mathieson, Aditya Ghose, Pierre Marcenac, Lin Padgham and James Westendorp for their comments on various aspects of the RoboCup Simulation League and multi-agent modelling. Special thanks to Marc Butler for his contributions towards prototyping the Cyberoos'98, and John Allinson for valuable technical assistance.

Designing and Modelling Situated Agents Systematically: Cyberoos'98

Abstract. The paper describes a hierarchical logic-based framework for intelligent agent architectures and proposes an approach to formalising interactivity in dynamic multi-agent systems. We first attempt to formally define various types of agent architectures, and encapsulate them in an ontology of synthetic worlds and situated agents. Then we evaluate the role of systematic methodology for multi-agent modelling. The approach can specifically assist in mapping logic theories of actions to reactive agent architectures, where ramifications are embedded in situated behaviours. The described hierarchical framework has been mapped to the RoboCup Simulation League domain, resulting in an implementation of the Cyberoos'98 – a soccer team of heterogeneous software agents.

1 Introduction

World modelling as the basis of artificial intelligence has been subject to questioning and criticism in the post-modern paradigm of artificial intelligence. In particular, since a model is always an approximation to reality, some aspects of reality are always omitted [23]. Another difficulty in building a general purpose perception system is that the correct symbolic description of the world must be task-dependent. Behavioural and multi-agent approaches to artificial intelligence are often proposed as a way to avoid these problems. They feature situatedness of agents reacting to changes in environment instead of reliance on abstract representation and inferential reasoning [3, 4, 23].

The methodology evolved in the field of Multi-agent Systems usually considers autonomous agents reacting to changes in external environment and (ideally) exhibiting emergent behaviour. On the other hand, approaches developed in the framework of Reasoning about Action are mostly logic-based, rely on a centralised world model, and try to (explicitly) capture various aspects of rationality.

The idea that reactive behaviours can be proved to be correct with respect to a theory of actions (and in some cases can be derived from it) is relatively new. For instance, connection between theories of actions and reactive robot control architectures based on the paradigm of situated activity is explored in [1]. The approach described in [1] formalises further the concept of "an action leading to a goal" defined at the representation level in the *situated automata approach* [12] and follows the latter in relating declarative agent specifications and situated behaviours. Development of agent architectures with "a formal model in logic and a direct link between that model and its implementation" is discussed in [16]. Recently the related problem of "formally proving high-level effect descriptions of actions from low-level operational definitions" [21] was addressed in the context of robotic knowledge validation, where an operational action definition is given in terms understood by the executing robot, that is, through sensors, actuators, and control algorithms.

The approach presented in [17, 18] does not require from an agent architecture derived from a high-level theory of continuous actions to be a logic-based formalism. On the contrary, the resulting architecture may contain only reactive behaviours validated with respect to a higher-level representation. The approach aims not only at obtaining sound translation procedures but also (and more importantly) at analysing and identifying classes of action domains corresponding to types of agent architectures. This paper complements results of [17, 18] by formally defining various types of agent architectures. The proposed hierarchical framework enables

- systematic design of situated agents architectures;
- construction of algorithms for a top-down design of agent architectures;
- rigorous comparative analysis of different architectures and their ranges of applicability with respect to provably correct logics.

2 Intelligent-Agent Architecture

In this section we define various classes of intelligent agent architectures and analyse their formal properties. Some of the architectures are well-known – for example, tropistic and hysteretic agents are discussed in [6]. We first attempt to incorporate these results in a framework suitable for situated synthetic agents. Then we try to extend the architecture, while retaining the rigour and clarity of fundamental definitions in [6].

A tropistic agent is usually defined as a tuple

$$<\boldsymbol{W}, \boldsymbol{P}, \boldsymbol{E}, \mathit{sense}, \mathit{do}, \mathit{behaviour}>, \quad (\mathbf{1})$$

where $\boldsymbol{W}$ is a set of all external states, $\boldsymbol{P}$ is a set of disjoint subsets of $\boldsymbol{W}$, and $\boldsymbol{E}$ is a set of effectors [6]. An agent is able to distinguish between external states in $\boldsymbol{W}$ by a sensory function *sense*: $\boldsymbol{W} \rightarrow \boldsymbol{P}$, but is not supposed to distinguish between sets in the same subset $\boldsymbol{P}$ due to sensory limitations. Activity of the agent is characterised by *behaviour*: $\boldsymbol{P} \rightarrow \boldsymbol{E}$. An execution (effectory) function maps each effector and an external state into the next state and is described as *do*: $\boldsymbol{E}$ x $\boldsymbol{W} \rightarrow \boldsymbol{W}$.

The architecture (1) does not seem to be suitable for an autonomous situated agent operating in a synthetic world. The reason is that, generally, sets $\boldsymbol{W}$, $\boldsymbol{P}$ and the execution function *do* belong to an environment simulator, rather than to autonomous situated agents themselves. Therefore, there is a need to withdraw the "alien" elements from the tropistic agent architecture, and introduce a specific agent type responsible for external environment simulation.

2.1 Environment Simulator

We define an *Abstract Simulator* agent as a tuple $\boldsymbol{A}_{AS}$

$$<\boldsymbol{W}, \boldsymbol{G}, \boldsymbol{A}, \boldsymbol{E}, \boldsymbol{C}, \mathit{view}, \mathit{projection}, \mathit{send}, \mathit{receive}, \mathit{do}>, \quad (\mathbf{2})$$

where sets $\boldsymbol{W}$, $\boldsymbol{E}$ and function *do* are defined as before, $\boldsymbol{G}$ is a set of all possible partitions of $\boldsymbol{W}$, $\boldsymbol{A}$ is a set of situated agents, and $\boldsymbol{C}$ is an implementation-dependent communication channel type. Function *view* structures situated agent perceptions by selecting a partition of external states for each agent. In other words, it maps an agent into an external states partition and defines *view*: $\boldsymbol{A} \rightarrow \boldsymbol{G}$. Dependent on a current situation in the synthetic world, the *Abstract Simulator* determines which particular element from a viewable partition is currently observable by every situated agent in $\boldsymbol{A}$. In other words, the *Abstract Simulator* projects an external state and a situated agent into an element of the viewable partition of external states, by using *projection*: $\boldsymbol{W}$ x $\boldsymbol{A} \rightarrow \boldsymbol{2}^{\boldsymbol{W}}$, where $\boldsymbol{2}^{\boldsymbol{W}}$ is the power-set of $\boldsymbol{W}$. The exact range of the *projection* function is the external states partition selected by *view* from the set $\boldsymbol{G}$ of all possible partitions of $\boldsymbol{W}$. More precisely,

$$\forall\, \boldsymbol{w} \in \boldsymbol{W}, \forall\, \boldsymbol{a} \in \boldsymbol{A}, \quad \mathit{projection}(\boldsymbol{w}, \boldsymbol{a}) \in \mathit{view}(\boldsymbol{a}) \quad (\mathbf{3})$$

The projected partition element is a set of external states ($\mathit{projection}(\boldsymbol{w}, \boldsymbol{a}) \subseteq \boldsymbol{W}$), and is sent by the *Abstract Simulator* to the situated agent. The domain of the *send* function is $\boldsymbol{A}$ x $\boldsymbol{2}^{\boldsymbol{W}}$, but its precise range $\boldsymbol{C}$ is implementation-dependent (e.g., is a string in the RoboCup Soccer Server). We will presume that situated agents are able to decode $\mathit{projection}(\boldsymbol{w}, \boldsymbol{a})$ from the input message, and respond back to the *Abstract Simulator* with an effector name. The received communication is decoded by *receive*: $\boldsymbol{A}$ x $\boldsymbol{C} \rightarrow \boldsymbol{E}$ (in general, the types of input and output messages may be different). The communicated effector is processed by the *do* function as *do*: $\boldsymbol{E}$ x $\boldsymbol{W} \rightarrow \boldsymbol{W}$.

This class is called abstract following the terminology of formal object-oriented methods [2]: it may not have any instances as all its functions are pure virtual. We aim to introduce an abstract class for each new agent type before implementing agent functions. This may facilitate applications of the proposed framework not only to the RoboCup Simulation League but also to other domains.

The *Simulator* agent is a sub-class of the *Abstract Simulator*, where the functions are implemented. Its architecture $\boldsymbol{A}_S$ is given as

$$\textbf{<}\boldsymbol{W}, \boldsymbol{G}, \boldsymbol{A}, \boldsymbol{E}, \boldsymbol{C}, \textbf{\textit{view}}, \textbf{\textit{projection}}, \textbf{\textit{send}}, \textbf{\textit{receive}}, \textbf{\textit{do}}\textbf{>}, \quad (\mathbf{4})$$

where the ***bold*** style indicates that a function is implemented.

2.2 Tropistic Agents

Having defined the architecture of the *Abstract Simulator* agent, we can formally describe an *Abstract Tropistic* agent as a tuple $\boldsymbol{A}_{AT}$

$$\textbf{<}\boldsymbol{C}, \boldsymbol{S}, \boldsymbol{E}, \textbf{\textit{sense}}, \textit{tropistic-behaviour}, \textbf{\textit{response}}\textbf{>}, \quad (\mathbf{5})$$

where $\boldsymbol{S}$ is a set of agent sensory states. The sensory function is re-defined as *sense*: $\boldsymbol{C} \rightarrow \boldsymbol{S}$, where an element of $\boldsymbol{C}$ is expected to carry the information on *projection*($\boldsymbol{w}$, $\boldsymbol{a}$). Activity of the agent is characterised by *tropistic-behaviour*: $\boldsymbol{S} \rightarrow \boldsymbol{E}$ as usual. We do not intend here to formally define the notion of reactive planning. However, by allowing the set $\boldsymbol{E}$ to include composite effectors $\boldsymbol{e}_1$; $\boldsymbol{e}_2$, where $\boldsymbol{e}_1 \in \boldsymbol{E}$, $\boldsymbol{e}_2 \in \boldsymbol{E}$, we can implicitly account for the case of tropistic planning - when a situated agent reacts to stimuli $\boldsymbol{S}$ with an n-length sequence of effectors.

The *response* function takes care of communicating the selected behaviour to the *Simulator* by encoding *response*: $\boldsymbol{E} \rightarrow \boldsymbol{C}$. This class is an abstract because the *tropistic-behaviour* function is not implemented. However, even a very simplistic client of the RoboCup Soccer Server must have a networking capability: to connect with the server, to initialise itself, etc. This is the reason for having *sense* and *response* functions implemented at this level.

It is interesting at this stage to consider a very simple sub-class of the *Abstract Tropistic* agent - a *Basic Situated* agent. This class has a specialised sensory function *timer*: $\boldsymbol{C} \rightarrow \boldsymbol{S}$, and does not specialise the function *sense* in any other way. In other words, a *Basic Situated* agent is able to distinguish only between external states which have different time values, having no other sensors apart from the *timer*. Moreover, its *tropistic-behaviour* function is not implemented as well, leaving this class among abstract classes. Formally, the *Basic Situated* agent class is defined as a tuple $\boldsymbol{A}_{BS}$

$$\textbf{<}\boldsymbol{C}, \boldsymbol{S}, \boldsymbol{E}, \textbf{\textit{sense}}, \textbf{\textit{timer}}, \textit{tropistic-behaviour}, \textbf{\textit{response}}\textbf{>}, \quad (\mathbf{6})$$

The next class in the hierarchy is a *Clockwork* agent. This sub-class of the *Basic Situated* agent specialises the *tropistic-behaviour* function by introducing the *command* function defined as *command*: $\boldsymbol{S} \rightarrow \boldsymbol{E}$. Since the only sensor available at this level is the *timer*, the agent behaviour is predefined and is totally driven by time values. In other words, like a clockwork mechanism, a *Clockwork* agent executes its fixed behaviour as a sequence of commands sent to the *Simulator* at predefined time points. Formally, the *Clockwork* agent class is defined as a tuple $\boldsymbol{A}_{CW}$

$$\textbf{<}\boldsymbol{C}, \boldsymbol{S}, \boldsymbol{E}, \textbf{\textit{sense}}, \textbf{\textit{timer}}, \textit{tropistic-behaviour}, \textbf{\textit{command}}, \textbf{\textit{response}}\textbf{>}, \quad (\mathbf{7})$$

The *Tropistic* agent class is derived from the *Clockwork* agent and finally allows us to implement the *tropistic-behaviour* function. It is given as tuple $\boldsymbol{A}_T$

$$\textbf{<}\boldsymbol{C}, \boldsymbol{S}, \boldsymbol{E}, \textbf{\textit{sense}}, \textbf{\textit{timer}}, \textbf{\textit{tropistic-behaviour}}, \textbf{\textit{command}}, \textbf{\textit{response}}\textbf{>}, \quad (\mathbf{8})$$

In practice, it is almost impossible to express each instantiation ($\boldsymbol{e}$, $\boldsymbol{s}$) of the *tropistic-behaviour* function $\boldsymbol{e}$ = *tropistic-behaviour*($\boldsymbol{s}$) in terms of complete sensory states. Instead, we represent such behaviour instantiations in terms of partial sensory states. For example, the following rules, given in the form similar to control rules [1] or condition-action pairs [12], describe behaviour instantiations:

if [SeeOpponent: (n, a, b, c, d, e) $\wedge$ NotZero(b)] **then** *turn*(b)

if [SeeOpponent: (n, a, b, c, d, e) ∧ NearZero(b)] **then** *dash*(2*a)
if [SeeBall: (a, b, c, d) ∧ Near(a)] **then** *kick*(100, b)

The bracketed component on the left-hand side correspond to elements of ***S*** and has to be evaluated as true[1] in order to activate an effector on the right-hand side. A sentence α in this component specifies the set of states from ***S*** consistent with α. In other words, it specifies a partial sensory state. Technically, each premise could be represented by a DNF, where each conjunct describes a complete sensory state. The DNF may be divided into a number of conjunctive premises, and each modified premise can be treated as a set of atomic formulae describing a complete sensory state and triggering the effector on the right-hand side.

2.3 Hysteretic Agents

The *Abstract Hysteretic* agent is defined here as a reactive agent maintaining internal state ***I*** and using it as well as sensory states ***S*** in activating effectors ***E***; i.e. its activity is characterised by *hysteretic-behaviour*: ***I*** x ***S*** → ***E***. Again, we allow the set ***E*** to include composite effectors $\boldsymbol{e}_1$; $\boldsymbol{e}_2$, where $\boldsymbol{e}_1 \in \boldsymbol{E}$, $\boldsymbol{e}_2 \in \boldsymbol{E}$, covering the case of hysteretic planning. A memory update function maps an internal state and an observation into the next internal state, i.e. it defines *update*: ***I*** x ***S*** → ***I***. An *Abstract Hysteretic* agent reacts to stimuli ***s*** sensed by *sense*(***c***) and activates effectors ***e*** according to *hysteretic-behaviour*(***i***, ***s***). The agent neither has full knowledge about the state *do*(***e***, ***w***) obtained by the *Simulator*, nor reasons about the transition. The next interaction with the world may bring partial knowledge about its new state.

It is well-known that inheritance is both restriction and extension [2], and in the proposed hierarchy the *Abstract Hysteretic* agent extends its superclasses by adding the *hysteretic-behaviour* function, while retaining all previously defined functions (i.e., it is a sub-class of the *Tropistic* agent). So an architecture $\boldsymbol{A}_{AH}$ of the *Abstract Hysteretic* agent is defined as a tuple

<***C***, ***S***, ***E, I***, ***sense***, ***timer***, ***tropistic-behaviour***, ***command***, *hysteretic-behaviour*, *update*, ***response***> (**9**)

It is worth noting that it is proposed for the *Abstract Hysteretic* agent to inherit implementation of the *tropistic-behaviour* function. Alternatively, both types of behaviour may be implemented simultaneously at the next level.

The following sub-class of the *Abstract Hysteretic* agent - *Hysteretic* agent - implements the *hysteretic-behaviour* and *update* functions, and is given as tuple $\boldsymbol{A}_H$

<***C***, ***S***, ***E, I***, ***sense***, ***timer***, ***tropistic-behaviour***, ***command***, ***hysteretic-behaviour***, ***update***, ***response***> (**10**)

Again, *hysteretic-behaviour* instantiations may be represented in terms of partial internal and sensory states. For example, the following rule describe a *hysteretic-behaviour* instantiation, where the effector on the right-hand side is composite:

if [SufficientStamina ∧ OpponentGoal: (e, f, g, h)] **and**
[SeeBall: (a, b, c, d) ∧ NearBall(self) ∧ AlmostOpposite(b, f)]
then *weak_kick*(90 - b); *strong_kick*(f - 5)

Two bracketed components on the left-hand side correspond to elements of ***I*** and ***S*** respectively. These sentences are abbreviations for longer sentences corresponding to complete internal and

[1] Assuming standard definitions of boolean connectives and the notion of being true for atomic formulae f: v.

sensory states, and have to be evaluated as true in order to activate the effector on the right-hand side.

An *Extended Hysteretic* agent $\boldsymbol{A_{EH}}$ is derived from the *Hysteretic* agent. Its architecture contains two additional communication components *notify* and *listen*, and is defined as a tuple

$$\textbf{<}\boldsymbol{C}, \boldsymbol{S}, \boldsymbol{E, I}, \boldsymbol{sense}, \boldsymbol{timer}, \boldsymbol{tropistic\text{-}behaviour}, \boldsymbol{command}, \boldsymbol{hysteretic\text{-}behaviour}, \boldsymbol{notify}, \boldsymbol{listen}, \boldsymbol{update}, \boldsymbol{response}\textbf{>} \quad (\mathbf{11})$$

where the communication functions are responsible for dealing with outcoming and incoming messages exchanged among situated agents (rather than between the *Simulator* and a situated agent). The *listen* function is specialised from the *sense* function, and *notify* function is a specialised *hysteretic-behaviour*. The reason for introducing these communication functions to the architecture is that ramification constraints may influence internal variables of other agents or require invocation of other agents' actions [18]. The distinction between *structural ramifications* when "the action can affect features of other objects than those which occur as arguments of the action" and *local ramifications* involving only "features of the argument objects" was identified in [20]. For example, the following domain constraint

$$\forall \text{ t}, \quad \mathbf{H}(\text{t, near(x): y}) \leftrightarrow \mathbf{H}(\text{t, near(y): x})$$

demands from a model to include the atomic formula near(B): A, whenever it contains the atomic formula near(A): B. Therefore, at the moment when agent A evaluates near(A): B as true (either by sensing a new observation, or by updating an internal variable), another agent (B in this case) needs to be notified. If the agent B has limited sensory capabilities (preventing, for example, a direct sensing of near(B): A), then the communication is the only way of ensuring a synchronous assignment.

It is worth noting that "listening" to a message is a form of sensing, and "speaking" is a form of action [16]. Therefore, the incoming messages can be sensed by a suitable specialised sensor, let us say, Told: ***e***, and the outcoming messages can be sent by specialised behaviour activating a suitable effector, let us say, Tell(g, ***e***), where ***e*** is an effector name, and g is a name of a receiving agent. For example,

if [LowStamina] **and** [SeePartner: (n, a, b, c, d, e) $\wedge$ SeeBall: (f, g, h, i) $\wedge$ NearBall(n)]
then Tell(NameOf(n), *turn*(e-f))

if [LookingForBall] **and** [Told: *turn*(x)] **then** *turn*(x)

The effector Tell(g, ***e***) has to be encoded in a *response*(Tell(g, ***e***)) as any other effector, and the sensory input Told: ***e*** must be decoded by sense(***c***) as any other sensory input.

2.4 Task-Oriented Agents

The behaviour functions in the described classes are uniformly defined across their respective domains and ranges. In other words, the sets of all behaviour instantiations $\boldsymbol{T} = \{(\boldsymbol{s}, \boldsymbol{e})$: $\boldsymbol{e}$ = *tropistic-behaviour*($\boldsymbol{s}$)$\}$ and $\boldsymbol{H} = \{(\boldsymbol{i}, \boldsymbol{s}, \boldsymbol{e})$: $\boldsymbol{e}$ = *hysteretic-behaviour*($\boldsymbol{i}$, $\boldsymbol{s}$)$\}$ are not partitioned or structured otherwise. This means that all agent's behaviour instantiations are always enabled. This may be acceptable in case of tropistic behaviour. Sometimes, however, it is desirable to disable all but a subset of behaviour instantiations – for example, when a tactical task requires concentration on a specific activity. The following agent class – *Abstract Task-Oriented* agent - is intended to capture this intention while retaining properties of the *Extended Hysteretic* agent (i.e., it is a sub-class of the latter). Instead of explicitly defining and structuring the sets $\boldsymbol{T}$ and $\boldsymbol{H}$, we define a *task* relation in the

domain of $\boldsymbol{I}$ x $\boldsymbol{S}$ x $\boldsymbol{2^I}$ x $\boldsymbol{2^S}$ x $\boldsymbol{2^E}$. Semantically, *task*($\boldsymbol{i}$, $\boldsymbol{s}$, $\boldsymbol{t_i}$, $\boldsymbol{t_s}$, $\boldsymbol{t_e}$) means that, given the internal state $\boldsymbol{i}$ and sensory state $\boldsymbol{s}$, an agent may activate only a particular part of behaviour function, where its domain and range are constrained by sets $\boldsymbol{t_i}$, $\boldsymbol{t_s}$ and $\boldsymbol{t_e}$. More precisely,

$$\forall\, \boldsymbol{i} \in \boldsymbol{I},\ \ \forall\, \boldsymbol{s} \in \boldsymbol{S},\ \ \forall\, \boldsymbol{e} \in \boldsymbol{E}, \qquad (\mathbf{12})$$
$$\boldsymbol{e} = \textit{hysteretic-behaviour}(\boldsymbol{i}, \boldsymbol{s}) \ \rightarrow\ (\textit{task}(\boldsymbol{i}, \boldsymbol{s}, \boldsymbol{t_i}, \boldsymbol{t_s}, \boldsymbol{t_e}),\ \ \boldsymbol{i} \in \boldsymbol{t_i},\ \boldsymbol{s} \in \boldsymbol{t_s},\ \boldsymbol{e} \in \boldsymbol{t_e})$$

Given its internal and sensory states, an *Abstract Task-Oriented* agent decides which particular task should be triggered. In other words, the agent architecture includes a decision relation, defined in the same domain as *task*. The following axiom

$$\forall\, \boldsymbol{i} \in \boldsymbol{I},\ \ \forall\, \boldsymbol{s} \in \boldsymbol{S},\ \ \forall\, \boldsymbol{t_i} \in \boldsymbol{2^I},\ \ \forall\, \boldsymbol{t_s} \in \boldsymbol{2^S},\ \ \forall\, \boldsymbol{t_e} \in \boldsymbol{2^E}, \qquad (\mathbf{13})$$
$$\textit{decision}(\boldsymbol{i}, \boldsymbol{s}, \boldsymbol{t_i}, \boldsymbol{t_s}, \boldsymbol{t_e}) \ \rightarrow\ \textit{task}(\boldsymbol{i}, \boldsymbol{s}, \boldsymbol{t_i}, \boldsymbol{t_s}, \boldsymbol{t_e})$$

ensures that a new task is activated if a new decision is taken. However, the axiom (13) by itself is not sufficient to solve the frame problem [15], which immediately appeared when a task constraint ($\boldsymbol{t_i}$, $\boldsymbol{t_s}$, $\boldsymbol{t_e}$) was associated with the argument ($\boldsymbol{i}$, $\boldsymbol{s}$) in the *task* relation. In other words, a task may not be carried forward without formally stating that in the absence of a new decision, a current task is preserved. The following default axiom accounts for a change in an internal state due to an *update*

$$\forall\, \boldsymbol{i} \in \boldsymbol{I},\ \ \forall\, \boldsymbol{s} \in \boldsymbol{S},\ \ \forall\, \boldsymbol{t_i}, \boldsymbol{t_i}^* \in \boldsymbol{2^I},\ \ \forall\, \boldsymbol{t_s}, \boldsymbol{t_s} \in \boldsymbol{2^S},\ \ \forall\, \boldsymbol{t_e}, \boldsymbol{t_e}^* \in \boldsymbol{2^E}, \qquad (\mathbf{14})$$
$$\neg(\textit{decision}(\textit{update}(\boldsymbol{i}, \boldsymbol{s}), \boldsymbol{s}, \boldsymbol{t_i}^*, \boldsymbol{t_s}^*, \boldsymbol{t_e}^*)) \ \rightarrow\ (\textit{task}(\boldsymbol{i}, \boldsymbol{s}, \boldsymbol{t_i}, \boldsymbol{t_s}, \boldsymbol{t_e}) \ \leftrightarrow\ \textit{task}(\textit{update}(\boldsymbol{i}, \boldsymbol{s}), \boldsymbol{s}, \boldsymbol{t_i}, \boldsymbol{t_s}, \boldsymbol{t_e}))$$

The second default axiom is concerned with a change due to a new sensory input:

$$\forall\, \boldsymbol{i} \in \boldsymbol{I},\ \ \forall\, \boldsymbol{s}, \boldsymbol{s}^* \in \boldsymbol{S},\ \ \forall\, \boldsymbol{t_i}, \boldsymbol{t_i}^* \in \boldsymbol{2^I},\ \ \forall\, \boldsymbol{t_s}, \boldsymbol{t_s} \in \boldsymbol{2^S},\ \ \forall\, \boldsymbol{t_e}, \boldsymbol{t_e}^* \in \boldsymbol{2^E}, \qquad (\mathbf{15})$$
$$\neg(\textit{decision}(\boldsymbol{i}, \boldsymbol{s}^*, \boldsymbol{t_i}^*, \boldsymbol{t_s}^*, \boldsymbol{t_e}^*)) \ \rightarrow\ (\textit{task}(\boldsymbol{i}, \boldsymbol{s}, \boldsymbol{t_i}, \boldsymbol{t_s}, \boldsymbol{t_e}) \ \leftrightarrow\ \textit{task}(\boldsymbol{i}, \boldsymbol{s}^*, \boldsymbol{t_i}, \boldsymbol{t_s}, \boldsymbol{t_e}))$$

The axiom (14) states that whenever the update results in an internal state, where a decision is not given, the current task is preserved. The axiom (15) ensures that a sensory state does not trigger a new task by itself, unless there is a decision involving a current internal state as well. Both axioms need to be complemented by an appropriate minimisation policy. Namely, the occurrence of *decision*($\boldsymbol{i}$, $\boldsymbol{s}$, $\boldsymbol{t_i}$, $\boldsymbol{t_s}$, $\boldsymbol{t_e}$) has to be minimised. In other words, when a decision is not specified, the agent assumes $\neg$*decision*($\boldsymbol{i}$, $\boldsymbol{s}$, $\boldsymbol{t_i}$, $\boldsymbol{t_s}$, $\boldsymbol{t_e}$) – the closed world assumption [19].

Finally, we can define the architecture of the *Abstract Task-Oriented* agent as the tuple $\boldsymbol{A_{ATO}}$

$$\langle \boldsymbol{C}, \boldsymbol{S}, \boldsymbol{E}, \boldsymbol{I},\ \boldsymbol{sense}, \boldsymbol{timer}, \boldsymbol{tropistic\text{-}behaviour}, \boldsymbol{command}, \boldsymbol{hysteretic\text{-}behaviour}, \boldsymbol{notify}, \qquad (\mathbf{16})$$
$$\boldsymbol{listen}, \boldsymbol{update}, \textit{task}, \textit{decision}, \boldsymbol{response}\rangle$$

The *Task-Oriented* agent is a sub-class of the *Abstract Task-Oriented* agent. It implements the *task* and *decision* functions, and is given as tuple $\boldsymbol{A_{TO}}$

$$\langle \boldsymbol{C}, \boldsymbol{S}, \boldsymbol{E}, \boldsymbol{I},\ \boldsymbol{sense}, \boldsymbol{timer}, \boldsymbol{tropistic\text{-}behaviour}, \boldsymbol{command}, \boldsymbol{hysteretic\text{-}behaviour}, \boldsymbol{notify}, \qquad (\mathbf{17})$$
$$\boldsymbol{listen}, \boldsymbol{update}, \boldsymbol{task}, \boldsymbol{decision}, \boldsymbol{response}\rangle$$

A *Task-Oriented* agent is capable of deciding which task is appropriate at a particular internal state given sensory inputs. However, it cannot share the decision-making process with other situated agents. Although, it is possible for the agent to notify other agents as to what effectors to activate, and to listen to such notifications, the decisions could not be communicated. The next logical step is to introduce this ability by appropriately specialising *sense* and *hysteretic-behaviour* functions – analogously to the extension made to the *Hysteretic* agent.

An architecture of an *Extended Task-Oriented* agent $\boldsymbol{A_{ETO}}$ (a sub-class of the *Task-Oriented* agent) contains two additional communication components *request* and *accept*, and is defined as a tuple

$$\langle \boldsymbol{C}, \boldsymbol{S}, \boldsymbol{E}, \boldsymbol{I},\ \boldsymbol{sense}, \boldsymbol{timer}, \boldsymbol{tropistic\text{-}behaviour}, \boldsymbol{command}, \boldsymbol{hysteretic\text{-}behaviour}, \boldsymbol{notify}, \qquad (\mathbf{18})$$
$$\boldsymbol{listen}, \boldsymbol{update}, \boldsymbol{task}, \boldsymbol{decision}, \boldsymbol{request}, \boldsymbol{accept}, \boldsymbol{response}\rangle$$

The *accept* function is specialised from the *sense* function, and *request* function is a specialised *hysteretic-behaviour*. The *Extended Task-Oriented* agent extends the *Task-Oriented* agent in the same way as the *Extended Hysteretic* agent extends the *Hysteretic* agent. However, new specialised functions deal with communicated decisions rather than behaviour instantiations. In other words, the *accept* function senses that an incoming message contains a *decision*, using a suitable sensor, Told: ***d***, and the *request* function arranges to send a *decision* to another situated agent by activating a suitable effector, Tell(g, ***d***), where ***d*** is a decision, and g is a name of a receiving agent. For example,

if [HasBall $\wedge$ CentrePath: blocked] **and** [SeePartner: (n, a, b, c, d, e) $\wedge$ Near(n) $\wedge$ Free(n)]
then Tell(NameOf(n), *decision*(*wall_pass*))

if [LookingForBall] **and** [Told: *decision*(*wall_pass*)] **then** *decision*(*wall_pass*)

The effector Tell(g, ***d***) has to be encoded in a *response*(Tell(g, ***d***)) as any other effector, and the sensory input Told: ***d*** must be decoded by sense(***c***) as any other sensory input. On accepting a *decision*, an agent activates the corresponding *task*.

The proposed hierarchical framework includes a number of implementation classes { $\boldsymbol{A}_S$, $\boldsymbol{A}_T$, $\boldsymbol{A}_H$, $\boldsymbol{A}_{EH}$, $\boldsymbol{A}_{TO}$, $\boldsymbol{A}_{ETO}$ }, allowing us to define teams of heterogeneous agents participating in a simulated soccer game.

2.5 Dynamic multi-agent systems

A dynamic multi-agent system is determined by a set of architecture types $\boldsymbol{A} = \{\boldsymbol{A}_1, \ldots \boldsymbol{A}_M\}$, where $\boldsymbol{A}_j \in \{ \boldsymbol{A}_S, \boldsymbol{A}_T, \boldsymbol{A}_H, \boldsymbol{A}_{EH}, \boldsymbol{A}_{TO}, \boldsymbol{A}_{ETO} \}$, $1 \le j \le M$, and a particular value of a discrete time parameter *t*. In other words, given a finite set of agents g_k ($1 \le k \le N$) instantiated from the architectures in $\{\boldsymbol{A}_1, \ldots \boldsymbol{A}_M\}$:

$$\forall\, g_k,\ 1 \le k \le N,\quad \exists\, \boldsymbol{A}_j,\ g_k \in \boldsymbol{A}_j,\ 1 \le j \le M,$$

one can construct a dynamic system $\boldsymbol{V}_A$, which maps an initial state and a time value to a state. More precisely, $\boldsymbol{V}_A$ is a function $\boldsymbol{U} \times \boldsymbol{R} \rightarrow \boldsymbol{U}$, where $\boldsymbol{U}$ is the set of possible states[2] $\boldsymbol{I}_1 \times \ldots \times \boldsymbol{I}_{N-1} \times \boldsymbol{W}$ and $\boldsymbol{R}$ is the set of real numbers. We will denote a state generated by the dynamic system $\boldsymbol{V}_A$ at the time instant *t* as $\boldsymbol{V}_A(t)$.

3 Synthetic Worlds Ontology

An ontology is defined as an explicit specification of conceptualisation [10], where the latter is a set of definitions allowing one to construct formal expressions about an application domain. We used the Ontolingua system [9] to provide a required level of detail in the RoboCup Simulation League domain and to capture the intelligent agent architectures described in the previous section. The Ontolingua is a mechanism for defining portable ontologies. It allows the user not only to define classes, relations, and distinguished objects using KIF [7] sentences, but also to translate these definitions into several implemented representation systems.

We specify a `State` as a `sentence` (defined in the KIF-Meta Ontology)

```
(Define-Class State (?X) "Any state."
 :Def (And (Sentence ?X)))
```

[2] Assuming without loss of generality that the agent g_N is a *Simulator* agent.

and External−State, Internal−State, Sensory−State as sub-classes of the State class. For example,

```
(Define-Okbc-Frame External-State
 :Type
 :Class
 :Direct-Types (Class Primitive)
 :Own-Slots
 ((Arity 1) (Documentation "Any External State."))
 :Direct-Superclasses (State))
```

The External−State class corresponds to W used in A_{AS} defined by (2), and any element $w \in W$ is an instance of the External−State class .

Now we can define External−States−Partition and Partition. The former is a sub-class of set (defined in the KIF-Sets Ontology) and corresponds to the set G of A_{AS}:

```
(Define-Frame External-States-Partition
 :Own-Slots
 ((Arity 1)
        (Documentation "Any Partition of External States.")
        (Instance-Of Class Primitive)
        (Subclass-Of Set))
 :Axioms
 ((<=>       (And    (External-States-Partition ?G)
                     (= ?W (Setofall ?S (External-State ?S))))
             (Set-Partition ?W ?G))))
```

An element $g \in G$ is an instance of the External−States−Partition class. The setofall operator and set−partition relation are defined in the KIF-Relations Ontology.

Having defined the External−States−Partition class, we introduce the Partition class. The Partition class is a sub-class of set as well, and any its instance is meant to be an element of some instance of the External−States−Partition class:

```
(Define-Frame Partition
 :Own-Slots
 ((Arity 1)
     (Documentation "An element of a partition.")
     (Instance-Of Class Primitive)
     (Subclass-Of Set))
 :Axioms
 ((<=> (Partition ?X0)      (Exists (?G)
                            (And    (External-States-Partition ?G)
                                    (Member ?X0 ?G))))))
```

The functions required by the Simulator are defined in a usual way. For example, the following function corresponds to the *projection* function:

```
(Define-Frame Situation-Projection
 :Own-Slots
 ((Arity 4)
     (Documentation "Projection function.")
     (Instance-Of Function))
 :Axioms
 (   (Nth-Domain Situation-Projection 1 Environment)
     (Nth-Domain Situation-Projection 2 External-State)
     (Nth-Domain Situation-Projection 3 Situated-Agent)
     (Nth-Domain Situation-Projection 4 Partition)
     (=> (And (View ?E ?A ?G)
              (Situation-Projection ?E ?S ?A ?P))
         (Member ?P ?G))))
```

where the last axiom captures the property (3). The Ontolingua system allows one to ensure that a class function is inherited by its sub-classes by specifying appropriate inheritance axioms. For

example, the following axiom, isomorphic to the inheritance axioms in the OKBC knowledge model [5], enables inheritance of quaternary functions by sub-classes and instances:

```
(Define-Frame Template-Quaternary-Function-Value
 :Own-Slots
  ((Arity 5)
      (Documentation "Inheritance of 4-nary functions.")
      (Instance-Of Relation))
 :Axioms
 ((Nth-Domain Template-Quaternary-Function-Value 1 Function)
  (Nth-Domain Template-Quaternary-Function-Value 2 Class)
  (Nth-Domain Template-Quaternary-Function-Value 3 Class)
  (Nth-Domain Template-Quaternary-Function-Value 4 Class)
  (Nth-Domain Template-Quaternary-Function-Value 5 Class)
  (=> (Template-Quaternary-Function-Value ?R ?Class ?Arg1 ?Arg2 ?Arg3)
      (=> (Subclass-Of ?C2 ?Class)
          (Template-Quaternary-Function-Value ?R ?C2 ?Arg1 ?Arg2 ?Arg3)))
  (=> (Template-Quaternary-Function-Value ?X0 ?X1 ?X2 ?X3 ?X4)
      (=> (Instance-Of ?I ?X1)
          (Holds ?X0 ?I ?X2 ?X3 ?X4)))))
```

Now it is sufficient to assert that a function is `Template-Quaternary-Function-Value` and specify the second argument as the class it belongs to. For example, the `Environment` class introduced as

```
(Define-Frame Environment
 :Own-Slots
 ((Arity 1)
      (Documentation "Environment Simulator.")
      (Instance-Of Class Primitive)
      (Subclass-Of Frame))
 :Axioms
 (    (Template-Quaternary-Function-Value   Do
       Environment External-State Effector External-State)
      (Template-Quaternary-Function-Value   Situation-Projection
       Environment External-State Situated-Agent Partition)
      (Template-Ternary-Function-Value      View
       Environment Situated-Agent External-States-Partition)))
```

asserts the `Situation-Projection` function as `Template-Quaternary-Function-Value`, so it may be properly inherited.

The situated agent classes of the hierarchy can be represented analogously. For example,

```
(Define-Frame Hysteretic-Agent
 :Own-Slots
 ((Arity 1)
      (Documentation "Hysteretic Agent.")
      (Instance-Of Class Primitive)
      (Subclass-Of Tropistic-Agent))
 :Axioms
 (    (Template-Quaternary-Function-Value  Hysteretic-Behaviour
       Hysteretic-Agent Internal-State Sensor-State Effector)))
```

The Gruber's definition of an ontology has been recently refined to interpret ontology as a partial specification [11]. Another clarification was proposed in [22]: "An ontology is an explicit, partial specification of a conceptualisation that is expressible as a meta-level viewpoint on a set of possible domain theories for the purpose of modular design, redesign and reuse of knowledge-intensive system components". This definition does not restrict ontologies to account for intended semantics only as in [11] and allows one to add new semantics through ontology mappings, thus enabling broader knowledge sharing and re-use. In general, ontologies defined in the Ontolingua system can be used within the framework of [22] to access other re-usable knowledge types.

4 Embedding Ramifications in Situated Behaviours

Most of traditional logic-based formalisations of reasoning about action describe the world as a set of time-dependent facts (fluents). In other words, truth value of each fluent is associated with a particular world state (situation). A reasoning system tries to *infer* what facts are true once the events have occurred (actions have been performed) and thus answer queries about the theory without actually updating it. The introduction of an additional situation (temporal) argument brings the well-known frame problem [15] of how to derive that a certain fact not affected by an action persist through its execution. Various reasoning systems designed to address the problem frequently employ a default rule (*law of inertia*) stating that truth value of a proposition persists through an action unless there is a specific information to the contrary. However, it is irrational to explicitly specify all indirect changes (ramifications) caused by the action [8].

Logic-based approaches to the ramification problem usually attempt to derive the indirect effects of an action as consequences of additional domain-specific information, such as domain constraints or causal laws [8, 14, 25]. In a multi-agent framework, a similar solution can be achieved by embedding indirect propagated effects in situated behaviours of autonomous reactive agents [18]. Therefore, to solve *the ramification problem* in multi-agent modelling, one needs to carefully examine the notion of independence and analyse what formalisms and intelligent agent architectures are best suited for the task of *situated reasoning*.

We follow the approach of [18] in selecting a certain class of domains and translating a domain description (given as a logic theory of actions) into a dynamic multi-agent system. The conversion preserves the meaning of the domain description as compared with the multi-agent system's dynamics. In other words, state transitions produced by behaviours of autonomous agents are warranted by logic-based reasoning about actions and change.

Initially, a basic action theory describing unconstrained domains is used to derive a dynamic multi-agent system based on the *Hysteretic* agent architecture. Then a more complex class of domains with logical and causal constraints is mapped into another dynamic system (based on the *Extended Hysteretic* agent architecture), using an augmented translation procedure. Both obtained translations are sound with respect to the underlying action theories.

4.1 From Logic-based Reasoning to Behaviour-based Dynamics

The Sandewall approach to representing operational definitions and effect descriptions of continuous actions [21] follows a *narrative time-line approach* and allow the definition of continuous change, discrete discontinuities, the distinction between true and estimated values of state variables, and the distinction between success and failure of an action. We will adopt from [21] the following notation:

H(t, f:v): fluent f has the value v at time t;
X(t,f): fluent f is exempt from minimisation of discontinuities (the occlusion operator);
G(s, *a*): the action *a* is invoked at time s;
A(s, *a*): the action *a* is applicable at time s;
$\mathbf{D}_s$([s,t], *a*): the action *a* is *successfully executed* over the time interval [s,t];
$\mathbf{D}_f$([s,t], *a*): the action *a* *fails* over the time interval [s,t];
$\mathbf{D}_c$([s,t], *a*): the action *a* is *being executed* during the time interval [s,t).

All state variables (fluents or features) in a described domain may have an argument. The reassignment operator := is used to abbreviate $\mathbf{H}(t,f:v) \wedge \mathbf{X}(t,f)$ as $\mathbf{H}(t, f:=v)$.

4.2 Basic Action Theory for Unconstrained Domains

We start with simple deterministic synthetic worlds where domain constraints are not defined, and therefore all action effects are direct. The action *success* description has the following form [21]:

$$\mathbf{D}_s([s,t], a) \rightarrow \mathbf{H}(t, \omega_a), \quad (\mathbf{19})$$

where ω_a is the post-condition of the action a given at the termination time. For example,

$$\mathbf{D}_s([s,t], PASS(x, y, p)) \rightarrow \mathbf{H}(t, has\_ball(y)) \quad (\mathbf{20})$$

describes successful execution of the *PASS* action, applicability description of which can be given as

$$\mathbf{A}(s, PASS(x, y, p)) \leftrightarrow \mathbf{H}(s, status(y): free) \wedge \mathbf{H}(s, adequate\_stamina(x, p)) \quad (\mathbf{21})$$

An action, once invoked, continues towards a success at which instant it terminates (unless there is a qualification that forces it to fail earlier):

$$\mathbf{A}(s, PASS(x, y, p)) \wedge \mathbf{G}(s, PASS(x, y, p)) \rightarrow \mathbf{H}(s, ball\_velocity(x) := p) \quad (\mathbf{22})$$

$$\mathbf{H}(t, see\_ball(x): z) \wedge \mathbf{H}(t, see\_partner(x): y) \wedge \mathbf{H}(t, very\_near(y): z) \wedge \mathbf{D}_c([s,t], PASS(x, y, p)) \rightarrow \mathbf{D}_s([s,t], PASS(x, y, p)) \quad (\mathbf{23})$$

Axioms exemplified by (22) and (23) are called *invocation* and *termination* descriptions respectively:

$$\mathbf{A}(s, a) \wedge \mathbf{G}(s, a) \rightarrow \mathbf{H}(s, \gamma_a) \quad (\mathbf{24})$$

$$\mu_i \wedge \mathbf{D}_c([s,t], a) \rightarrow \mathbf{D}_s([s,t], a) \quad (\mathbf{25})$$

where γ_a is the *invocation* condition and μ_i is one of the *termination* conditions.

An action failure is defined by a *failure* description (where in general the failure reasons δ_i can be implied by other information):

$$\delta_i \wedge \mathbf{D}_c([s,t], a) \wedge \neg\mathbf{D}_s([s,t], a) \rightarrow \mathbf{D}_f([s,t], a) \quad (\mathbf{26})$$

and by a *failure effects* description [17]:

$$\mathbf{D}_c([s,t], a) \wedge \mathbf{D}_f([s,t], a) \rightarrow \mathbf{H}(t, \tau_a), \quad (\mathbf{27})$$

where τ_a is the *failure post-condition*.

We will denote the described theory of actions as $\boldsymbol{T}_1 = \langle \boldsymbol{D}, \boldsymbol{M} \rangle$, where all domain axioms compose $\boldsymbol{D}$, and $\boldsymbol{M}$ is a specific minimisation policy. As mentioned in [21], chronological minimisation (or restriction) of discontinuities in piecewise continuous fluents should, in general, be integrated into a non-monotonic policy preferring "a model where an action continues over a model where it terminates, either by success or by failure".

It can be shown [18] that the described theory of actions $\boldsymbol{T}_1$ provides a validation criterion for the dynamic system $\boldsymbol{V}_A$, where each agent is uniformly defined by the architecture $\boldsymbol{A}_H$ (*Hysteretic* Agent). The following validation criterion is suggested: a translated atomic formula tr(f):v is valid in a state $\boldsymbol{V}_A(t)$ if $\boldsymbol{H}(t, f{:}v)$ is entailed by a consistent theory $\boldsymbol{T}_1$.

4.3 Extended Action Theory

The extended action theory allows us to reason about ramifications and interactions. Arguably, the most obvious ramification of a continuous action is an entailment of the so-called prevail condition during the action execution, and its cancellation upon the action failure termination [21]. On the

operational level, such changes must be (non-monotonically) derived as consequences of domain constraints. For example,

$$\forall\ t,\ \mathbf{H}(t, near(x): y) \wedge \mathbf{H}(t, near(x): z) \rightarrow \mathbf{H}(t, near(y) := z)$$

The last constraint uses the occlusion operator **X**(t,f) and excludes (releases) the indirect effects from the law of inertia. This effectively specifies the direction of the dependency and makes the latter look like a "causal rule" producing necessary ramifications [14, 25].

Another, more generic, form of ramifications describes an *interaction* between two continuous concurrent actions:

$$\lambda_i \wedge \mathbf{D}_c([s,t], a) \wedge \neg\mathbf{D}_f([s,t], a) \rightarrow \mathbf{G}(t, b), \qquad (\mathbf{28})$$

where each λ_i represents an interaction condition, and *b* is another action invoked by occurrences of λ_i during the execution of the action *a*. For example,

$$\mathbf{H}(t, see\_opponent(x): z) \wedge \mathbf{H}(t, near(x): z) \wedge \mathbf{H}(t, see\_partner(x): y) \wedge \qquad (\mathbf{29})$$
$$\mathbf{D}_c([s,t], DRIBBLE(x, d)) \wedge \neg\mathbf{D}_f([s,t], DRIBBLE(x, d)) \rightarrow \mathbf{G}(t, PASS(x, y, \textbf{\textit{distance}}(x, y)))^3$$

G(t, *b*) is the only specified effect of the interaction. Therefore other effects of the action *b* (defined in its success, failure, and/or interaction descriptions) can be viewed as ramifications of this interaction. They do not have to be specified explicitly with every such interaction and are supposed to be implied indirectly. The applicability description **A**(t, *b*) is responsible for the actual execution or immediate termination of the action *b*. For example, **A**(t, *PASS*(x, y)) specified in (21) checks other preconditions for the action invocation.

Thus at least two ways to address the ramification problem in a logic characterising piecewise continuous change can be observed: by defining constraints and by specifying *interaction* descriptions for continuous actions.

The described theory of actions $\boldsymbol{T}_2 = \langle \boldsymbol{D}, \boldsymbol{M} \rangle$ allows the axioms composing $\boldsymbol{D}$ to include domain constraints and interaction descriptions. It is worth noting that the policy of chronological minimisation $\boldsymbol{M}$ does not have to be modified. It can be shown [18] that the theory of actions $\boldsymbol{T}_2$ provides a validation criterion for the dynamic system $\boldsymbol{V}_A$, where each agent is uniformly defined by the architecture $\boldsymbol{A}_{EH}$ (*Extended Hysteretic* Agent).

The obtained results can be generalised by translating broader classes of action domains into more complex agent architectures. Ideally, any extended translation $\boldsymbol{Tr}_k$: $\boldsymbol{D} \Rightarrow \boldsymbol{V}_k$ must satisfy the important soundness property: state transitions produced by a dynamic multi-agent system $\boldsymbol{V}_k$ are valid with respect to reasoning warranted by an action theory $\boldsymbol{T}_k$. The intention is to consider a generic class of *systematic models* $\langle \boldsymbol{T}, \boldsymbol{Tr}, \boldsymbol{V} \rangle$, where each instance of an action theory $\boldsymbol{T}$ provides a validation criterion for a dynamic system $\boldsymbol{V}$, and the translation $\boldsymbol{Tr}$ is sound.

5 Skills Acquisition as Multi-Layered Learning

In the proposed hierarchy of intelligent agent architectures, each implementation agent class may require learning of appropriate skills. The acquired skills must be suitable to behaviours defined within the class. Because the general behavioural objectives differ among classes, skills acquisition can be seen as a process applicable to each implementation agent class and defined with respect to its functions, i.e. as a process orthogonal to the design. This suggests to consider a hierarchical and multi-layered learning, where choice of learning methodology depends on the class functionality.

[3] ***distance***(a, b) is an auxiliary function with an obvious definition.

Learning for each class may still address the on-line and off-line types as described in [13, 24], but prevailing learning objectives will be distinct within each agent class.

For example, a *Tropistic* agent may learn how to link suitable partial sensory states with suitable effectors – using, for example, methods of evolutionary computing. On the other hand, a *Hysteretic* agent may learn how to fine-tune the *hysteretic-behaviour* function – employing, for instance, artificial neural nets or statistical regression. A *Task* agent may utilise a pattern recognition learning algorithm in order to classify behaviour instantiations into appropriate clusters, while an *Extended Hysteretic* or *Extended Task* agent may concentrate on learning which types of communication are most efficient, etc.

On-line version of each learning mechanism remains the least-defined at this stage. In general, learning methods must be adapted to cope with incremental adjustments during a match. For instance, reinforcement learning techniques can be fused with each learning mechanism to achieve its optimal performance when on-line information becomes available.

6 Implementation and Conclusions

The described hierarchical framework has been mapped to the RoboCup Simulation League domain, resulting in an implementation of the Cyberoos'98. The Cyberoos'98 is a soccer team of heterogeneous software agents, instantiated from the implementation classes $\{ \boldsymbol{A}_S, \boldsymbol{A}_T, \boldsymbol{A}_H, \boldsymbol{A}_{EH} \}$. We are using GNU g++ 2.8.2 as an implementation language. The development environment is the Solaris 2.5 operating system running on SPARC work-stations. A Cyberoos'98 player is a multi-threaded software agent which uses native Solaris threads for sensing, reasoning and acting. It is early at this stage to report tournament results as the RoboCup Pacific Rim Series 98 is the first Cyberoos'98 experience.

References

1. Baral, C., and Son, T. Relating Theories of Actions and Reactive Robot Control. In Proceedings of the AAAI 1996 Workshop on Theories of Action and Planning: Bridging the Gap. Portland (1996).
2. Booch, G. *Object-Oriented Analysis and Design, with Applications*, Second Edition, Benjamin/Cummings (1994).
3. Brooks, R.A. Elephants Don't Play Chess. In Maes, P. (ed) *Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back*, Mass.: MIT/Elsevier (1990) 3-15.
4. Brooks, R.A. Intelligence Without Reason. In Proceedings of the Twelfth International Joint Conference on Artificial Intelligence, Morgan Kaufmann (1991) 569-595.
5. Chaudhri, V.K, Farquhar, A., Fikes, R., Karp, P.D., Rice, J.P. Open Knowledge Base Connectivity 2.0.2, http://ontolingua.stanford.edu/okbc/.
6. Genesereth, M.R., and Nilsson, N.J. *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann (1987).
7. Genesereth, M.R. Knowledge Interchange Format. In Allen, J., Fikes, R., and Sandewall, E. (eds) Proceedings of the Conference of the Principles of Knowledge Representation and Reasoning (1991).
8. Ginsberg, M.L., and Smith, D.E. Reasoning about Action I: A Possible Worlds Approach. *Artificial Intelligence* 35: (1988) 165-195.
9. Gruber, T. Ontolingua: A Mechanism to Support Portable Ontologies. Technical Report, KSl 91-66, Stanford University, Knowledge Systems Laboratory (1992).
10. Gruber, T. A Translational Approach to Portable Ontologies, Knowledge Acquisition, Vol 5, No 2, 199-220 (1993).
11. Guarino, N., Giaretta, P. Ontologies and Knowledge Bases: Towards Terminological Clarification, in Proceedings of the KB&KS'95 Conference, the Netherlands (1995).
12. Kaelbling, L.P. and Rosenschein, S.J. Action and planning in embedded agents. In Maes, P. (ed) *Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back*, Mass.: MIT/Elsevier (1990) 35 - 48.

13. Kitano, H., Tambe, M., Stone, P., Veloso, M., Coradeschi, S., Osawa, E., Matsubara, H., Noda, I., and Asada, M. The RoboCup Synthetic Agent Challenge. In Proceedings of the 15th International Joint Conference on Artificial Intelligence, Nagoya (1997).
14. McCain, N., and Turner, H. A Causal Theory of Ramifications and Qualifications. In Proceedings of the 14th International Joint Conference on Artificial Intelligence, Montreal (1995) 1978-1984.
15. McCarthy, J., and Hayes, P. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In Meltzer, B., and Michie, D. eds. *Machine Intelligence*, vol. IV: (1969) 463 - 502.
16. Parsons, S., Sierra, C., and Jennings, N. Multi-context Argumentative Agents. In Proceedings of the Fourth International Symposium on Logical Formalizations of Commonsense Reasoning (1998).
17. Prokopenko, M., Jauregui, V. Reasoning about Actions in Virtual Reality. In Proceedings of the IJCAI-97 Workshop on Nonmonotonic Reasoning, Action and Change, Nagoya (1997) 159-171.
18. Prokopenko, M. Situated Reasoning of Software Agents (Preliminary Report). CMIS Technical Report 98/119, Sydney (1998).
19. Reiter, R. On Closed-World Data Bases. In *Logic and Data Bases*, edited by H. Gallaire & J. Minker, 55 - 76. New York: Plenum Press (1978).
20. Sandewall, E. *Features and Fluents. The Representation of Knowledge about Dynamical Systems. Volume I.* Oxford University Press (1994).
21. Sandewall, E. Towards the Validation of High-level Action Descriptions from their Low-level Definitions. *Linköping electronic articles in Computer and Information science*, Vol. 1 (1996): 4.
22. Schreiber, A. Th, Wielinga, B.J, and Jansweijer, W.H.J. The KACTUS view on the 'O' word. In Proceedings of the IJCAI Workshop on Basic Ontological Issues in Knowledge Sharing (1995).
23. Steels, L. Exploiting Analogical Representations. In Maes, P. (ed) *Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back*. Mass.: MIT/Elsevier (1990) 71 - 88.
24. Stone, P., Veloso, M. A Layered Approach to Learning Client Behaviours in the RoboCup Soccer Server. Applied Artificial Intelligence, Volume 12 (1998).
25. Thielscher, M. Computing Ramification by Postprocessing. In Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, Montreal (1995) 1994-2000.