Language Support for Multi Agent Reinforcement Learning

Tony Clark
Aston University
Birmingham, UK
tony.clark@aston.ac.uk

Balbir Barn
Middlesex University
London, UK
b.barn@mdx.ac.uk

Vinay Kulkarni, Souvik Barat
TCS Research
Pune, India
vinay.vkulkarni@tcs.com
souvik.barat@tcs.com

ABSTRACT

Software Engineering must increasingly address the issues of complexity and uncertainty that arise when systems are to be deployed into a dynamic software ecosystem. There is also interest in using digital twins of systems in order to design, adapt and control them when faced with such issues. The use of multi-agent systems in combination with reinforcement learning is an approach that will allow software to intelligently adapt to respond to changes in the environment. This paper proposes a language extension that encapsulates learning-based agents and system building operations and shows how it is implemented in ESL. The paper includes examples the key features and describes the application of agent-based learning implemented in ESL applied to a real-world supply chain.

CCS CONCEPTS

• **Software and its engineering → Multiparadigm languages**.

KEYWORDS

Agents, Reinforcement Learning

ACM Reference Format:
Tony Clark, Balbir Barn, and Vinay Kulkarni, Souvik Barat. 2020. Language Support for Multi Agent Reinforcement Learning. In *13th Innovations in Software Engineering Conference (formerly known as India Software Engineering Conference) (ISEC 2020), February 27–29, 2020, Jabalpur, India.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3385032.3385041

1 INTRODUCTION

The current era of digitisation, enabled through the technologies underpinning the so-called Fourth Industrial Revolution such as ubiquity of sensors, big data, artificial intelligence and low latency telecommunications [29], has led to increasing design complexity, and result in systems that need to be deployed into an uncertain environment. Such complex systems, for example, production plants, logistics networks, IT service companies, and international financial companies, are complex systems of systems that operate in highly dynamic environments that require rapid response to change. The characteristic features of such systems include scale, complex interactions, knowledge of behaviour limited to localised contexts, and inherent uncertainty.

ISEC 2020, February 27–29, 2020, Jabalpur, India

ACM ISBN 978-1-4503-7594-8/20/02…$15.00
https://doi.org/10.1145/3385032.3385041

Our hypothesis is that these issues can be addressed through the use of multi-agent reinforcement learning (MARL) [13] which has traditionally been used in simulation to design controllers for robots, resource management and automated trading systems. Our proposal is that MARL is potentially a much more fundamental concept that can be used to address intra and inter system-complexity by allowing general systems to achieve an optimal behaviour through dynamic learning. Similarly, design problems arising from a desire to achieve an algorithmically deterministic solution can be addressed by leaving non-determinism in the run-time.

The use of MARL to develop and deploy general purpose systems raises many problems. The design of such systems requires agent goals to be expressed effectively and in such a way that they do not conflict. In any non-trivial system, agents need to interact and collaborate to achieve shared goals which is an area of active research [18]. A MARL-based system also needs to address composition as a first class function. A mechanism by which collaborating agents, each with their own learning capabilities can be combined in an optimised way is an essential requirement for a complex system of systems.

Language support for MARL-based system development is not widespread, for example AgentSpeak provides support for agents, but not reinforcement learning [4]. Recent research has been successful in integrating reinforcement learning and agent oriented programming [5], but without introducing a new language construct with the potential for static analyis. Typically reinforcement learning libraries provide dynamic APIs that limit the potential for tool supported verification and analysis of system definitions [32]. Semantic constructs have been proposed for agent-based reinforcement learning [3] without the associated language integration proposed in this paper. Our proposal is the same as that described by Simpkins *et al* [31] which is that programming languages, and thereby Software Engineering, will benefit by using reinforcement learning in order to become more *adaptive*. Our work goes further than their A^{2}BL language by showing how a strongly typed language can be extended with both an agent construct with inheritance and associated system building and transformation operations.

This paper provides a contribution to the issue of language support for MARL. Section 2 describes the motivation for our approach in more detail and establishes the problem to be addressed. It also describes approaches to MARL provided by other technology platforms, establishing that our approach is novel and that, if effective, it provides a contribution to system development. Section 3 establishes the requirements on a language-based approach and describes an extension to the language ESL that has been developed to support MARL. Section 4 shows how a supply chain can be implemented using the language feature together with a demonstration

of the potential of the approach by describing a real-life supply chain implemented in ESL.

Our work in this area has demonstrated that a MARL-based approach can be effective in the area of digital twins, but there is much work to do in broadening the domain of application. Section 5 presents concluding remarks and outlines future steps in our research plans.

2 BACKGROUND

2.1 Taming System Complexity

Complex systems of systems need to respond quickly to a variety of change drivers. The characteristic features of such systems include scale, complex interactions, knowledge of behaviour limited to localised contexts, and inherent uncertainty. The scale make it difficult to know the precise behaviour of the system in its entirety even when structure of the overall system in terms of the various localized modular units and relationships between them is known. Behaviour of the localized modular units is typically understood as event-condition-action chains. Moreover, incomplete knowledge and inherent uncertainty impart a stochastic nature to this understanding. Behaviour of the overall system emerges from the interactions of the behaviours of these modular units. Knowing how to analyse, control, adapt and design such systems is a difficult problem.

One approach that has been proposed to address this problem is to define a system in terms of a collection of autonomous, goal-driven, adaptive agents [6, 16, 27, 28]. Complexity and uncertainty can be addressed by such an approach by allowing the agents to adapt through learning mechanisms in order to achieve an overall system goal and to respond to changes in the operating environment. The use of agents can be used in: (1) analysis where an agent-based model is constructed in order to perform what-if scenarios, for example to see how the system should organise itself if goals change; (2) system design by building a simulation model of the desired system and then allowing the model to self-organise in order to produce parameters or modifications to be applied to the real-world implementation; (3) control where an agent-based model is constructed and connected to the real-world system in order to produce control inputs learned by observing behavioural differences between the ideal system and the running system.

2.2 Digital Twins

Along with AI-driven development, smart spaces and autonomous things, Gartner lists the Digital Twin concept as one of the top 10 strategic technology trends for 2019 [1]. Such a twin is a digital representation of a real-world entity or system [19]. Early definitions of digital twins have centred around mirroring products [24] for example the structural behaviour of an aircraft by analyzing and simulating the aircraft's behaviour on its digital model [34]. Three recent systematic reviews focussing on digital twins have exposed a number of other important facets [24, 25, 33] concluding that most papers are concerned with digital models and digital shadows while pure-play digital twins remain very much a future project.

Boscher et al [12] propose that next generation digital twins are those that are purposive for specific tasks but may be combined with other digital twins and where the feedback loop is closed not only to the real system but also to earlier lifecycle phases. Further, such next generation digital twins are coincident with the next generation of simulation technologies that support both human led interventions for optimisation as well as automated optimisation [36].

A high level abstract reference model based derived from a systematic review of the literature in [25] offers three components:"(1) an information model that abstracts the specifications of a physical object, (2) a communication mechanism that transfers bi-directional data between a Digital Twin and its physical counterpart, and (3) a data processing module that can extract information from heterogeneous multi-source data to construct the live representation of a physical object." Such a model offers routes to defining a roadmap for the development of appropriate technologies. For example, in the context of the Internet of Things, ubiquity of sensors and 5G technology enables an implementation route to the communication mechanism in the reference model. Like systems in complex environments, digital twins must address the dual problems of complexity and uncertainty. In order to achieve a high fidelity twin, the system must incrementally learn the behaviour of a partially understood system.

2.3 Actors and Multi-Agent Systems

The actor model of computation [2] has been shown to be effective when modelling large complex systems whose behaviour can only be known in localized contexts [7] and we have developed a technology platform called ESL[2] that has used actors to create digital twins of complex systems. By adding goals and intentionality to actors we arrive at intentional, autonomous, composable modular units called *agents*. An agent tries to achieve its stated goal by responding suitably to the events of interest and by exchanging messages with other agents and where behaviour may be stochastic *i.e.* there could be a probability distribution of actions associated with an event.

An agent observes the environment, makes sense of the observations, and performs actions so as to achieve its objectives. The action could change the local state of agent or send a message to other agents. These actions are often stochastic due to uncertainty and incomplete domain knowledge [8]. An agent should be capable of adapting its behaviour in response to the changes in its environment so that it is able to switch from one behaviour to another depending on the situation it finds itself in. An agent adapts and extends its behaviour not only to achieve its local objectives but also to ensure robustness of the overall system.

Multi-Agent Systems (MAS) have been used extensively as the basis for simulation [26] where the macro system behaviour is viewed as arising from the interactions of many different micro behaviours each of which is defined in terms of *perception*, *deliberation* and *action* as each agent interacts with its environment. Although MAS is a general concept, many of the agent-based technology platforms are designed for Artificial Intelligence and simulation [1]; many of

[1] gartner.com/smarterwithgartner/gartner-top-10-strategic-technology-trends-for-2019

[2] http://www.esl-lang.org

the languages and development environments supporting MAS are logic-based [11] and development support is generally weak.

As noted in [10] "The fact that multiple agents interact leads to a highly dynamic, non-deterministic environment. In such an environment, defining proper behaviour for each agent in advance is non-trivial and therefore learning is crucial." Reinforcement Learning (RL) is a promising approach that can easily be integrated with agent behaviour since it supports the incremental improvement of selecting local actions based on an agent's observation of a global environment. The integration of RL with MAS leads to Multi Agent Reinforcement Learning (MARL) which raises agent-specific challenges including the trade-off between independent and joint learning.

Although there have been many applications shown to benefit from MARL techniques, including simulation and digital twins, the language support for MARL appears to be at the level of libraries embedded in other languages, for example [37] which creates learning agents through an API implemented in Python and C++.

Our proposal is that MARL is a key technique that can be used both to address the creation of digital twins and to tame system complexity and uncertainty and therefore MARL should be considered a foundational part of Software Engineering when striving to deliver quality assured systems into highly dynamic ecosystems. However, there has been little study of how to integrate agents into programming languages in order for them to become first-class concepts together with type-safe agent construction operations. This paper proposes a MARL language extension to ESL together with system building operations. Before introducing the extension, this section concludes with a brief overview of reinforcement learning.

2.4 Reinforcement Learning

Reinforcement Learning (RL) is the problem faced by an agent that must learn behaviour through trial-and-error interactions with a dynamic environment [21]. Typically an RL model consists of a collection of states s, a collection of actions a that change the state and the environment, and a reinforcement signal r that tells an agent how well an action has performed in the environment. RL algorithms aim to maximise the reinforcement over the life-time of the agent.

Q-Learning [35] is an approach to implementing RL where a Q-table is used to retain information about the reward signal in order to select actions. Agents interact with their environment in two modes: *exploration* which takes random actions and *exploitation* which uses the Q-table to select the best available action at the time. The table maps states and actions to values: $Q(s, a) = v$. In any state s if we choose the action a with the highest value v then this should maximise the likelihood of achieving the goal.

The Q-table is populated incrementally, initially with a random value or 0.0. Then, each step from state s to state s' selects an action a based on the best value $Q(s, a)$ and updates the table:

$$Q(s, a) = (1 - \alpha)Q(s, a) + \alpha(r + \gamma \max_{a'}(Q(s', a'))) \quad (1)$$

where $0 < \alpha \leq 1$ is the *learning rate*, γ is the *discount factor* and r is the reward received by performing a in state s.

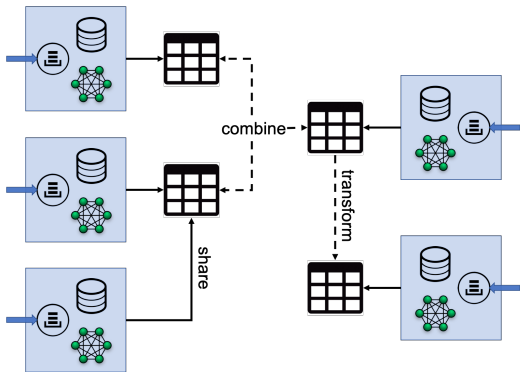


Figure 1: Agent Based Reinforcement Learning

3 LANGUAGE SUPPORT

Conventional approaches to agent-based reinforcement learning use a collection of synchronised agents to produce a single policy based on their collective state and behaviour. Typically this is achieved using a standard Q-table based algorithm, without any intrinsic language support. The lack of such support makes static checking of the relationship between the agents, their states, behaviour and actions impossible. Furthermore, if our proposal for learning based agents is key to addressing the issues of complexity and uncertainty in systems is correct, then technology platforms require a range of operations for building and manipulating agent-based systems.

Consider the situation shown in figure 1 which shows a collection of agents each with a message queue, a state and a behaviour. The behaviour is envisaged as a non-deterministic state machine whose transitions are controlled by a Q-table. In order to take this approach to systems development, it will be necessary to statically check agents, to describe how agents can be combined and disaggregated, to share learning between agents, and to transform learning by one agent so that it can be used by others.

This section describes a language construct for defining reinforcement learning based agents in ESL. The construct is fully integrated with the ESL type system and therefore the state, behaviour and actions of an agent can be checked. In addition, we describe a collection of operations that can be used as shown in figure 1 to build agent-based systems in ESL. All the features in this section have been implemented and we present some simple examples and demonstrate the learning through Q-tables and reward graphs.

3.1 Agents

Agents communicate by sending messages which are queued when they are received. Each message is processed in order and each system cycle, all agents handle the next message in the queue (or **Tick** if no message is present). An agent is in a particular state and a message causes the agent to update its state and perform an action that can send messages to any agent in the system.

An agent has the following type **Agent**[**S**,**M**,**A**] and consists of a state s::**S**, a collection of messages m::**M**, and a collection of actions a::**A**. The behaviour of an agent is a possibly non-deterministic state machine where the states are labelled with values of type **S** and whose transitions are labelled with message-action-value triples.

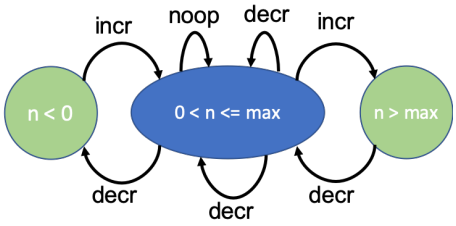


Figure 2: Counter Behaviour

Given an agent in state s, that receives a message m, the transition `s -(m,a,v)→ s'` with *best* value v is selected leading to a new agent state `s'` such that action `a(s,m)=s'`. The meaning of *best* depends on a global parameter that determines whether it means *highest* or *lowest*. Where there are multiple transitions with the same best value, a transition is selected at random. ESL provides a language construct to create a singleton agent:

```
agent[best] name(args)::Agent[S,M,A] extends parent {
   ...learning parameters...
   ...local values and operations...
  states::[S] = ...
  messages::[M] = ...
  actions::[A] = ...
  init()::S = ...
  terminalState(history::[S])::Bool = ...
  reward(history::[S])::Float = ...
  p_m^i, p_s^i when b → {
    a_1^i : e_1^i
  | a_2^i : e_2^i
  | ...
  }
}
```

where *best* is + or − depending on whether the agent is trying to maximise or minimise the reward. The learning parameters control the reinforcement learning algorithm defined by equation 1. The definitions for `states`, `messages` and `actions` define those elements of the associated data types that will be used in the state transitions. The function `terminalState` returns true when the supplied execution history has terminated. The function `reward` returns a value calculated in terms of the current history.

The rest of the agent body defines a collection of i arms consisting of a message pattern p_m^i, a state pattern p_s^i, a guard `b` and expressions e_j^i each of which is labelled with an action a_j^i. When a message is processed, each arm is tried in turn. When the message and current state matches the patterns and the guard is true then *one* of the expressions is performed and returns the new state of the agent. Each expression is labelled with an action: the best action given the reinforcement learning strategy is selected. Initially, each action will have the same weighting, however as the agent is trained, some actions will be assigned a higher weighting than others.

The agent `counter` provides a simple example of many features:

```
type Sc = Int;
data Mc = Move;
data Ac = Inc | Dec | Noop;
agent[-] counter(goal::Int)::Agent[Sc,Mc,Ac]{
  explorationFactor::Float       = 0.9;
  explorationDecay::Float        = 0.9995;
  states::[Sc]                   = -1..goal+1;
  messages::[Mc]                 = [Move];
  actions::[Ac]                  = [Inc,Dec,Noop];
  init()::Int                    = 0;
  terminalState([])::Bool        = false;
  terminalState(n:_)::Bool       = n = goal;
  reward(n:h)::Float when n=goal = length[Int](n:h);
  reward(_)::Float               = 40.0;
  Move,n when n<0       → { Inc:n+1 }
  Move,n when n>goal    → { Dec:n-1 }
  Move,n when n<goal    → { Inc:n+1 | Noop:n | Dec:n-1 }
  Move,n when n=goal    → { Noop:n }
}
```

The under-specified behaviour of a counter is given in figure 2 where the state of a counter is n and the agent is driven by a single message **Move**. When the state is less than 0 or greater than a limit then the behaviour is deterministic: it increments or decrements the state. In between these limits, the behaviour is non-deterministic and may choose actions **Inc**, **Dec** or **Noop**.

The goal of the agent is specified by its terminal state: it stops when it reaches `goal`. The reward is calculated in each state. If the state is currently `goal` then the reward is the length of the agent's history. Otherwise the reward is `40.0`. Given that the agent is required to minimise the reward over the history, reinforcement learning will tend to favour shorter histories that lead to `goal`.

3.2 Training and Running Agents

The behaviour of an agent is defined as a collection of non deterministic transitions. Running such an agent will select actions at random. In order to ensure an agent achieves its goal, it must be trained. Training involves running an agent for several *epochs* using the reward function to evaluate the history of each run. ESL provides agent operations that can be used to defined training operations:

```
train[S,M,A](a::Agent[S,M,A],epochs::Int,steps::Int,m::[M]) =
  for epoch in 0..epochs do {
    initAgent[S,M,A](a);
    for step in 0..steps do
      if not(isTerminatedAgent[S,M,A](a))
      then {
        sendAgent[S,M,A](a,select[M](m));
        runAgent[S,M,A](a);
      }
  }
```

The operation `train` uses `initAgent` to reset the state of an agent at the beginning of each epoch using its `init` function. Providing the agent is not terminated, the epoch repeatedly sends a message selected at random to the agent and then runs the agent. The operation `runAgent` takes the oldest message from the agent's queue and performs the best transition based on the current Q-table which is then updated. Once trained, an agent can be run using the following operation:

```
run[S,M,A](a::Agent[S,M,A],steps::Int,m::[M],pp::(S)→Str) ={
  initAgent[S,M,A](a);
  stabiliseAgent[S,M,A](a);
  for y in 0..steps do {
    if not(isTerminatedAgent[S,M,A](a))
    then {
      sendAgent[S,M,A](a,head[M](m));
      m := tail[M](m);
      runAgent[S,M,A](a);
```

Agent Operation	Description
initAgent::(Agent[S,M,A])→Void	Reset an agent's state by calling init().
sendAgent::(Agent[S,M,A],M)→Void	Add a message to the end of an agent's queue.
runAgent::(Agent[S,M,A])→Void	Process a single message at the head of the queue.
isTerminatedAgent::(Agent[S,M,A])→Bool	True when current state satisfies terminatedState.
stabiliseAgent::(Agent[S,M,A])→Void	Turn off exploration.
agentState::(Agent[S,M,A])→[S]	Return the current agent history.
displayQTable::(Agent[S,M,A],Str,Str)→Void	Show the current Q-table for the agent.
agentPair::(Agent[S_1,M_1,A_1],Agent[S_2,M_2,A_2])→Agent[S_1 * S_2,M_1 * M_2,A_1 * A_2]	Create agent product.
agentFst::(Agent[S_1 * S_2,M_1 * M_2,A_1 * A_2]) → Agent[S_1,M_1,A_1]	Project first element.
agentsnd::(Agent[S_1 * S_2,M_1 * M_2,A_1 * A_2]) → Agent[S_2,M_2,A_2]	Project second element.
agentNil::Agent[[S],[M],[A]]	Empty agent list.
agentCons::(Agent[S,M,A],Agent[[S],[M],[A]])→Agent[[S],[M],[A]]	Extend agent list.
agentHead::(Agent[[S],[M],[A]])→Agent[S,M,A]	Head agent.
agentTail::(Agent[[S],[M],[A]])→Agent[[S],[M],[A]]	Tail agent.
agentQTable::(Agent[S,M,A])→[S * M * Hash[A,Float]]	Reify the Q-Table.
setAgentQTable::(Agent[S,M,A],[S * M * Hash[A,Float]])→Agent[S,M,A]	Set agent Q-Table.

Table 1: Agent Operations

States	Messages	Inc	Dec	Noop
-1	Move	264.15		
0	Move	249.072	277.907	264.242
1	Move	232.315	264.204	249.168
2	Move	213.682	249.121	232.323
3	Move	192.983	232.449	213.699
4	Move	170.05	213.691	193.189
5	Move	144.632	193.005	170.045
6	Move	116.277	169.897	144.349
7	Move	84.735	144.497	116.005
8	Move	49.629	115.901	84.925
9	Move	10.282	84.475	49.375
10	Move			0

Figure 3: QTable

```
      print[Str]('→ '+ pp(head[S](agentState[S,M](a))));
    }
  }
}
```

Suppose that we create a counter that starts at 0 and has a goal of reaching 10 in the shortest possible number of steps. The following ESL code trains the agent over 500 epochs of 3 steps each, displays the Q-table and then runs the trained agent for 10 moves:

```
let a::Agent[Sc,Mc,Ac] = counter(10);
    moves::[Ac] = [Move | n←0..10];
in {
  train[Sc,Mc,Ac](a,500,30,[Move]);
  displayQTable[Sc,Mc,Ac](a,'Table','#.###');
  run[Sc,Mc,Ac](a,10,moves,fun(i::Int)::Str i+'');
}
```

The Q-table for the trained counter is shown in figure 3. It shows that a counter in state 0 that receives a message **Move** will select the action **Inc** because 249.072 is the lowest value compared to 277.907 and 264.242. Repeated **Move** messages select **Inc** until state 10 is reached when **Noop** is the only available action.

The reward function for the agent counter produces the length of the history when the goal is reached. By minimising the reward, the training produces a policy that reaches the goal in the shortest number of steps. The training for three counter agents with goals 10, 15 and 20 is shown in figure 4. Initially, exploration causes the agent to jump around, but once the goal is discovered, the reinforcement learning algorithm causes the policy to increasingly focus on the shortest path.

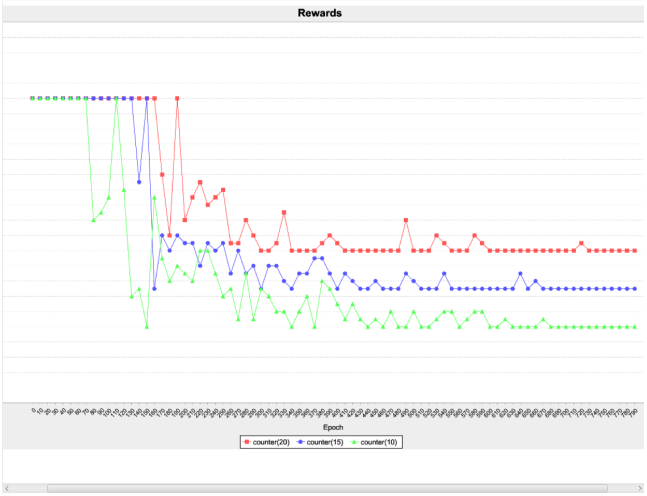


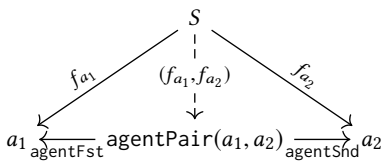
Figure 4: Training the Counter Agent

3.3 Building Systems

Single agents can be defined and trained and placed within a complex system. However, our proposal is that complex software systems that deal with uncertainty should be constructed from many collaborating agents, each of which uses reinforcement learning to adapt and incrementally improve. Therefore ESL provides a collection of system building operations so that composite agents can be constructed, trained and then decomposed.

The operation agentPair shown in table 1 can be used to construct a composite agent from two component agents. When two agents are composed, their Q-tables are also composed which allows training and composition to be performed in any order. The

result of composition produces a *product* agent which is the least constrained agent that can be projected back onto the component agents:



The effect is that agentPair(a_1, a_2) is the free combination of the behaviours defined by the corresponding Q-tables. The system S requires the product agent to be the *smallest* such agent. For example, two counter agents can be trained and then composed:

```
type Pair = Agent[Sc * Sc,Mc * Mc,Ac * Ac];
mkPair    = agentPair[Sc,Mc,Ac,Sc,Mc,Ac];
let a1:Agent[Sc,Mc,Ac] = counter(3);
    a2:Agent[Sc,Mc,Ac] = counter(2);
in {
  trainAgent[Sc,Mc,Ac](a1,100,20,[Move]);
  trainAgent[Sc,Mc,Ac](a2,100,15,[Move]);
  let a:Pair = mkPair(a1,a2);
      ms:[Mc * Mc] = [(Move,Move) | n ← 0..10];
  in {
    displayQTable[Sc*Sc,Mc*Mc,Ac*Ac](a,'Pair','#.###');
    run[Pos,Action](a,10,ms,fun(s:Int * Int):Str s+'');
  }
}
```

The Q-table produced by the composed agent is shown in figure 5. Notice that the values in the table are of the form (v_1, v_2). In general values can be tree shaped with floating point numbers at the leaves. Comparison between tree-shaped values is point-wise at the leaves, for example two trees are related by $<$ providing that all the leaf values are equivalently related.

3.4 Agent Inheritance

In order to support modularity and reuse, the ESL agent construct supports inheritance. Each agent may extend a parent. For example, the following is an agent that extends the composition of two counters to form a two-dimensional world. The following actor has the goal of finding a particular point in the world:

```
type P = Int * Int;
agent[-] target(w:Int,h:Int,p:P):Pair
  extends mkPair(counter(w),counter(h)) {
  messages:[Mc * Mc] = [(Move,Move)];
  init():Pos              = (0,0);
  terminalState([]):Bool  = false;
  terminalState(s:_):Bool = s = p;
  reward(s:h):Float       = length[Pos](h) + 1 when s=p;
  reward(_):Float         = 101.0;
}
```

The agent target inherits the learning behaviour of a pair of counter agents which, at the point of agent creation, are untrained. The inherited terminal state and reward of the inherited agent are overridden to introduce a new goal that aims to move towards the point p. Figure 6 shows the result of training three different target agents on grid-worlds of increasing sizes and with different goals.

3.5 Mapping Q-Tables

Once an agent has been trained, it may be desirable to reuse the learning in a different context, or to compose learning from multiple agents [30]. One such situation arises when a system requires multiple independent agents each of which has been trained as a single system: the system can be trained using a single agent with a state which contains the information from the composite agents, then the resulting single Q-table can be mapped to each individual agent in the system. Once mapped, the resulting agents use their individual Q-tables to guide their behaviour and communicate with other agents using message passing.

In order to support mappings, ESL provides two operations that can be used to process Q-tables: agentQTable and setAgentQTable such that a = setAgentQTable(a,agentQTable(a)). Given an agent of type Agent[S,M,A], its Q-table has type [S*M*Hash[A,Float]] where a value (s,(m,t)) represents the values attributed to each action when processing message m in state s. The following example defines an agent with a goal of reaching position (0,0):

```
data Ap = SubRight | SubLeft;
agent[+] pair(left:Int,right:Int):Agent[P,Mc,Ap] {
  init():P = (left,right);
  messages:[Mc] = [Move];
  states:[P] = [(n,m)|n ← -1..left+1,m ← -1..right+1];
  actions:[PairActions] = [ SubLeft, SubRight ];
  reward((0,0):_):Float = 100.0;
  reward(_):Float       = 0.0;
  terminalState((0,0):_):Bool = true;
  terminalState((l,r):_):Bool = l < 0 or r < 0;
  terminalState(_):Bool = false;
  Move,(l,r) → {
    SubRight  : (l,r-1)
  | SubLeft   : (l-1,r)
  }
}
```

The agent pair learns to navigate from (left,right) to (0,0). However, this is a single agent that achieves its goal without any collaboration with other agents. Suppose that we want two independent agents of the following type to collaborate in order to achieve the same goal:

```
data As  = SubSelf | SubOther | Skip;
data Ms  = Go(Int) | Tick;
agent single(n:Int,a:Agent[Int,Ms,As]):Agent[Int,Ms,As] {
  init():Int = n;
  messages:[Ms] = Tick:[ Go(o) | o ← -1..n+1 ];
  states:[Int] = -1..n+1;
  actions:[As] = [ SubOther, SubSelf ];
  reward(ss:[Int]):Float = 0.0;
  terminalState(0:_):Bool = true;
  terminalState(_):Bool = false;
  Go(o),s → {
    SubOther: { sendAgent[Int,Ms,As](a,Go(s-1)); s-1; }
  | SubSelf:  { sendAgent[Int,Ms,As](self,Go(o)); s-1; }
  }
  Tick,s → { Skip: s }
}
```

Note that the reward for the agent single is set to 0.0 because it will not be used for learning. The learning occurs in a pair agent whose Q-table is then mapped in two different ways to a left and right single agent. The mapping mapLeft is defined as follows:

```
mapLeft(state:P,tp:Hash[Ap,Float]):Int*Ms*Hash[As,Float] =
  case state {
```

States	Messages	(Noop,Noop)	(Noop,Dec)	(Noop,Inc)	(Dec,Noop)	(Dec,Dec)	(Dec,Inc)	(Inc,Noop)	(Inc,Dec)	(Inc,Inc)
(3,2)	(Move,Move)	(0,0)								
(3,1)	(Move,Move)	(0,45.295)	(0,81.971)	(0,5.984)						
(3,0)	(Move,Move)	(0,81.756)	(0,113.544)	(0,46.733)						
(3,-1)	(Move,Move)			(0,81.785)						
(2,2)	(Move,Move)	(51.258,0)			(84.671,0)			(4.787,0)		
(2,1)	(Move,Move)	(51.258,45.295)	(51.258,81.971)	(51.258,5.984)	(84.671,45.295)	(84.671,81.971)	(84.671,5.984)	(4.787,45.295)	(4.787,81.971)	(4.787,5.984)
(2,0)	(Move,Move)	(51.258,81.756)	(51.258,113.544)	(51.258,46.733)	(84.671,81.756)	(84.671,113.544)	(84.671,46.733)	(4.787,81.756)	(4.787,113.544)	(4.787,46.733)
(2,-1)	(Move,Move)			(51.258,81.785)			(84.671,81.785)			(4.787,81.785)
(1,2)	(Move,Move)	(83.267,0)			(115.679,0)			(46.085,0)		
(1,1)	(Move,Move)	(83.267,45.295)	(83.267,81.971)	(83.267,5.984)	(115.679,45.295)	(115.679,81.971)	(115.679,5.984)	(46.085,45.295)	(46.085,81.971)	(46.085,5.984)
(1,0)	(Move,Move)	(83.267,81.756)	(83.267,113.544)	(83.267,46.733)	(115.679,81.756)	(115.679,113.5...	(115.679,46.733)	(46.085,81.756)	(46.085,113.544)	(46.085,46.733)
(1,-1)	(Move,Move)			(83.267,81.785)			(115.679,81.785)			(46.085,81.785)
(0,2)	(Move,Move)	(115.658,0)			(144.536,0)			(83.18,0)		
(0,1)	(Move,Move)	(115.658,45.295)	(115.658,81.971)	(115.658,5.984)	(144.536,45.295)	(144.536,81.971)	(144.536,5.984)	(83.18,45.295)	(83.18,81.971)	(83.18,5.984)
(0,0)	(Move,Move)	(115.658,81.756)	(115.658,113.544)	(115.658,46.733)	(144.536,81.756)	(144.536,113.5...	(144.536,46.733)	(83.18,81.756)	(83.18,113.544)	(83.18,46.733)
(0,-1)	(Move,Move)			(115.658,81.785)			(144.536,81.785)			(83.18,81.785)
(-1,2)	(Move,Move)							(115.726,0)		
(-1,1)	(Move,Move)							(115.726,45.295)	(115.726,81.971)	(115.726,5.984)
(-1,0)	(Move,Move)							(115.726,81.756)	(115.726,113.544)	(115.726,46.733)
(-1,-1)	(Move,Move)									(115.726,81.785)

Figure 5: Composed QTable

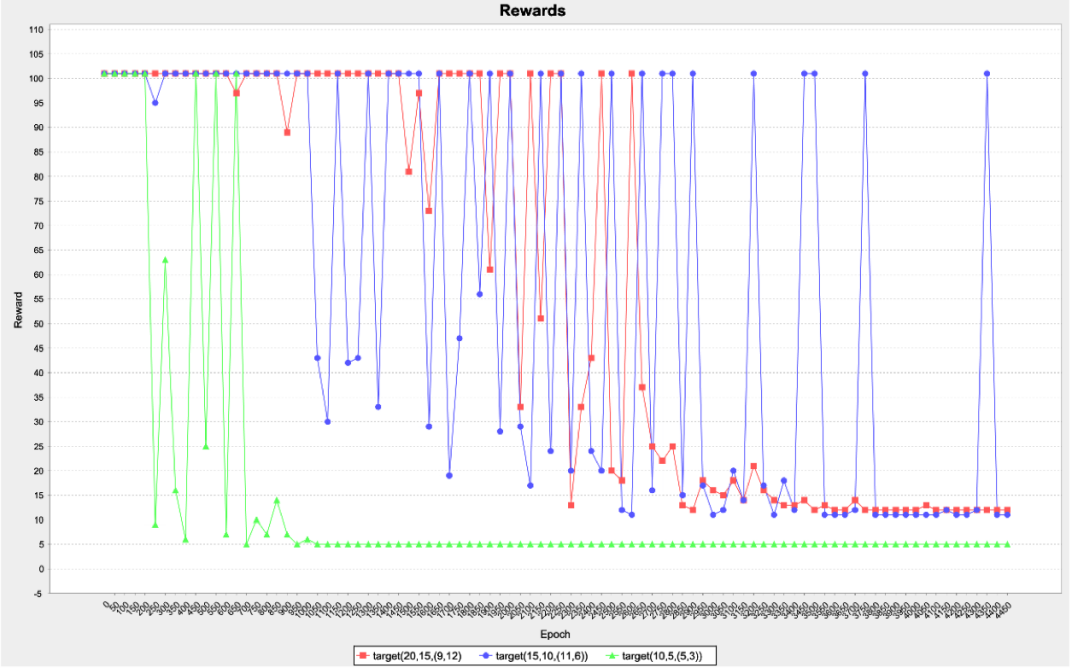


Figure 6: Training the Target Agent

```
  (l,r) → {
    let ts::Hash[Ap,Float] = new Hash[Ap,Float];
    in {
      for a in tp.keys do {
        case a {
          SubLeft → ts.put(SubSelf,tp.get(SubLeft));
          SubRight → ts.put(SubOther,tp.get(SubRight));
          _ → {}
        }
      }
      (l,(Go(r),ts));
    }
  }
}
```

An equivalent definition for `mapRight` allows two instances of `single` to be populated with Q-tables that are derived from a trained `pair` agent:

```
type Tp = [P * Mc * Hash[Ap,Float]];
type Ts = [Int * Ms * Hash[As,Float]]
let a::Agent[P,Mc,Ap] = pair(4,4);
    left::Agent[Int,Ms,As] = point(4,right);
    right::Agent[Int,Ms,As] = point(4,left);
in {
  trainAgent[P,Mc,Ap]](a,3000,9,[Move]);
  let q::Tp = agentQTable[P,Mc,Ap](a); in
  let ql::Ts = [ mapLeft(s,tp) |  (s,(Move,tp)) ← q ];
      qr::Ts = [ mapRight(s,tp) | (s,(Move,tp)) ← q ];
  in {
    initAgent[Int,Ms,As](left);
    initAgent[Int,Ms,As](right);
```

```
    setAgentQTable[Int,Ms,As](left,ql );
    setAgentQTable[Int,Ms,As](right,qr);
    stabiliseAgent[Int,Ms,As](left);
    stabiliseAgent[Int,Ms,As](right);
    sendAgent[Int,Ms,As](left,Go(4));
    for n in 0..20 do {
      if not(isTerminatedAgent[Int,Ms,As](left))
      then runAgent[Int,Ms,As](left);
      if not(isTerminatedAgent[Int,Ms,As](right))
      then runAgent[Int,Ms,As](right);
    }
  }
```

The resulting Q-tables in agents `left` and `right` cause both to collaborate by sending messages that arrive at the position `(0,0)` even though they have not been individually trained.

3.6 Implementation

ESL programs translate to Java source code which is then compiled and run using a standard Java compiler. All ESL values are instances of concrete sub-classes of `ESLVal` and ESL agents translate to an object of type `ESLAgent` that contains a message queue, a state, a Q-table, and a Java function that maps states `s` and messages `m` to *action-lists* of the form `[(a,f),...]` where `a` is an ESL value representing one of the actions possible when processing `s` and `m` and `f` is an *action-function* that will perform the action. The Q-table `q` is a Java hashtable of type:

```
Hashtable<ESLVal,Hashtable<ESLVal,HashTable<ESLVal,Action>>>
```

where `x = q.get(m).get(s)` is an action that contains an action function `f = x.getFunction()` and a value `v = x.getValue()` where a value is a tree of Java doubles contained in the Q-table. The Q-table is populated from the action-lists and then the values in the Q-table are updated by training according to equation 1.

4 A SUPPLY CHAIN AGENT

The previous section has introduced an extension to ESL that supports agent definition, composition, inheritance and transformation. RL has been used to learn optimal policies for supply chains which are inherently complex systems that can exhibit chaotic behaviour through sub-optimal ordering policies [22]. The MIT Beer Game is used to teach supply chain dynamics and has been the study of RL approaches [14]. This section shows how the Beer Game can be implemented using ESL agents and then describes how ESL has been used to implement a real-world supply-chain.

4.1 The Beer Game

The Beer Game [15] is a simulation used to teach students about the issues of supply chains. It is a linear supply chain where customers place orders for beer at regular intervals to a retailer who connects to a supply chain involving a wholesaler, a distributor and a factory. Storage costs are attributed to the nodes in the supply chain in addition to starvation costs when a node cannot supply product when requested.

The use of reinforcement learning to produce an optimal policy that predicts customer demand and thereby minimises the costs associated with storage and starvation is described in [14] which is typical of approaches by other researchers where a single policy is learned to control the co-ordination of supply chain nodes represented as agents. However, these approaches are not supported by a language construct for RL-agents, and there is no support for nodes as independent agents, both of which are supported by ESL as described in this paper.

Figure 7 shows a simplified version of the beer game implemented as a single agent in ESL with two supply nodes (compared to four in the standard game). Unlike other approaches, the key features of the states and actions are explicit in the definition. A state is a list consisting of a factory, two processing nodes and a customer: **Factory**`(n)` contains `n` units of beer to be supplied on the next cycle; **Processor**`(o,i,d,as,s)` records the number of units `o` to order on the next cycle, the current inventory `i`, a record of the units ordered `d`, a history of actions `as` and a number of units to supply on the next cycle `s`; **Customer**`(os)` containing a list of units `os` to order per cycle.

Each cycle of the agent involves two passes through the supply chain as defined by operations:

`supply(n,chain)` Each node in the chain is visited: `n` is the amount to be supplied to the node at the head of `chain`. The flow of supply is generated by the factory which has unlimited beer and each node decides on the supply pass how much it will send downstream on the next cycle. The customer simply consumes beer as it is supplied.

`order(as,n,chain)` Each node in the chain is visited in reverse: `n` is the amount to be ordered from the node at the head of `chain` and `head(as)` is an additional amount to add to the order. The schedule of all beer requirement for the whole cycle is controlled by the customer and each node stores the amount to order upstream on the next cycle.

The additional amount added to each order as supplied to `order` is the agent policy to be learned. It will aim to predict the future demand of the customer so that additional beer is ordered and thereby minimise future costs. The implementation of the agent in figure 7 provides four actions that control the policy: **ZeroZero**, **ZeroTwo**, **TwoZero**, **TwoTwo**.

The reward for each state is calculated in terms of the `storageCost` and the `starvationCost`:

```
chainReward(chain::Chain)::Float = {
  case chain {
    [] → 0.0;
    Customer(_):chain  →  chainReward(chain);
    Factory(_):chain   →  chainReward(chain);
    Processor(_,i,_,_,_):chain when i<0 →
      -i * starvationCost + chainReward(chain);
    Processor(_,i,_,_,_):chain →
      i * storageCost + chainReward(chain);
    _:chain →  chainReward(chain);
  }
}
```

The training for the simple Beer Game agent is shown in Figure 8. Four different customer demand profiles are shown increasing from `0`. The training is shown to stabilise and to find the best preemptive ordering profile for the demand. Although the example is simple, this demonstrates that the RL-agent construct in ESL can be used to design and verify a model. Furthermore, the method of transforming a single trained ESL agent to produce multiple communicating trained agents as shown in section 3.5 can be applied to the supply chain to produce individually trained supply node agents.

```
data Link   = Customer(orders::[Int])
            | Factory(supplies::Int)
            | Processor(onOrder::Int,inventory::Int,demand::[Int],actions::[Int],toSupply::Int);
data Asc    = ZeroZero | ZeroTwo | TwoZero | TwoTwo;
type Chain  = [Link];
agent[-] supplyChain(orders::[Int])::Agent[Chain,Mc,Asc] {
  init()::Chain                                = [Factory(0),Processor(0,0,[],[],0),Processor(0,0,[],[],0),Customer(orders)];
  states::[Chain]                              =  ...generate all states...
  messages::[Mc]                               = [Move];
  actions::[Asc]                               = [ ZeroZero, ZeroTwo, TwoZero, TwoTwo ];
  reward(chains::[Chain])::Float               = sum([ chainReward(chain) | chain ← chains ]);
  terminalState(c::[Chain])::Bool              = length(c) = length(orders);
  processChain(a::[Int],chain::Chain)::Chain = reverse(order(reverse[Int](a),0,reverse(supply(0,chain))));
  supply(n::Int,chain::Chain)::Chain = {
   case chain {
     Factory(pending):chain → Factory(0):supply(pending,chain);
     [Customer(demand)] → chain;
     Processor(o,i,d,as,s):chain when i+n <= 0   → Processor(o,i+n,d,as,0):supply(s+n,chain);
     Processor(o,i,d,as,s):chain when i < 0      → Processor(o,i+n,d,as,0):supply(s+(n-i),chain);
     Processor(o,i,d,as,s):chain                 → Processor(o,i+n,d,as,0):supply(s,chain);
    }
  }
  order(actions::[Int],n::Int,chain::Chain)::Chain = {
    case actions,chain {
      actions,Customer(o:os):chain                          → Customer(os):order(actions,o,chain);
      [],[Factory(_)]                                       → [Factory(n)];
      a:actions,Processor(o,i,d,as,_):chain when i < 0      → Processor(n+a,i-n,n:d,a:as,0):order(actions,o,chain);
      a:actions,Processor(o,i,d,as,_):chain when i-n >= 0   → Processor(a,i-n,n:d,a:as,n):order(actions,o,chain);
      a:actions,Processor(o,i,d,as,_):chain                 → Processor(a + (n-i),i-n,n:d,a:as,i):order(actions,o,chain);
    }
  }
  Move,chain → {
    ZeroZero: processChain([0,0],chain)
  | ZeroTwo:  processChain([0,2],chain)
  | TwoZero:  processChain([2,0],chain)
  | TwoTwo:   processChain([2,2],chain)
  }
}
```

Figure 7: The Beer Game Agent

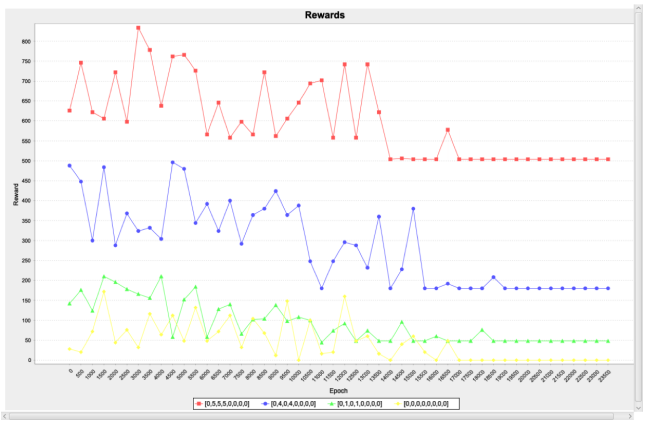


Figure 8: Training the Supply Chain

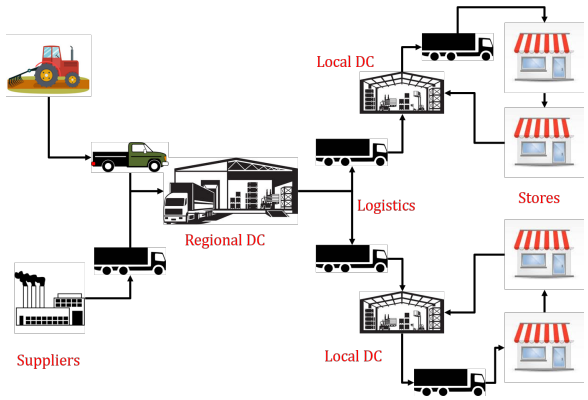


Figure 9: Schematic of supply chain replenishment use case

4.2 A Real Life Supply Chain

This section illustrates a scenario of a real grocery retailer that has been implemented using reinforcement learning using agents in ESL. A large grocery retailer is a network of network of stores (S), local distribution centers (LDC), regional distribution centers (RDC), and retailers (R) as shown in Figure 9. The end points of this network sell thousands of products to its customers, local distribution centers replenish the products to the stores, and regional distribution centers procure products from retailers and supply to local distribution centers. All elements in this network are connected through a fleet of trucks. The goal of the retailers is to regulate the availability of the entire product range in each store, subject to the spatio-temporal constraints imposed by (i) available stocks in the local and regional distribution centers, (ii) labour capacity for picking and packaging products in the all distributions centers, (iii) the volume and weight carrying capacity of the trucks, (iv) the transportation times between distribution centers and stores, (v)

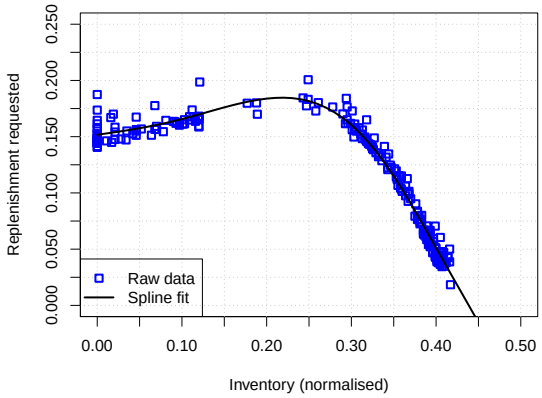


Figure 10: Replenishment Policy

the product receiving capacity of each store, and (vi) available shelf space for each product in each store. A typical retailer could have tens of regional distribution centers, hundreds of local distribution centers, thousands of stores, thousands of trucks, and a hundred thousand unique product types.

The overall state and behaviour of the networks emerge from individual behaviours and their interactions. For example, each warehouse may have its own policies to package products, utilise labours, load products into the trucks; each shop may have a set of situational policies to order products from warehouse. Trucks have their probabilistic characteristics to transport and deliver products. Each product type has its shelf-life, propensity for damage (when they are packed with other product types and transported in specific carrier/truck). Each product may be damaged, expired, in store, sold, *etc*).

Assume that there are m local distribution centers, p trucks, and n stores in the system. From operational perspective, each store stocks $i = \{1, \ldots, k\}$ unique varieties of products, each with a maximum shelf capacity $c_{i,j}$ where $j \leq n$ is the index of the store. Further, let us denote by $x_{i,j}(t)$ the inventory of product i in store j at time t. The replenishment quantities d are denoted by $a_{i,j}(t_d)$, and are to be computed at time $(t_d - \Delta)$. The observation $O(t_d - \Delta)$ consists of the inventory of each product in each store at the time, the demand forecast for each product between the next two delivery moments, and meta-data such as the unit volume v_i and weight w_i, and its shelf life l_i.

The reward $r(t_{d-1})$ is a function of the previous actions $a_{i,j}(t_{d-1})$ and the evolution of inventory states $x_{i,j}(t)$ in $t \in [t_{d-1}, t_d)$. From a business perspective, of particular interest are: (i) the number of products that remain available throughout the time interval $[t_{d-1}, t_d)$, and (ii) the wastage of any products that remain unsold past their shelf lives. We define this as:

$$r(t_{d-1}) = 1 - \frac{\text{count}(x_{i,j} < \rho)}{k\,n} - \frac{\sum_{i=1}^{k}\sum_{j=1}^{n} w_{i,j}(t_{d-1})}{\sum_{i=1}^{k}\sum_{j=1}^{n} X_{i,j}}, \quad (2)$$

where $\text{count}(x_{i,j} < \rho)$ is the number of products that run out of inventory (drop below fraction ρ) at some time $t \in [t_{d-1}, t_d)$, $w_{i,j}(t_{d-1})$ is the number of units of product i in store j that had to be discarded in the time interval because they exceeded their shelf lives, and $X_{i,j}$ is the maximum shelf capacity for product i in store j. Since both negative terms in (2) fall in the range $[0, 1]$, we see that $-1 \leq r(t_{d-1}) \leq 1$. The goal of the control algorithm is to compute actions $a_{i,j}(t_{d-1})$ that maximise the discounted sum of these rewards from the present moment onwards, $\sum_{z=0}^{\infty} \gamma^z r(t_{d+z})$.

Our initial observations are described in [9]. We developed a closed loop multi-agent simulation setup for training a reinforcement learning based control policy. A set of RL agents is realised using multi-layered neural network and Advantage Actor Critic (A2C) algorithm [23] to compute replenishment orders of the shops.

Figure 10 shows the replenishment quantity requested by trained RL agent as a function of the inventory level of the product. We note that the requested replenishment rises as inventory levels drop from 0.5. A majority of data points are located in the range between 0.3 and 0.4, where (i) the rate of wastage is lower than higher inventory levels, and (ii) there is no penalty for dropping below ρ (in this case it is 0.25). The requested replenishment decreases as inventory further reduces to 0, probably because the products have a low probability of recovery due to the severe transportation constraints (a meaningful learning).

The ESL actor based simulation is used as an environment to understand the overall implication locally optimum solution of multiple RL actions (produced for all individual shops) in a global system context. Our initial tests show that training using ESL is both feasible and effective. However, the RL realisation presented in [9] is specified using an external Python library - our next step is to implement the Python-based learning as a model using the features described in this paper.

5 EVALUATION AND CONCLUSION

This paper has motivated the need for Software Engineering to address the issues of complexity and uncertainty that arise due to dynamic software ecosystems and a desire to construct digital twins. The use of multi-agent systems and reinforcement learning is a promising approach. Whilst research has addressed different approaches to agent architectures and learning algorithms, there is little work addressing the extension of software languages with constructs for adaptive agent implementation and associated system construction. This paper has proposed an agent definition construct and operators that build systems from agents and has described the implementation of these in the language ESL.

Software Engineering aims to assure the quality of software through the use of verification techniques. Therefore, if MARL is to be a core part of system development, agents must not only be supported as first-class concepts, but there must be associated techniques for checking that agent-based systems are correct and exhibit the desired properties. The ESL agent construct and operations presented in this paper is a step in this direction by allowing the types associated with agents messages, state and actions to be statically checked. Further work is required to specify the behaviour of agents and to check this statically, especially in the context of learning mechanisms.

The real-world ESL-based supply chain implementation described in section 4.2 shows that MARL scales, however it uses an external library to provide *deep learning* via neural networks which approximate the Q-table function due to the size of the system state-space.

Such an approach is not consistent with the need to statically verify the properties of a single system. The next steps for this work are to support deep learning [17, 20] in ESL-based agent definitions by integrating neural networks with the Q-table mechanisms described in this paper.

REFERENCES

[1] Sameera Abar, Georgios K Theodoropoulos, Pierre Lemarinier, and Gregory MP O'Hare. Agent based modelling and simulation tools: A review of the state-of-art software. *Computer Science Review*, 24:13–33, 2017.

[2] Gul Agha, Ian A Mason, Scott Smith, and Carolyn Talcott. Towards a theory of actor computation. In *International Conference on Concurrency Theory*, pages 565–579. Springer, 1992.

[3] David Andre and Stuart J Russell. Programmable reinforcement learning agents. In *Advances in neural information processing systems*, pages 1019–1025, 2001.

[4] Amelia Badica, Costin Badica, Mirjana Ivanovic, and Dejan Mitrovic. An approach of temporal difference learning using agent-oriented programming. In *2015 20th International Conference on Control Systems and Computer Science*, pages 735–742. IEEE, 2015.

[5] Costin Badica, Alex Becheru, and Samuel Felton. Integration of jason reinforcement learning agents into an interactive application. In *2017 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pages 361–368. IEEE, 2017.

[6] Souvik Barat, Vinay Kulkarni, Tony Clark, and Balbir Barn. A method for effective use of enterprise modelling techniques in complex dynamic decision making. In Geert Poels, Frederik Gailly, Estefanía Serral Asensio, and Monique Snoeck, editors, *The Practice of Enterprise Modeling - 10th IFIP WG 8.1. Working Conference, PoEM 2017, Leuven, Belgium, November 22-24, 2017, Proceedings*, volume 305 of *Lecture Notes in Business Information Processing*, pages 319–330. Springer, 2017. ISBN 978-3-319-70240-7. doi: 10.1007/978-3-319-70241-4_21. URL https://doi.org/10.1007/978-3-319-70241-4_21.

[7] Souvik Barat, Vinay Kulkarni, Tony Clark, and Balbir Barn. An actor-model based bottom-up simulation: an experiment on indian demonetisation initiative. In *Proceedings of the 2017 Winter Simulation Conference*, page 61. IEEE Press, 2017.

[8] Souvik Barat, Vinay Kulkarni, Tony Clark, and Balbir Barn. A domain-specific language for complex dynamic decision making. In *The European Simulation and Modelling Conference, ESM 2017, Lisbon, Portugal, October 25-27, 2017*, 2017.

[9] Souvik Barat, Harshad Khadilkar, Hardik Meisheri, Vinay Kulkarni, Vinita Baniwal, Prashant Kumar, and Monika Gajrani. Actor based simulation for closed loop control of supply chain using reinforcement learning. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems*, pages 1802–1804. International Foundation for Autonomous Agents and Multiagent Systems, 2019.

[10] Daan Bloembergen, Karl Tuyls, Daniel Hennes, and Michael Kaisers. Evolutionary dynamics of multi-agent learning: A survey. *Journal of Artificial Intelligence Research*, 53:659–697, 2015.

[11] Rafael H Bordini, Lars Braubach, Mehdi Dastani, A El F Seghrouchni, Jorge J Gomez-Sanz, Joao Leite, Gregory O'Hare, Alexander Pokahr, and Alessandro Ricci. A survey of programming languages and platforms for multi-agent systems. *Informatica*, 30(1), 2006.

[12] Stefan Boschert, Christoph Heinrich, and Roland Rosen. Next generation digital twin. *Proceedings of TMCE, Las Palmas de Gran Canaria, Spain Edited by: Horvath I., Suarez Rivero JP and Hernandez Castellano PM*, 2018.

[13] Lucian Buşoniu, Robert Babuška, and Bart De Schutter. Multi-agent reinforcement learning: An overview. In *Innovations in multi-agent systems and applications-1*, pages 183–221. Springer, 2010.

[14] S Kamal Chaharsooghi, Jafar Heydari, and S Hessameddin Zegordi. A reinforcement learning model for supply chain ordering management: An application to the beer game. *Decision Support Systems*, 45(4):949–959, 2008.

[15] José Costas, Borja Ponte, David de la Fuente, Jesús Lozano, and José Parreño. Agents playing the beer distribution game: Solving the dilemma through the drum-buffer-rope methodology. In *Engineering Systems and Networks*, pages 337–345. Springer, 2017.

[16] Jacques Ferber and Olivier Gutknecht. A meta-model for the analysis and design of organizations in multi-agent systems. In *Multi Agent Systems, 1998. Proceedings. International Conference on*, pages 128–135. IEEE, 1998.

[17] Jakob Foerster, Ioannis Alexandros Assael, Nando de Freitas, and Shimon Whiteson. Learning to communicate with deep multi-agent reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 2137–2145, 2016.

[18] Fatemeh Golpayegani, Zahra Sahaf, Ivana Dusparic, and Siobhán Clarke. Participant selection for short-term collaboration in open multi-agent systems. *Simulation Modelling Practice and Theory*, 83:149–161, 2018.

[19] Michael Grieves. Digital twin: Manufacturing excellence through virtual factory replication. *White paper*, pages 1–7, 2014.

[20] Jayesh K Gupta, Maxim Egorov, and Mykel Kochenderfer. Cooperative multi-agent control using deep reinforcement learning. In *International Conference on Autonomous Agents and Multiagent Systems*, pages 66–83. Springer, 2017.

[21] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[22] Ahmet Kara and Ibrahim Dogan. Reinforcement learning approaches for specifying ordering policies of perishable inventory systems. *Expert Systems with Applications*, 91:150–158, 2018.

[23] Vijay R Konda and John N Tsitsiklis. Actor-critic algorithms. In *Advances in neural information processing systems*, pages 1008–1014, 2000.

[24] Werner Kritzinger, Matthias Karner, Georg Traar, Jan Henjes, and Wilfried Sihn. Digital twin in manufacturing: A categorical literature review and classification. *IFAC-PapersOnLine*, 51(11):1016–1022, 2018.

[25] Yuqian Lu, Chao Liu, I Kevin, Kai Wang, Huiyue Huang, and Xun Xu. Digital twin-driven smart manufacturing: Connotation, reference model, applications and research issues. *Robotics and Computer-Integrated Manufacturing*, 61:101837, 2020.

[26] Fabien Michel, Jacques Ferber, and Alexis Drogoul. Multi-agent systems and simulation: A survey from the agent commu-nity's perspective. In *Multi-Agent Systems*, pages 17–66. CRC Press, 2018.

[27] Geoffrey P Morgan and Kathleen M Carley. An agent-based framework for active multi-level modeling of organizations. In *International Conference on Social Computing, Behavioral-Cultural Modeling and Prediction and Behavior Representation in Modeling and Simulation*, pages 272–281. Springer, 2016.

[28] David V Pynadath and Milind Tambe. An automated teamwork infrastructure for heterogeneous software agents and humans. *Autonomous Agents and Multi-Agent Systems*, 7(1-2):71–100, 2003.

[29] Klaus Schwab. *The fourth industrial revolution*. Davos, World Economic Forum, 2017.

[30] Christopher Simpkins and Charles Isbell. Composable modular reinforcement learning. In *AAAI Conference on Artificial Intelligence*, 2019.

[31] Christopher Simpkins, Sooraj Bhat, Charles Isbell Jr, and Michael Mateas. Towards adaptive programming: integrating reinforcement learning into a programming language. In *ACM Sigplan Notices*, volume 43, pages 603–614. ACM, 2008.

[32] Brian Tanner and Adam White. Rl-glue: Language-independent software for reinforcement-learning experiments. *Journal of Machine Learning Research*, 10 (Sep):2133–2136, 2009.

[33] Fei Tao, He Zhang, Ang Liu, and Andrew YC Nee. Digital twin in industry: state-of-the-art. *IEEE Transactions on Industrial Informatics*, 15(4):2405–2415, 2018.

[34] Eric J Tuegel, Anthony R Ingraffea, Thomas G Eason, and S Michael Spottswood. Reengineering aircraft structural life prediction using a digital twin. *International Journal of Aerospace Engineering*, 2011, 2011.

[35] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4): 279–292, 1992.

[36] Stephan Weyer, Torben Meyer, Moritz Ohmer, Dominic Gorecky, and Detlef Zühlke. Future modeling and simulation of cps-based factories: an example from the automotive industry. *IFAC-PapersOnLine*, 49(31):97–102, 2016.

[37] Lianmin Zheng, Jiacheng Yang, Han Cai, Ming Zhou, Weinan Zhang, Jun Wang, and Yong Yu. Magent: A many-agent reinforcement learning platform for artificial collective intelligence. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.