Marlon – A Domain-Specific Language for Multi-Agent Reinforcement Learning on Networks

Tim Molderez Bjarno Oeyen Coen De Roover Wolfgang De Meuter
Vrije Universiteit Brussel
Brussels, Belgium
[tim.molderez,bjarno.oeyen,coen.de.roover,wolfgang.de.meuter]@vub.be

Abstract

Distributed systems can consist of thousands of network nodes interacting with each other. Given their size, managing these systems to perform optimally is a task that should be automated. Multi-agent reinforcement learning (MARL) is a suitable technique for tackling such problems. However, the application of MARL in a distributed system is not trivial. To bridge the gap between these two domains, we introduce the Marlon language. It enables MARL experts to focus on solving machine learning problems, rather than the complexities of distributed computing. We evaluate Marlon by comparing the implementation of a load balancing use case in Marlon with an ad-hoc implementation.

CCS Concepts • **Theory of computation → Multi-agent reinforcement learning**; • **Software and its engineering → Distributed programming languages**; **Domain specific languages**;

Keywords Multi-agent reinforcement learning, distributed systems, domain-specific languages

ACM Reference Format:
Tim Molderez Bjarno Oeyen Coen De Roover Wolfgang De Meuter. 2019. Marlon – A Domain-Specific Language for Multi-Agent Reinforcement Learning on Networks. In *The 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19), April 8–12, 2019, Limassol, Cyprus.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3297280.3297413

1 Introduction

This work targets the use of multi-agent reinforcement learning [8] (MARL) in a distributed context. That is, our primary focus is on the use of MARL where the environment is a distributed system, in a broad sense of the term. For instance, it includes applications intended for smart grids, traffic control systems, wireless sensor networks, or vehicle-to-vehicle communication networks. It is increasingly common for such different types of distributed systems to involve a growing number of networked devices. Due to their scale, it is not trivial to ensure such systems perform optimally in terms of e.g. throughput, reliability, latency or resource usage. Observing that each node in the network is an autonomous entity, MARL would be a natural fit for tackling this type of optimization problems. However, the use of MARL is complicated by a number of factors inherent to distributed systems: a cluster needs to be set up; nodes may need to share information; nodes may dynamically join/leave the network, lose their connection or crash. While MARL (and RL in general) is a widely applicable technique, these factors are often not considered. Moreover, to apply MARL to a *large* distributed environment, this also suggests that the learning process itself should be distributed as well.

This paper aims to bridge the gap between distributed systems and MARL by means of a domain-specific language (DSL) called Marlon[1]. Its purpose is to enable MARL experts to focus on machine learning problems, rather than issues related to distributed computing. Likewise, it enables distributed systems experts to make use of MARL without knowing the intricacies of machine learning.

Marlon is implemented on top of the Elixir language. Elixir is purpose-built for developing scalable, fault-tolerant distributed applications. Elixir programs run on top of the Erlang VM, which has a proven track record for large distributed systems (used by e.g. Amazon, Facebook, Whatsapp, Ericsson). One of the attributes that makes Elixir suitable for distributed systems is its support for actor-based concurrency, which plays a key role in Marlon as well. Throughout the paper, a running example is used: a load balancing system that is configured using a MARL algorithm. This example is also used in the evaluation section of the paper, in which we compare a Marlon implementation of the example to an equivalent implementation written directly in Elixir.

This paper contains the following contributions:

- We introduce the Marlon language, with the aim of addressing the challenges listed in Sec. 2. Its key concepts, actors and agents, are discussed in Sec. 3. We

SAC '19, April 8–12, 2019, Limassol, Cyprus

ACM ISBN 978-1-4503-5933-7/19/04…$15.00
https://doi.org/10.1145/3297280.3297413

[1]Marlon is available at https://gitlab.soft.vub.ac.be/smileit/marlon-dsl

provide an overview of Marlon, a formal syntax and an informal semantics in Sec. 5 and Sec. 6. The API to implement MARL algorithms is provided in Sec. 7.
- To evaluate whether Marlon simplifies the use of MARL in distributed systems, we compare a Marlon implementation of the load balancing example of Sec. 4 with an Elixir version. The evaluation is found in Sec. 8.

2 Challenges

This section enumerates the challenges involved in combining the two domains of MARL and distributed systems. We refer back to these challenges in the remainder of the paper.

(A) Terminology - A relation should be established between the different terminology used in the domain of MARL and that of distributed systems.

(B) Fault tolerance - A distributed system should be able to handle faults, e.g. due to an unreliable or broken network connection. This factor is often not considered in MARL.

(C) Dynamic environment - In many distributed systems, it is expected that network nodes can join or leave the network at any time. Some MARL algorithms do take into account a dynamic environment, but many do not.

(D) Distributed environment - To avoid introducing bottlenecks in a distributed system, communication is often done asynchronously. In reinforcement learning, the environment (the distributed system) would be typically modeled as a Markov Decision Process. This distinction between the two domains should be addressed.

(E) Cost of communication - There is a certain cost associated with communicating between different nodes in the network. As MARL agents can run on different network nodes and may need to share information, this communication cost should be considered.

3 Actors and agents

Before presenting Marlon, we briefly introduce the actor model of concurrency [1], and how it relates to MARL.

The actor model - When implementing a distributed system using the actor model, it consists of a collection of *actors*. Each actor runs in its own thread of execution, has its own state and its own behaviour. The state of an actor is isolated, meaning that an actor is only allowed to directly access and modify its own state. This prevents race conditions, i.e. two threads simultaneously attempting to modify the same data. Actors communicate with each other by sending messages. These messages are typically sent asynchronously. This avoids deadlocks, where one thread is waiting for another thread indefinitely. Finally, a major advantage of using actors: it is easy to distribute them across network nodes. From the programmer's perspective, there is no difference between sending a message to a local or to a remote actor.

Relation to MARL - When using MARL, each agent typically can only observe and interact with a part of the entire environment. As our environment consists of a collection of actors, each agent in Marlon must be attached to one actor, as depicted in Fig. 1. Note that not all of the actors must be associated with an agent, as these actors may not be relevant for the MARL problem being solved. The basic reinforcement learning cycle of each agent is the following. An agent first observes the state of its actor. The agent then chooses an action, which corresponds to sending a specific message to that actor. The actor then executes the chosen action by handling the message it received. Finally, the actor computes a reward and send it to its associated agent.

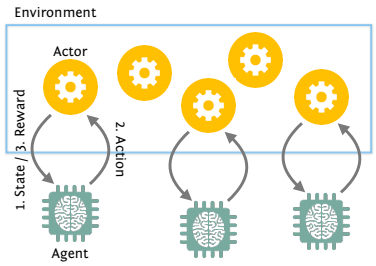

Figure 1. MARL process in an actor-based system

We can already discuss challenge (D) of Sec. 2, in which asynchronous communication is preferred in distributed systems. Agents are autonomous entities; they can act independently and do not necessarily need to synchronize their state-action-reward cycles. Marlon currently targets MARL approaches where such synchronization is not required.

4 Load balancing example

To illustrate the different constructs of Marlon, we will use a load balancing example as a running example. This example is based on the use case of Verbeeck et al. [11], where it was used to evaluate a MARL algorithm called "exploring selfish reinforcement learning".

An overview of the system is given in Fig. 2. It consists of a network with one master node and several worker nodes. The master has a certain job that needs to be distributed across these workers. Within this network, workers can join or leave at any time. The workers also run on heterogeneous hardware, i.e. some workers may be faster than others. The job that is initiated by the master can be subdivided into "chunks". As soon as a worker joins the network, it will request a chunk of work from the master (step 1 in Fig. 2). Next, the master will send the required data of that chunk to the worker (step 2). It is important to note that sending this data may take some time, and that multiple workers may request a chunk from the master simultaneously. In our implementation, the master handles these requests one at a time, so some workers may have to wait longer for their chunk. Once a worker has received its chunk, the worker can process it (step 3). When finished, the worker sends the results back to the master (step 4). Once this is done, the

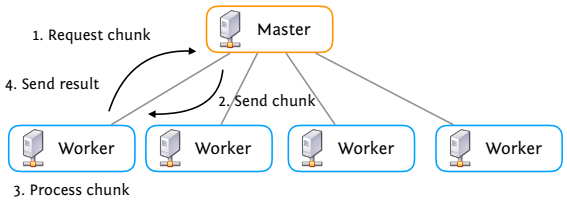

Figure 2. Load balancing use case overview

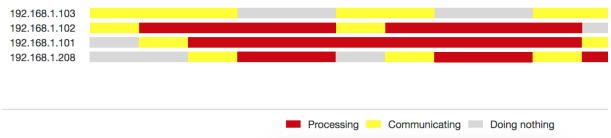

Figure 3. Load balancing visualization

actor	::=	`defactor` *type* `do` *member** `end`
member	::=	[*property*] `def` *identifier*`(`*identifier**`)` *block*
send	::=	*type*`.` *identifier*`(`*identifier**`)`
block	::=	`do` *stmt** `end`
property	::=	`sync` | `async` | `reply`
stmt	::=	an Elixir statement
atype	::=	a type name (in CamelCase)
identifier	::=	an identifier (in snake_case)

Figure 4. `defact or` syntax

worker will repeat the process until the master replies that the entire job is finished.

A snapshot of our live visualization of the system is given in Figure 3. It depicts a timeline of the activity of the master and all workers. The "Communicating" parts of the timeline indicate when chunk data is being sent/received.

The goal of this example is to use MARL to select the chunk size of each worker to approach an optimal system, in which neither the master nor the workers are ever idle. To achieve this, an agent is associated with each worker.

5 Distributed programming in Marlon

Having introduced our running example, we can now focus on representing the environment in Marlon.

5.1 Actor definitions

As mentioned in Sec. 3, an environment in Marlon is composed of actors. To define an actor, the `defactor` construct is used. Its syntax is specified in Fig. 4 (square brackets indicate an optional part; a * indicates repetition). Two concrete examples of actor definitions are given in Figures 6 and 7, which respectively define the behaviour of a master and a worker node in the load balancing example. We will walk through these examples in detail in the next section, Sec. 5.2. This section primarily focuses on introducing the different components of an actor definition:

`init` function - When spawning a new actor, its `init` function is called (line 2-5 in Fig. 6; line 2-5 in Fig. 7). Similar to a class constructor, it produces the actor's initial state. In case of the `Master` actor, its initial state is a dictionary/map that contains: the communication time (to simulate how long it takes to send a chunk of size 1), the number of chunks that have not yet been processed, how many received requests have not been processed yet, the list of workers that are currently processing a chunk. In case of the `Worker` actor, its initial state contains: a process ID of the master, its "speed factor" (to simulate that some workers are faster than others), how long it has to wait before the master responds to a request, how long it takes to process a chunk of size 1.

Apart from the `init` function, an actor can contain three types of message handlers: synchronous, asynchronous and reply message handlers:

`async` message handlers - When a message is handled by an `async`/asynchronous message handler, the actor that sent the message does not have to wait for any return value and can immediately continue. The first parameter of the `async` message handler, `this`, represents the actor's current state. Any other parameters are message parameters provided by the sender. When the message handler successfully returns, it returns a 2-tuple, where the first element always is `:noreply`. (If an error occurs, this element will be `:error`) The second element is the new state of this actor, i.e. the actor will have this state as soon as the message is handled.

`sync` message handlers - This is a synchronous message handler; the sender must wait until a result is returned. A `sync` handler has at least has two parameters, `this` and `from`: `this` is the current state of the actor. `from` is the process ID of the actor that sent the message. Any other parameters are the message parameters. A `sync` handler must return a tuple , where the first element is either `:reply` or `:noreply`. For `:reply`, the return tuple is a 3-tuple: the second element is the return value; the third is the new state of this actor. For `:noreply`, the return tuple is a 2-tuple, where the second element is the new state of this actor. A `:noreply` is used if the actual reply to the sender is provided later by a `reply` message handler. This is useful e.g., when the actor needs to wait before a reply can be sent, but it should still be responsive to handle any new messages while it is waiting.

`reply` message handlers - This handler is used when a reply is postponed by a `sync` message handler. The Elixir function `GenServer.reply` can be used to send a reply to the actor that originally sent a message to a `sync` handler.

5.2 Load balancing environment in Marlon

Having discussed the different types of message sends, we can now briefly walk through the code for the load balancing example to get a better idea of Marlon's use. The code that initiates the load balancing system, which is executed on the network node for the master, is given in Fig. 5. Please note that any commented-out lines (with a # prefix)

```
1 {:ok, m} = Master.start_link([2000],[])
2 Master.create_job(m, 10000)
3 Enum.map(Node.list(),
4   fn(node) ->
5     {:ok, w} = Worker.start_link_remote(
6       node, [m, random_int(1,5), 5000], [])
7     # Worker.attach_agent(w, ChunkSizeGoal)
8     Worker.start(w)
9   end)
```

Figure 5. Initiating the load balancing system

```
1  defactor Master do
2    def init([comm_time]) do
3      {:ok, %{comm_time: comm_time, chunks_remaining: 0,
4        pending_req_size: 0, chunks_in_progress: %{}}}
5    end
6
7    sync def create_job(this, _from, job_size) do
8      {:reply, :ok, %{this | chunks_remaining: job_size}}
9    end
10
11   sync def req_work(this, from, chunk_size) do
12     new_pending = this[:pending_req_size] + chunk_size
13     Worker.notify_wait_time(elem(from,0),
14       new_pending * this[:comm_time])
15     Process.send_after(self(),
16       {:req_reply, chunk_size, from}, this[:comm_time])
17     {:noreply, %{this | pending_req_size: new_pending}}
18   end
19
20   async def work_finished(this, worker) do ... end
21   async def work_cancelled(this, worker) do ... end
22   reply def req_reply(this, chunk_size, from) do ... end
23 end
```

Figure 6. Marlon code for the master actor

```
1  defactor Worker do
2    def init ([m, speed, chunk_time]) do
3      {:ok, %{master: m, speed_factor: speed,
4        wait_time: 0, chunk_time: chunk_time}}
5    end
6
7    async def start(this) do
8      Worker.process_chunk(self(), 1)
9      # Worker.do_action(self(), ChunkSizeGoal)
10     {:noreply, this}
11   end
12
13   async def process_chunk(this, chunk_size) do
14     result = Master.req_work(this[:master], chunk_size)
15     if result != :no_more_work do
16       Process.send_after self(), {:processed_chunk},
17         round(this[:chunk_time] *
18         chunk_size / this[:speed_factor])
19     end
20     {:noreply, this}
21   end
22
23   reply def processed_chunk(this) do
24     Master.work_finished(this[:master], self())
25     Worker.process_chunk(self(), 1)
26     # Worker.do_action(self(), ChunkSizeGoal)
27     {:noreply, this}
28   end
29
30   async def notify_wait_time(this, wait_time) do
31     new_state = %{this | wait_time: wait_time}
32     # Worker.update_reward(self(), new_state,
33     #   ChunkSizeGoal)
34     {:noreply, new_state}
35   end
36 end
```

Figure 7. Marlon code for a worker actor

can be ignored for now; these will be discussed once the MARL portion of Marlon is covered. Actors are spawned using the `start_link` Elixir function: line 1 first spawns a new master `m`. The `create_job` message is then sent to `m` on line 2, to create a new job of size 10000. The message handler for `create_job` is given in lines 7-9 of Fig. 6. Lines 3-9 of Fig. 5 iterate over every network node that this current node is connected to, and start a worker on that node using `start_link_remote`. The worker is then activated by sending it a `start` message. The `async` handler for the `start` message is given on lines 6-11 in Fig. 7, where the worker sends itself a `process_chunk` message with chunk size 1.

In the `process_chunk` handler, the worker first sends a request to the master to obtain a new chunk of size 1. This corresponds to step 1 of our overview figure, Fig. 2. As an immediate response, the master first tells the worker how long to wait before it will receive its chunk (line 13-14 of Fig. 6). The master computes this waiting time based on how many other workers sent a request that is not handled yet, multiplied by how long it takes to send a chunk, `this[:comm_time]`. As this example does not send any actual chunk data, this is simulated by letting the master wait, and only send itself a `req_reply` message after `this[:comm_time]`. In the `req_reply` handler (not shown due to space limitations), the master updates how much work still remains, and sends a reply to the worker whether it can go ahead with processing the chunk. Step 2 of Fig. 2 is now complete.

The worker can now process the chunk, i.e., step 3 of the overview figure. Chunk processing is simulated on lines 16-18 of Fig. 7 by waiting for an amount of time, based on the chunk size multiplied by the time it takes to process a chunk of size 1. Once the chunk has been processed, the worker sends itself a `processed_chunk` message, in which the worker reports back to the master that it has finished. This completes step 4 of the overview figure. Next, the worker restarts the entire process by requesting a new chunk.

This concludes our overview of the entire load balancing code. Note that, up to now, we have not introduced anything novel, which is also intentional. The main takeaway of this section is that distributed system developers can develop their code without specific restrictions, i.e., without having to design the system with machine learning in mind.

```
goal ::=      defgoal type do
                marltype [marlparams]
                actions rewardfn [abstractfn]
                [shared] [deviation] end
marltype ::=  type type
marlparams ::= params [keyvalue*]
keyvalue ::=  identifier: expr
actions ::=   actions [msgpar*]
msgpar ::=    identifier: [parvals*]
parvals ::=   [expr*]
rewardfn ::=  reward fn(agent, state) -> stmt* end
abstractfn ::= abstraction fn(agent, state) ->
                stmt* end
shared ::=    shared [(:identifier)*]
deviation ::= share_deviation [keyvalue*]
expr ::=      an Elixir expression
```

Figure 8. Marlon syntax for the `defgoal` construct

```
1  defgoal ChunkSizeGoal do
2    type Marlon.ESRL
3    params [explorations: 7, steps: 20]
4    actions [process_chunk: [[1,2,3]]]
5    reward fn(_agent, worker_state) ->
6      1 / worker_state[:wait_time] end
7  end
```

Figure 9. Goal specification to optimize the chunk size

6 MARL integration

The integration of MARL in Marlon consists of two key parts. First, the Marlon developer should describe the problem to be solved with a *goal definition*. (Sec. 6.1) Second, this specification needs to be integrated with the distributed system for it to have an effect. (Sec. 6.2)

6.1 Goal specification

The syntax of `defgoal`, a goal definition, is provided in Fig. 8. Together with Fig. 4, this forms the complete syntax definition of Marlon. A concrete example of a `defgoal` for our load balancing example is given in Fig. 9. As this example shows, the definition of a goal can be quite concise. This is part of addressing the terminology challenge (A) of Sec. 2, in the sense that the developer is not required to interact with the MARL algorithm directly, and only needs to be aware of the basic "choose an action"-"update the reward" cycle to know about the terms action and reward. A goal definition consists of the following components:

`type` and `params` - The `type` component of `defgoal` specifies which specific MARL algorithm to use by providing its Elixir module name. The API to implement a MARL algorithm is discussed later in Sec. 7. In our example, we are using the exploring selfish reinforcement learning (ESRL) algorithm [11]. It is possible to configure the algorithm's parameters, using the `params` component. These parameters can be different depending on which algorithm is chosen. If a parameter value is not provided, a default value is used.

`actions` - The action space in Marlon is specified in terms of which messages an agent is allowed to send, together with the possible parameter values of that message. If the action space is `[a: [[1,2]], b: [["foo", "bar"], [3,4]]`, there are 6 possible actions. The agent can either send the message `a(1)` , `a(2)`, `b("foo",3)`, `b("bar", 3)`, `b("foo", 3)` or `b("foo", 4)`. *When* exactly the agent must choose an action is discussed in the next section. In our example of Fig. 9 (line 10), the agent can only send the `process_chunk` message, but it can choose between a chunk size of 1, 2 or 3 as the parameter of that message.

`rewardfn` - When an agent is asked to compute a reward based on the current state of its associated actor, this function is called. The reward function of our load balancing example is given on line 5-6 of Fig. 9.

`abstractfn` - An abstraction function receives the state of the associated actor as input, and returns an abstracted version of that state. Using an "abstraction function" is recommended for MARL algorithms that aim to learn the best action to take *for each possible state* of the associated actor. Because an actor can be in many different possible states (often an infinite number), the abstraction function is used to abstract away information from the current state to obtain a significantly smaller (and finite) state space. For instance, in our load balancing example, the only relevant field of a worker is the amount of time it should wait for a request. A suitable abstraction function would take a worker's state as input, and produce e.g. either `"long"`, `"medium"`, `"short"` or `"none"` to represent the waiting time. This reduces the number of possible states of a worker from infinity to 4.

`shared` and `share_deviation` - When MARL algorithms require that information is shared among all agents, the `shared` component can be used to list what information exactly is automatically shared among agents. If the list includes the `:action` keyword, the last action chosen by each agent is shared. If it includes `:reward`, the last reward value is shared. Any other keyword is interpreted as a field of the actor that an agent is associated with, which is then shared. For instance, `shared [:wait_time]` shares the waiting time of a worker. Whenever any shared information changes in one of the agents, the change is communicated to the other agents. Any information that is shared will then be available in the agent's reward computation function. We should note this currently is the only mechanism for agents to communicate with each other. MARL algorithms where agents need to negotiate with each other (in a synchronized manner), e.g. to agree on a joint action, are not supported yet.

Finally, there is the `share_deviation` component. To reduce the amount of communication among actors (challenge (E) of Sec. 2), an agent will only contact the other agents when the information of interest changes. If an agent is not contacted, it assumes the other agents' information has remained unchanged. To further reduce communication, `share_deviation` can also be used to specify how much the

value of an integer/float field is allowed to deviate before an agent will communicate the new value to other agents. For instance, `share_deviation [wait_time: 0.5]` will only share a changed waiting time if it changed more than 0.5.

6.2 Attaching agents to a distributed system

To make use of a goal definition, we should still attach agents to the distributed systems, and indicate when an agent must choose an action, and when it can compute its reward. To do this, every actor has three built-in functions: `attach_agent`, `do_action` and `update_reward`.

`attach_agent` - To attach an agent, we only need to add one line to the instantiation of the distributed system. The line `Worker.attach_agent(w, ChunkSizeGoal)` can now be uncommented in Fig. 5. The `w` parameter refers to a worker actor, and the `ChunkSizeGoal` goal was defined in Fig. 8. Note that it is possible to attach multiple agents to the same actor, but they must stem from different goals.

`do_action` - Next, we need to indicate *when* an actor must choose to perform an action. In our current implementation of the load balancing example, the worker always requests a chunk size of 1; this is done on line 8 and line 25 of Fig. 7. These lines can now be removed, and the `Worker.do_action` lines are uncommented. The agent associated with this actor can now choose from its action space (line 4 of Fig. 8), and perform this action. An action in Marlon always corresponds to sending a message. `Worker.do_action(self(), ChunkSizeGoal)` corresponds to sending a `process_chunk` message with either 1, 2 or 3 as the parameter.

`update_reward` - Finally, we still need to indicate when the agent can compute its reward, based on the associated actor's state and the action that was chosen. An `update_reward` call must always take place after a `do_action` call, as soon as the effect of the chosen action can be observed. In the load balancing example, the effect of choosing a specific chunk size can be observed once the worker knows how long it needs to wait before the chunk is sent. That is, when the worker receives a `notify_wait_time` message from the master. Lines 32-33 are now uncommented in Fig. 7. This `update_reward` call will compute the reward value using the reward function of lines 5-6 of Fig. 9.

6.3 Fault tolerance

Up to now, we can use Marlon to implement distributed systems that operate correctly in a fixed network in which no network failures occur. To address challenges (B) and (C) of Sec. 2, the system should also consider a dynamic network where nodes can join or leave, where leaving can occur intentionally or due to failure. Marlon makes use of the libcluster library, which can be configured to automatically detect nodes joining or leaving the network. It does this using a simple "gossip" protocol, where nodes confirm their presence by broadcasting a message using multicast UDP to the other nodes at a fixed time interval. If a node fails, this can be detected eventually by the absence of this message. If a node leaves intentionally, it can immediately announce its leaving. A Marlon application can be configured such that a message is sent to a user-specified actor whenever a node joins or leaves. This is used in the load balancing system to automatically start a new worker when a node joins, or to cancel a chunk being processed when a node leaves One drawback of relying one user-specified actor is that we assume the node this actor is running on is reliable. This could be resolved in future work by, for instance, specifying multiple actors and a leadership election protocol to determine which actor should handle nodes joining/leaving the network. Finally, the MARL algorithm is also notified whenever an agent joins or leaves the system. If an algorithm does not support a dynamic environment, this notification can be used to reset its learning progress. However, with regards to challenge (C), it should be noted that resetting the learning progress is not a suitable solution if nodes are joining/leaving so frequently that the learning algorithm does not have enough time to be of value. In this situation, it is necessary that the MARL algorithm is designed with a dynamic environment in mind.

7 Using existing MARL algorithms

The Marlon language in itself is not intended to write new MARL algorithms. Given that Marlon's host language, Elixir, is not commonly used for machine learning, we only expose an Elixir API for MARL algorithms. The idea is that an implementation of this API only acts as a thin wrapper around an existing MARL algorithm written in e.g. Python or C++. For instance, most of our implementations of the API call out to a C++ library called AI-toolbox[2], which provides a collection of (MA)RL algorithms. This enables MARL experts to continue developing algorithms in a familiar setting, without needing to focus on the intricacies of distributed computing.

The API for MARL algorithms is shown in 10. Note that, to address challenge (A) of Sec. 2, the API only uses MARL terminology and has no references to actors or messages.

`init_algorithm` is used for any initialisation code that is run once (per goal definition). `init_agent` can be used for initialisation code that is run whenever an agent is attached. `is_learning` returns false when a MARL algorithm has finished learning. `choose_action` requires the MARL algorithm to choose an action, in the form an integer. Whereas the action space is defined in terms of messages and their possible parameter values, Marlon internally maps each possible message + parameter-value combination to a unique integer, so the MARL algorithm can remain unaware of actor messages. `update_reward` is called whenever the agent has computed a new reward value. Marlon will ensure the desired (shared) state information is available in the agent's state at that point, which is why it is not necessary to pass

[2] https://github.com/Svalorzen/AI-Toolbox

```
defprotocol MarlAlgo do
  def init_algorithm(MarlAlgo, data) :: MarlAlgo
  def init_agent(this, init_state) :: MarlAlgo
  def update_reward(this, reward) :: MarlAlgo
  def is_learning(this) :: boolean
  def choose_action(this) :: integer
  def action_probabilities(this) :: map
  def best_action(this) :: integer
  def reset(this) :: MarlAlgo
  def agent_joined(this, pid) :: MarlAlgo
  def agent_left(this, pid) :: MarlAlgo
end
```

Figure 10. API of a MARL algorithm

it as a parameter. `reset` can be used to restart the learning algorithm. `action_probabilities` returns a map that indicates for each state-action combination, the probability that the given action is the best choice for the given state. `best_action` returns the action with the highest probability of being the best action, given the current state. Finally, `agent_joined` and `agent_left` are called to notify the algorithm whenever a specific agent and its associated actor have joined or left the network.

8 Evaluation

To evaluate whether Marlon effectively simplifies the use of MARL in a distributed environment, we have compared the Marlon implementation of our load balancing example with an implementation written directly in Elixir.[3] The Elixir version of the application is provided in two versions: the first version is simpler; the MARL algorithm does not require agents to communicate with each other, and it does not need to be notified whenever workers join or leave the network. In the second version, the chosen MARL algorithm does require that the agents share a part of the worker's state (i.e., how long a worker should wait before it receives chunk data). Additionally, in this version the algorithm also needs to be notified whenever a worker joins or leaves the network.

Fig. 11 shows a "zoomed-out" view of the entire source code for the three versions of the load balancing system, where each line of code is coloured to reflect a specific concern. The Marlon code is shown on the left; this version of the code counts 109 (non-empty) lines of code (LOC) in total. The first, simpler version of the Elixir code is shown in the middle (Elixir - v1); it has 168 LOC. The second version of the Elixir code is shown on the right (Elixir - v2), and has 202 LOC. Note that the Marlon version that is shown only corresponds to Elixir - v1. However, it can be easily updated such that it corresponds to Elixir - v2 by simply adding `shared[:wait_time]` to the goal specification.

The colour coding in Fig. 11 indicates five different concerns in the source code. The majority of the code, in all three

Figure 11. Comparison of load balancer implementations

versions, represents the distributed system itself; this code is not MARL-related. In Marlon, it occupies 92 LOC; in Elixir - v1: 116 LOC; in Elixir - v2: 120 LOC. For the other concerns, we will use a shorthand notation, (LOC: 92, 116, 120). The difference between Marlon and Elixir can be attributed to a more concise syntax of `defactor`.

Dynamic environment - (LOC: 6, 6, 15) As the Marlon and Elixir-v1 implementations both use libcluster to handle nodes joining/leaving, this code is identical. In Elixir-v2 more code is required because the MARL algorithm needs to be notified of nodes joining/leaving the network.

Agent interaction - (LOC: 4, 22, 23) This category refers to all communication with agents: attaching an agent, asking it for an action, or providing a new reward value. In Marlon, this communication is handled with one-liners, i.e. the `attach_agent`, `do_action` and the `update_reward` functions built into each actor (Sec. 6.2). In the Elixir versions, these are now implemented directly in the worker code. These Elixir implementations assume which goal the system is trying to achieve, which makes it less flexible to experiment with different goals.

Goal specification - (LOC: 6, 22, 22) The code labeled "goal specification" refers to all code that directly interacts with the MARL algorithm. In the Marlon version, this corresponds to the `defgoal` construct. The closest a Marlon developer can get to interacting with the MARL algorithm

[3]The code for the different implementations is available at: https://gitlab.soft.vub.ac.be/smileit/marlon-dsl/raw/master/res/Load-balancing.zip

is by providing its parameters, reward function and action space. In the Elixir versions, it is the developer's responsibility to know how to set up and interact with a MARL algorithm, which is also why its "goal specification" code is larger. (In Elixir-v2 this code has a few more LOC than v1, due to an extra function that notifies the algorithm that the network has changed.)

State sharing - (LOC, 1, 0, 12) To share parts of the worker state among agents, several parts of Elixir - v2 are affected. The master, initialization code, and workers are adapted to share the waiting time with all agents. Moreover, these modifications are specific to sharing the waiting time; substantial changes may be necessary to share other information.

The main takeaway of this evaluation is not per se that Marlon saves a large amount of LOC. While the effect is substantial in this small load balancing example, in a larger system the amount of MARL-related code may be much smaller than the amount of distributed system code. Instead, our conclusion is that Marlon provides a clean separation between the distributed system and any MARL-related code, whereas the MARL-related code is scattered throughout the distributed system code in the ad-hoc Elixir implementations.

9 Related work

To our knowledge there is no directly related work that aims to facilitate the use of MARL in distributed systems. However, there is related work that involves the two domains: Todd et al. [9] have developed two frameworks for multi-agent systems in Scala, one that uses an actor-based concurrency model, and one that uses a thread-based model. The two frameworks were mainly developed to compare the performance of the two models. While the thread-based framework shows better performance on a single machine, this may no longer be the case once multiple machines are involved. The work of Sujeeth et al. [7] presents the OptiML DSL. This work is related in the sense that it applies machine learning in a distributed setting. However, its main focus is on exploiting parallelism to improve the performance of machine learning algorithms, which may or may not be applicable to multi-agent environments. Our main focus is on making MARL more accessible for use in distributed systems.

The close relation between agents and actors has also been observed by Tošić et al. [10], Di Stefano and Santoro [3]. They make use of the relation to develop agent-oriented languages on top of actor-based languages, such as Erlang and Scala. While these languages have their origins in artificial intelligence, they are intended as a general-purpose language rather than to solve machine learning problems.

Next to general-purpose MARL frameworks, there also are frameworks specific to the domains of smart grids [5], communications networks [4], access control [2] or traffic signal control [6]. These frameworks may be more suited for these specific domains, but it could also be argued they may be less flexible to be deployed in a variety of different network environments.

10 Conclusion and future work

In this paper we have introduced the Marlon language to simplify making use of MARL in a distributed environment. Regarding future work, we would like to make use of Marlon to implement additional real-world use cases. This may require the language to be extended to cope with the needs of a specific distributed environment. There may also be types of MARL algorithms that do not fit our current API yet. Finally, it would be interesting to measure to what extent our current design scales to learning within a larger group of agents, or whether it is possible to effectively learn multiple goals simultaneously.

Acknowledgments

We would like to thank Eugenio Bargiacchi and Roxana Rădulescu for the fruitful discussions when designing Marlon.

References

[1] Gul Abdulnabi Agha. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems.* Ph.D. Dissertation. University of Michigan.

[2] Marco Cremonini, Andrea Omicini, and Franco Zambonelli. 2000. Coordination and Access Control in Open Distributed Agent Systems: The TuCSoN Approach. In *Coordination Languages and Models*, António Porto and Gruia-Catalin Roman (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 99–114.

[3] Antonella Di Stefano and Corrado Santoro. 2003. eXAT: an Experimental Tool for Programming Multi-Agent Systems in Erlang.. In *WOA*, Vol. 12. 1–127.

[4] J. Famaey, S. LatrÃľ, J. Strassner, and F. De Turck. 2010. A hierarchical approach to autonomic network management. In *2010 IEEE/IFIP Network Operations and Management Symposium Workshops*. 225–232.

[5] E. C. Kara, M. Berges, B. Krogh, and S. Kar. 2012. Using smart devices for system-level management and control in the smart grid: A reinforcement learning framework. In *2012 IEEE Third International Conference on Smart Grid Communications (SmartGridComm)*. 85–90.

[6] Fernando Santos, Ingrid Nunes, and Ana L.C. Bazzan. 2018. Model-driven agent-based simulation development: A modeling language and empirical evaluation in the adaptive traffic signal control domain. *Simulation Modelling Practice and Theory* 83 (2018), 162 – 187. Agent-based Modelling and Simulation.

[7] Arvind Sujeeth, HyoukJoong Lee, Kevin Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand Atreya, Martin Odersky, and Kunle Olukotun. 2011. OptiML: an implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*. 609–616.

[8] Ming Tan. 1993. Multi-agent reinforcement learning: Independent vs. cooperative agents. In *Proceedings of the tenth international conference on machine learning*. 330–337.

[9] Aaron B Todd, Amara K Keller, Mark C Lewis, and Martin G Kelly. 2011. Multi-agent system simulation in scala: An evaluation of actors for parallel simulation. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*.

[10] Saša Tošić, Dejan Mitrović, and Mirjana Ivanović. 2013. Agent planning in AgScala. In *AIP Conference Proceedings*, Vol. 1558. AIP, 362–365.

[11] Katja Verbeeck, Ann Nowé, Johan Parent, and Karl Tuyls. 2007. Exploring selfish reinforcement learning in repeated games with stochastic rewards. *Autonomous Agents and Multi-Agent Systems* 14, 3 (2007).