Towards a Domain-Specific Language for Automated Network Management

Tim Molderez
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
tim.molderez@vub.be

Coen De Roover
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
coen.de.roover@vub.be

Wolfgang De Meuter
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
wolfgang.de.meuter@vub.be

***Abstract*—Software applications involving networks, in a broad sense of the term, are becoming more complex and are deployed on a growing number of devices. These applications can involve wireless sensor networks, smart grids, intelligent traffic light systems, and so on. Manually managing such networks is becoming increasingly difficult. To automate this management process, this paper introduces the initial design of the Marlon domain-specific language. Marlon is suited to specify the desired management policies that should be achieved. It can automatically apply these policies using machine learning techniques, effectively reducing the amount of effort needed to manage such systems.**

***Index Terms*—domain-specific languages, multi-agent systems, machine learning**

I. Introduction

This work is situated in the context of software applications that are meant to be deployed on a network. We use the term *network* in a broad sense. While it includes the commonly used notion of computer networks, it also involves quite different environments such as wireless sensor networks, power grids or traffic light systems. Hardware plays a large role in such networked environments, but as there is growing need to make these environments "smart" (e.g. smart grids, intelligent traffic light systems), software is necessary. This software is becoming increasingly more complex, and it is deployed in an environment that can potentially scale up to millions of devices. As such, configuring and managing such systems, which is often done manually, does not come easy.

This paper introduces an initial version of Marlon (Multi-Agent Reinforcement Learning On Networks), a domain-specific language (DSL) that aims to simplify automating this management process. To achieve this goal, developers can use Marlon to specify a number of policies or goals that need to be attained. As the DSL's complete name implies, the automation itself is done using reinforcement learning [3], a machine learning technique. The network itself is represented in Marlon as a multi-agent system, a term from the domain of artificial intelligence. Each device in the network is then represented as an agent, which can be roughly defined as an entity that can act autonomously. We chose to implement Marlon as a DSL, with the aim of reducing the development and maintenance cost, compared to using general purpose-language to automate a specific multi-agent system. For instance, Marlon is designed to easily switch from a simulated multi-agent system to deployment in a real environment. It also is possible to specify and combine multiple machine learning goals, without depending on which specific machine learning algorithm is used.

Tim Molderez is supported by the FWO-SBO-SMILE-IT project, funded by the Research Foundation Flanders (FWO)

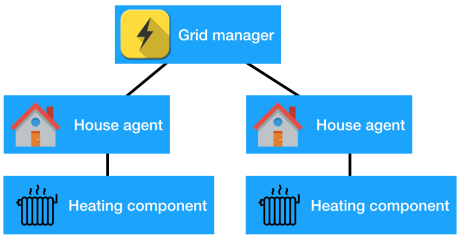

Fig. 1. Overview of a simple smart grid system

Marlon is a DSL implemented on top of the Elixir[1] language. Elixir was chosen as the host language for three main reasons. First, it focuses on building distributed, fault-tolerant systems. Elixir leverages the Erlang VM, which has a proven track record of scaling to very large systems, used by services such as Amazon and WhatsApp. Second, Elixir implements the actor concurrency model, where each actor/process is isolated and can only communicate with other actors via messages. This model coincides well with multi-agent systems, such that each agent corresponds with an actor. Finally, we chose Elixir because it has been designed with extensibility and domain-specific languages in mind. As such, a prototype implementation of Marlon, which essentially consists of a set of macros, could be developed in a short time frame.

The remainder of this paper introduces Marlon by means of an example in Sec. II, and we briefly describes its informal semantics in Sec. III.

II. Smart grid example

To illustrate the use of Marlon, we will discuss a small example in this section. This example is situated in the context of smart grids, i.e. an electrical grid where power

[1]https://elixir-lang.org/

usage/production is monitored with the aim of making a more efficient use of the available energy. An overview of the example system is given in Fig. 1: it consists of a grid manager and multiple houses each having a central heating system. The role of the grid manager is to provide power to each house, and to keep track of the total power usage. Each heating system keeps track of how much power it consumes, and its current temperature. The only policy we want to deploy in this example is that each house should reach and maintain its desired temperature.[2]

The entire Marlon source code that specifies how to simulate this system is given in Fig. 2. It also is possible to use Marlon to deploy this system in a real environment, but this is not discussed in more detail in this paper. Before examining the code of Fig. 2 in more detail, it is important to note that multi-agent systems commonly are modeled as *discrete* systems, which is also reflected in the design of Marlon. It means that the execution of a multi-agent system corresponds to an infinite main loop, where each iteration computes the next state of the system, based on the state of the previous iteration.

The code in Fig. 2 consists of four separate sections, each defining a different part of the smart grid; the `defworld` statement on lines 1-10 specifies the grid manager; the `defcomponent` statement on lines 12-25 specifies the central heating system of a house; the `defagent` statement on lines 27-46 represents the specification of a house. Finally, the `defgoal` statement on lines 48-68 specifies the goal/policy that each house should reach its desired temperature.

The code that initializes the entire system is the following:

```
House.create :h1
House.create :h2
House.add_goal :h1, ReachDesiredTemp
House.add_goal :h2, ReachDesiredTemp
{:ok, world} = World.start_link()
World.set_behaviour world, GridManager
World.add_agent world, :h1
World.add_agent world, :h2
```

This code snippet creates two houses, installs the `ReachDesiredTemp` policy in each house, initializes the grid manager and adds the two houses to it.

We can now examine the code in Fig. 2 in some more detail. At each iteration of the simulation, the machine learning algorithm must first make a decision, based on the goals that have been specified. In our case, there only is the `ReachDesiredTemp` goal, installed on both houses. Line 49 states that the goal should choose between the behaviours of the `centralheating` component of a house, defined on line 33. The CentralHeating component itself (lines 12-25), has two behaviours: on or off. Let us assume that the machine learning algorithm has currently decided to choose the ”on” behaviour in both houses. These chosen behaviours are now executed: the CentralHeating component updates the state of the house it belongs to by raising its temperature and energy consumption (lines 14-20). After executing the chosen component behaviours, each agent executes its step function (lines 42-45) to make any further adjustments to its state. Once this is done, the grid manager can update its global state, based on each of the house’s states. More specifically, on lines 5-9, the grid manager computes the total power consumption of all houses. On this is done, one iteration of the system has finished, and the next one can start.

While walking through the code of this example, we have not explained much yet regarding how the machine learning algorithm works. The algorithm we have currently implemented is a basic Q-learning [7] algorithm, in which a “Q-table” is maintained to learn which *action* needs to be taken when the system is in a given state. In this example, there are only two actions: turning the heating component on, or off. Representing the system’s state is more complex: as the system can be in an infinite amount of different states, an abstraction must be defined over the state in order to create a finite amount of abstract states. This abstraction is defined on lines 51-57, in which the current system state is mapped to either -1, 0 or 1. The 1 value represents an abstracted state where the temperature is too hot; -1 is too cold, and 0 is just right. Once the abstracted state space, and the list of possible actions is defined, we only need to specify the reward function that computes a reward value for a given combination of current abstracted state, and the action that is taken. This function is defined in lines 58-66. This completes the specification of the `ReachDesiredTemp` goal. To illustrate the Q-learning algorithm in action, Fig. 3 shows how the temperature of one house (y-axis) changes per iteration (x-axis). The learning algorithm keeps increasing the heating system’s temperature, until it crosses the desired temperature (22 °C) in iteration 20, after which the temperature remains fairly stable. (Note that the temperature slowly drops when the heating is turned off due to line 43.)

III. Marlon overview

After illustrating Marlon with an example, we can now describe the language’s concepts and informal semantics in general terms.

The four main concepts used in the language are: world, agents, components and goals.

World - A Marlon multi-agent system has one ”world”, an actor that maintains any global state in the system, which is shared with all agents. The `input_data` and `output_data` fields (lines 2-9 in Fig. 2) respectively define which data the world receives from its agents, and which parts of its state are published to all agents.

Agent - An agent corresponds to an actor. The `fields` field (line 28) specifies an agent’s internal state. The `components` field lists which components are contained by this agent. The `input_data` and `output_data` fields respectively define which data the agent receives from the world, and which parts of its state are published to the world. An agent also defines a ”step” function; this function is used

[2]We chose this policy only for its simplicity. Marlon uses machine learning to apply this policy, but there are simpler methods to implement a thermostat. The use of machine learning can be demonstrated in more complex examples, such as a grid where energy is traded between houses, and the optimal selling price is learned. This is part of future work.

```
defworld GridManager, [
  input_data: [
    {:agents, :power_consumption, :as, :agents_power_consumption}
  ],
  output_data: [
    {:data, :power_consumption, fn (_global_state, knowledge) ->
      knowledge[:agents_power_consumption] |> elem 1 |> Enum.sum
    end}
  ]
]

defcomponent CentralHeating, [
  behaviour: [
    on: fn(component, _knowledge, agent_state) ->
      agent_state = %{agent_state |
        temperature: agent_state.temperature + 1,
        power_consumption: agent_state.power_consumption + 100
      }
      {component, agent_state}
    end,
    off: fn(component, _knowledge, agent_state) ->
      {component, agent_state}
    end
  ]
]

defagent House, [
  fields: %{
    temperature: 5,
    power_consumption: 0
  },
  components: %{
    centralheating: CentralHeating
  },
  input_data: [
    {:world, :power_consumption, :as, :world_power_consumption}
  ],
  output_data: [
    {:data, :power_consumption,
     fn(_components, agent_state, _knowledge) -> agent_state[:power_consumption] end}
  ],
  step: fn(_identifier, components, knowledge, agent_state) ->
    agent_state = %{agent_state | temperature: agent_state.temperature - 0.125} # Subtraction to account
        for colder outside temperature
    {components, agent_state}
  end
]

defgoal ReachDesiredTemp, [
  components: [:centralheating],
  attributes: %{target_temperature: 22},
  state_fields: [
    {:delta_temperature, [-1, 0, 1], fn(attributes, _knowledge, _components, agent_state) ->
      %{temperature: temperature} = agent_state
      %{target_temperature: target_temperature} = attributes
      Utils.sign(temperature - target_temperature) # +1 = too hot, 0 = ok, -0 = too cold
    end}
  ],
  reward: fn (attributes, _components, _old_components, _knowledge, _old_knowledge, agent_state,
      old_agent_state) ->
    target_temperature = attributes.target_temperature
    if (abs(agent_state.temperature - target_temperature) <= 1) do
      10000
    else
      old_difference = abs(old_agent_state.temperature - target_temperature)
      new_difference = abs(agent_state.temperature     - target_temperature)
      if (old_difference >= new_difference), do: 5, else: -500
    end
  end
]
```

Fig. 2. Marlon code of the example smart grid

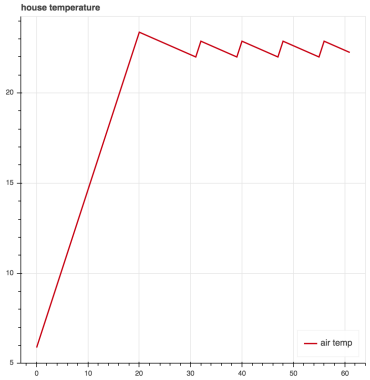

Fig. 3. Temperature evolution of a single house

```
1  step = 1
2  executeAndUpdate(step)
3
4  loop {
5          step++
6          Action selection + execution
7          executeAndUpdate(step)
8          Learning reward is computed
9  }
10
11 def executeAndUpdate(int x) {
12         Agents publish output data
13         World updates input data
14         World and all agents execute step x
15         World publishes output data
16         Agents update input data
17 }
```

Fig. 4. Pseudocode for the execution loop of a multi-agent system

to compute the agents' next state, based on its current state and the world's state.

Component - A component is part of an agent. It can (optionally) have its own internal state. It only contains a number functions that define the possible behaviours of this component. Only one of these functions is executed at each iteration of the system. Which function will be executed is determined by the machine learning algorithm. [3]

Goal - Finally, a goal specifies a desired property that an agent should reach, by means of a Q-learning algorithm. The `components` field (line 49 in Fig. 2) determines which components the machine learning algorithm can control. It is possible to attach multiple goals to the same component, but a weight function (not shown) should then be specified to determine which goal has the highest priority. The `attributes` field specifies any parameters that may be relevant to the goal. The `state_fields` field defines the abstract state space used by the Q-learning algorithm, together with a function that maps the current state to an abstracted state. Finally, there is the `reward` function that computes a reward value for the current state of the system, given the previous state.

As mentioned before, the multi-agent systems implemented with Marlon are discrete. The execution of such a system corresponds to a loop where each iteration represents the system's next state. The pseudocode in Fig. 4 gives a more precise idea of what happens in each iteration: first, for each goal, an action/behaviour is selected from the components it may affect. This selection is then executed. Next, all agents make their output data available to the world, which the world uses to update its input data. After this, all agent execute their step function. Once this is done, the world publishes its output data, and makes it available as the input data for all agents. The computation of the system's new current state is now finished, and all that remains is to use the reward function of each goal to compute how effective its chosen action was.

[3]Alternatively, it also is possible to write your own function that chooses which behaviour is executed, rather than letting the machine learning algorithm choose.

IV. Related work

Regarding related work, there are several existing frameworks and domain-specific languages that cater to specific types of multi-agent systems:

For instance, Frenetic [2] and Nettle [6] focus on programming computer networks. TeenyLime [1], TinyDb [5] and Semantic Streams [8] tackle querying and composing data in the area of wireless sensor networks. Whereas these papers do not involve machine learning techniques to manage networks, the work of Kara et al. [4] presents a learning-based framework to automate smart grid management. While the example we presented is also situated in a smart grid context, our aim for Marlon is to focus on the more general domain of multi-agent systems.

V. Conclusion and future work

This paper has presented an initial version of Marlon, a DSL for automating the management of multi-agent systems. The DSL was illustrated by means of an example in a smart grid context. As this initial version of the language was also developed starting from this context, one direction of future work is to apply the language in other types of multi-agent systems, and to evolve and extend the language with new features on an as-needed basis. We also need to evaluate the language in terms of its expressiveness, how it compares to frameworks/DSLs that focus on a specific domain, and how effective Marlon it is at reaching its machine learning goals. Another direction of future work is to add support for collaboration among agents, so it becomes possible to specify goals that span across groups of agents, rather than only specifying goals that apply to individual agents.

References

[1] Paolo Costa, Luca Mottola, Amy L Murphy, and Gian Pietro Picco. Teenylime: transiently shared tuple space middleware for wireless sensor networks. In *Proceedings of the international workshop on Middleware for sensor networks*, pages 43–48. ACM, 2006.

[2] Nate Foster, Rob Harrison, Michael J Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. In *ACM Sigplan Notices*, volume 46, pages 279–291. ACM, 2011.

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

[4] Emre Can Kara, Mario Berges, Bruce Krogh, and Soummya Kar. Using smart devices for system-level management and control in the smart grid: A reinforcement learning framework. In *Smart Grid Communications (SmartGridComm), 2012 IEEE Third International Conference on*, pages 85–90. IEEE, 2012.

[5] Samuel R Madden, Michael J Franklin, Joseph M Hellerstein, and Wei Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Transactions on database systems (TODS)*, 30(1):122–173, 2005.

[6] Andreas Voellmy and Paul Hudak. Nettle: Taking the sting out of programming network routers. *Practical Aspects of Declarative Languages*, pages 235–249, 2011.

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

[8] Kamin Whitehouse, Feng Zhao, and Jie Liu. Semantic streams: A framework for composable semantic interpretation of sensor data. *Wireless Sensor Networks*, pages 5–20, 2006.