Proceedings of the 2017 Winter Simulation Conference
W. K. V. Chan, A. D'Ambrogio, G. Zacharewicz, N. Mustafee, G. Wainer, and E. Page, eds.

HIERARCHICAL MARKOV DECISION PROCESS BASED ON DEVS FORMALISM

Celine Kessler
Laurent Capocchi
Jean-Francois Santucci

SPE UMR CNRS 6134
University of Corsica
Campus Grimaldi
20250, Corte, FRANCE

Bernard Zeigler

University of Arizona
Tucson, AZ 85721
Arizona Center for Integrative M&S
RTSync Corp.
12500 Park Potomac
MD, 20854, USA

ABSTRACT

Markov decision processes (MDPs) have proven useful as models of stochastic planning and decision problems. To try to propose practical implementation of MDPs, hierarchical methods are often used in MDPs or reinforcement learning to delegate the optimization of the total problem to simpler hierarchical sub-problems. The goal of the paper is to propose a generic discrete-event based software Framework allowing to use hierarchical MDPs and reinforcement learning to solve planning or decision problems. The proposed approach has been validated using the ”grid world” typical MDP use case.

1 INTRODUCTION

Planning and decision problems belong to an area of artificial intelligence that aims to produce plans for an autonomous agent, that is, a set of actions that it must perform in order to reach a given goal state from a known initial state. Because of the stochastic nature of the problems (the effects of the actions are stochastic, that is, the successor states of a given state for a given action are a probability distribution), the planning and decisions very often based on Markov Decision Process (MDP) (Bellman 1957). MDPs have proven useful as models of stochastic planning and decision problems. MDPs are remarkable for studying optimization problems solved via reinforcement learning. However the computer implementation of MDPs may require time polynomial in the size of the state and action spaces, and these spaces are generally too large to be explicitly enumerated. To try to propose practical implementation of MDPs, hierarchical methods are often used in MDPs or reinforcement learning to delegate the optimization of the total problem to simpler sub-problems. This allows for the use of prior knowledge to reduce the search space and provides a framework in which knowledge can be transferred across problems and in which component solutions can be recombined to solve larger and more complicated problems. The approach is to develop models of high level actions for MDPs that can be reused to solve multiple MDPs when objectives (or goals) change. In particular, high level actions are viewed as ”local” policies that are implemented until some termination condition is met, at which point a new high level action can be applied. Key to the success of this framework is the ability to construct models of high level actions that allow them to be treated as if they were ordinary actions in the original MDP.

The goal of the paper is to propose a DEVS-based approach allowing to use hierarchical MDPs and reinforcement learning to solve planning or decision problems. The genericity of the proposed approach based on the integration of MDPs, hierarchy concepts and reinforcement algorithms with the DEVS (Discrete Event System specification) formalism (Zeigler 1976, Zeigler et al. 2000). The DEVS formalism was introduced in the seventies for modeling discrete-event systems in a hierarchical and modular way. DEVS formalizes what a model is, what it must contain, and what it not contain (experimentation and simulation control parameters are not contained in the model). Moreover, DEVS is universal and unique for discrete-event system models. Any system that accepts events as inputs over time and generates events as outputs over time is equivalent to a DEVS model. Furthermore, we have also to mention that Markov chains can be easily taken into consideration using DEVS. Finite Probabilistic DEVS (FP-DEVS) (Nutaro and Zeigler 2015, Seo et al. 2015) is an extension of Finite Deterministic DEVS (FD-DEVS) (Hwang and Zeigler 2009). FP-DEVS allows the transition out of a state to be one of a finite set of possible states where the choice is made probabilistically whereas FD-DEVS makes an internal transition from a state to a state. FP-DEVS extends the FD-DEVS specification of the internal transition. The generic approach proposed in the paper has been defined using the DEVS concepts in order to: (i) model basic MDPs (basically one agent and one environment) and (ii) develop the concepts allowing to deal with a hierarchy of agents. To implement the proposed generic approach allowing to deal with hierarchical MDPs and reinforcement learning we propose to use the DEVS formalism and to give an implementation using PyDEVS (Van Mierlo et al. 2015, Van Tendeloo 2014).

The rest of the paper is as follows: section 2 gives the background on this work (mainly brief presentations of the DEVS formalism, the Markov Decision Process Framework and the hierarchical decomposition of MDPs). The generic proposed approach is described in section 3. Section 4 is dedicated to the case study while section 5 concerns the conclusion and future work.

2 BACKGROUND

2.1 DEVS Formalism

The classic DEVS (Discrete EVent system Specification) (Zeigler et al. 2000) formalism has been introduced as a mathematical abstract formalism for the modeling and the simulation of discrete-event systems allowing a complete independence from the simulator using the notion of abstract simulator. DEVS defines two kinds of models: atomic and coupled models. An atomic model is a basic model with specifications for the dynamics of the model. It describes the behavior of a component, which is indivisible, in a timed state transition level. Coupled models tell how to couple several atomic/coupled together to form a new model. This kind of model can be employed as a component in a larger coupled model, thus giving rise to the construction of complex models in a hierarchical fashion. As in general systems theory, a DEVS model contains a set of states and transition functions that are triggered by the simulator.

The behavior of a discrete-event system is a sequence of deterministic transitions between sequential states (S). An atomic model reacts depending on two types of events: external and internal events. When an input event occurs (X), an external event (coming from another model) triggers the external transition function $\delta_{ext}(X,S)$ of the atomic model in order to update its state. If no input event occurs, an internal event triggers the internal transition $\delta_{int}(S)$ of the atomic modeling in order to update its state. Then, the output function $\lambda(S)$ is executed to generate the outputs (Y). $t_a(S)$ is the time advance function which determine the life time of a state.

2.2 Markov Decision Process

A Markov Decision Process (MDP) (Bellman 1957) is a framework allowing to describe planning problems under uncertain environments. The model includes an environment and an agent: the agent takes decisions which modify the state of the environment. An MDP is completely defined by the following parameters:

- A finite set of discrete states $S = \{s_1, s_2, ..., s_n\}$ (finite set in this article).
- A finite set of actions (later referred to as *primitive* actions).
- Transition probabilities $P(s'|s,a)$, that are the probabilities for the environment to go from state s to state s' under the action a. These probabilities do not depend on previous history, only the current state S and the current action a matter (well known Markov property).

- Rewards attached to each state $R(s)$ (according to the definitions, rewards can be attached to transitions instead of states).
- $\gamma \in [0,1]$, the discount factor which represents the difference in importance between future rewards and present rewards.

The grid world is a classical example of MDP: the environment is a room represented as a table of cells, the agent is a robot, one cell in the room is the goal that the robot should reach. The state of the environment is the id of the cell in which the robot is. The robot can choose among 4 actions (GoNorth, GoEast, GoSouth and GoWest), but the result of its action is not deterministic (80% of the time, the requested action is realized, 20% of the time, the robot goes in an orthogonal direction). Finally, rewards are linked to the cells: the goal cell will report a positive reward, some cells to avoid will report negative rewards, other cells may report a small negative reward representing the cost an action, so that the trained agent should finally choose the shortest path to the goal.

The problem to solve is then to find the policy that will produce the maximum expected cumulative reward, called Value function $V(s) = E\left(\sum_{t=0}^{T} \gamma^t.R(s_t)\right)$.

A policy is expressed as a function $\pi(s)$ which returns the action to take when in a specific state. The optimal policy $\pi^*(s)$ is recursively defined by the following equations:

$$\pi^*(s) = \underset{a}{argmax} \sum_{s'} P(s'|s,a).V(s') \tag{1}$$

$$V(s) = R(s) + \gamma.\sum_{s'} P(s'|s,\pi^*(s)).V(s') \tag{2}$$

Equation 2 is called the Bellman equation. When all parameters are known, $S, A, P(s'|s,a)\, and\, R(s)$, the above equations can be solved and admit a unique solution. Several algorithms have been developed including value iteration (Bellman 1957), policy iteration (Howard 1960) and many variants of these two. In the value iteration algorithm, *V(s)* is iteratively estimated using Bellman's equation:

$$V_{i+1}(s) = R(s) + \underset{a}{max}\left(\sum_{s'} P(s'|s,a).\left(R(s') + \gamma.V_i(s')\right)\right) \tag{3}$$

$$\pi_{i+1}(s) = \underset{a}{argmax}\left(R(s) + \sum_{s'} P(s'|s,a).\left(R(s') + \gamma.V_i(s')\right)\right) \tag{4}$$

$V_0(s)$ is initialized with an arbitrary value and convergence toward V^* is proven.

When the transition probabilities, or the rewards, are unknown, the problem is called a reinforcement learning problem (Sutton and Barto 1998): the agent explores the state space, earns rewards and learns the optimal behavior from repeated experiences. Numerous algorithms intend to solve this problem. The most basic one is the Q-learning algorithm (Christopher J. C. H. Watkins 1992). The agent does not learn the transition probabilities but a Q-value that it updates after each transition from state *s* to state *s'* under action *a* as follows:

$$Q_{i+1}(s,a) = Q_i(s,a) + \alpha.\left(R(s) + \gamma.\underset{a}{max} Q_i(s',a') - Q_i(s,a)\right) \tag{5}$$

The adjustment of learning parameter α is sensitive since it should decrease as the number of recorded transitions from state *s* under action *a* increases, so that *Q(s,a)* finally converges despite environment stochasticity. The selection of the action to be taken in state*s* is sensitive too: a balance has to be found between exploration and exploitation, so that the agent does not stop exploration after it has found a 'good' solution which may be far from the optimal solution.

Of course, all these algorithms do work well on small state and action spaces but become soon intractable when the spaces get larger. Hence, the idea of introducing hierarchy.

2.3 Hierarchical Markov Decision Process

Decomposing a target MDP into a hierarchy of smaller MDPs is just the adaptation into the MDP framework of the most natural human way of planning actions: a person planning a trip to a distant city does not plan every primitive move (at the muscle level) from its starting point to its destination but plans higher level tasks such as calling a taxi, go to the airport, get in the plane, etc. Two advantages are expected from the definition of a hierarchy in a Markov Decision Process: (i) the hierarchy leads to divide the problem into smaller problems, then the state and action spaces should be reduced, and the problem easier to solve (ii) the hierarchy leads to define higher level actions that will be reusable for a variety of goals. For the trip planning problem, the 'calling a taxi' sub-task does not depend on the airport position or the destination: this is the state space reduction advantage. And the "calling a taxi" sub-task can be reused for many high level trip tasks: this is the re-usability advantage. However, there are various ways to consider hierarchy.

Hierarchy in actions results from the natural understanding of hierarchy in planning, as illustrated by our trip example. A global task is divided into sub-tasks. In (Sutton et al. 1999) sub-tasks are defined as options that consist of a fixed policy applied in a sub-set of states to achieve a given goal. In (Dietterich 2000), sub-goals are identified first, and the sub-tasks are defined to achieve these sub-goals. The sub-tasks are learned thanks to pseudo-rewards associated to the sub-goals. Dietterich has also established that the decomposition of a target Markov decision process (MDP) into a hierarchy of smaller MDPs corresponds to the decomposition of the value function of the target MDP into an additive combination of the value functions of the smaller MDPs: the value function is decomposed into a sum of terms such that no single term depends on the entire state of the MDP, even though the value function as a whole does depend on the entire state of the MDP.

Hierarchy in the learning agents: in (Dayan and Hinton 1993) a feudal system with several levels of agents is defined. An agent from an intermediary level receives orders from an upper level agent and gives orders to sub level agents. Only the lower level agents interact with the environment. The reward an agent receives is linked to the achievement of the task it was given by the upper level agent and not to the achievement of the global goal. The state information an agent uses depends on its level, the higher the level, the coarser the information granularity.

Hierarchy in the environment: in (Parr and Russell 1998) Hierarchical Abstract Machines (HAM) are used, also called "partial policies", since the machine applies a policy that is partially fixed but some states call for a non-deterministic choice (where optimal choice is to be learned) and some other states give control to another machine subroutine. Parr's maze is made of orthogonal hallways littered with obstacles. Higher level HAM is in charge of selecting a hallway, intermediate HAM are in charge of traversing the hallway and lower level HAM are in charge of avoiding obstacles. Whatever the original point of view, the aim of the hierarchy is to reduce the state space to be considered in the learning process. A sub-task is learned taking into account only the relevant parameters. The HAM is also designed to use only a partial description of the environment to determine the next state. The feudal agents are based on a partition of the state space, each agent using only one part of the space. Besides the state space reduction which proves very useful to solve the MDP, the fact to ignore the full context when executing a policy makes it easier to share and re-use. The drawback of any kind of hierarchy is that the learned global policy may be suboptimal since the hierarchy constrains the set of possible policies that can be considered.

Note that the equations 2 and 5 established in section 2.2, must be updated since the sub-tasks have an extended duration (N steps) and then the MDP turns into a Semi-Markov Decision Process. γ is replaced by γ^N giving equations 6 and 7:

$$V^{\pi}(s) = R(s) + \sum_{s',N} P(s',N|s,\pi(s)).\gamma^N.V^{\pi}(s') \tag{6}$$

$$Q_{i+1}(s,a) = Q_i(s,a) + \alpha . \Big(R(s) + \gamma . \underset{a}{max} Q_i(s',a') - Q_i(s,a) \Big) \tag{7}$$

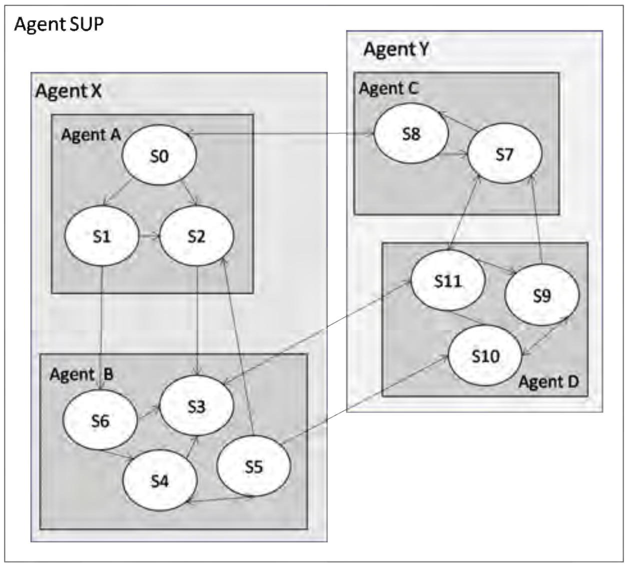

Figure 1: Hierarchy of feudal agents based on states aggregation.

The environment can be in any state of {*S0, S1, ..., S11*}. The first level of agents is made of agents {*A, B, C, D*}. Each one is in charge of a small set of states. The next level is made of agents {X, Y} and the last level is made of only one agent {SUP}. Figure 1 illustrates a feudal hierarchy of agents.

3 DEVS-BASED HIERARCHICAL MARKOV DECISION PROCESS

For an easier understanding understanding, we will first describe the implementation of a MDP with a single agent before introducing the hierarchy of agents.

3.1 MDP with a Unique Agent

3.1.1 The Environment

As part of a MDP, the environment is defined by: (i) a set of possible states (ii) a set of possible actions that modify the current state according to transition probabilities (iii) rewards linked to the current state.

The translation of this description to a DEVS model is obvious since a DEVS model has an internal state, that is modified upon reception of messages (external transitions) and sends messages after a transition: (i) the state of the environment equals the state of the DEVS model (ii) the actions applied by the agent are messages sent by the agent model to the environment model (iii) the reward and state reported to the agent are messages sent by the environment model to the agent model.

As the state transitions are not deterministic, we talk of Finite Probabilistic-DEVS instead of pure DEVS. This means that during a simulation, the next state will be chosen using a random draw and the pre-defined transition probabilities. In addition, DEVS model have a built-in *internal* transition function to simulate time-dependant evolution of the internal state. This kind of evolution can fit into a MDP description as a transition from a state to a another under the 'null' action (consisting in doing nothing).

In the example detailed in section 4, the environment is a simple grid world and is represented as an atomic DEVS model. In the future, in order to highlight the interest of using DEVS for solving this kind of problem, we intend to replace this simple atomic model by a more complex coupled model with temporal interactions between the elements. The stochastic nature of the system would not be synthesized in a single transition table, but dispatched within the elements. Then the hierarchy on agents/environment would be meaningful.

3.1.2 The Agent Atomic Model

A MDP agent applies actions to an environment, collects rewards and uses these data to find out the best way to achieve a goal. Then this agent is the symmetrical model of the environment model: (i) the state of the agent is made of: the *lastState* of the environment, the *lastAction* sent, and a *Q-value* table (Q-value for each state-action pair) (ii) the actions applied by the agent are messages sent by the agent model to the environment model (iii) the reward and state reported to the agent are messages sent by the environment model to the agent model. The learning happens when the reward/state message is received from environment reporting a state *s'*: the agent uses the newly reported state to update the *Q-value[last state, last action]* pair according to equation 7. Then the agent selects the action *a* that maximizes *Q-value[s',a]* and sends it to the environment.

3.2 MDP with a Hierarchy of Agents

The feudal hierarchy is built upon the concept of aggregation of sub-sets of states. The first level (level 0) of agents is defined as a set of agents that interact directly with the environment, sending it primitive actions. Each agent is in charge of a sub-set of the environment states. The upper levels do not communicate with the environment, but with the lower level agents. The agents of level L are processed just like states of a pseudo-environment by the level L+1. It is easy to understand on the grid world example of section 4: each Level 0 agent is in charge of a group of 4 adjacent cells, the agents of the Level 1 are in charge of a group of 4 Level 0 agents, then covering an area of 16 adjacent cells, the agents of the Level 2 are in charge of 4 Level 1 agents, and so on.

3.2.1 The Environment

The environment behavior is unchanged but it now interacts with a set of agents instead of a unique agent. Whatever the source agent is, the environment processes the action-message as in section 3.1 i.e. it updates the current state according to the given action. An easy way to deal with multiple agents would have been to send the new state to all the agents. However, considering the communication overload due to this solution, the environment will use a translation function to find the agent in charge of processing its new current state, it then sends this current state to the agent only. This translation is obvious in the grid world example (direct correspondence between cell and agent) but will require more care when dealing with a complex environment.

3.2.2 The Hierarchy of Agents

We had a requirement to define a generic hierarchical agent, i.e. an agent whose behavior does not depend on the level it belongs to. An agent is connected to a lower level, which may be either a set of agents or the environment itself, and to one upper level agent. From the point of view of an agent of level L, the agent of level L+1 is called the manager and the agents of level L-1 (or states of the environment) are called sub-agents. In our current definition of the feudal agent, the state of an agent is equal to the id of the active sub-agent. Actions go from top to bottom (from high level actions to primitive actions) and states go from bottom to top (from true environment state to coarse high level agent id). The actions received from the manager are called tasks as they set the goal to be achieved by this agent, there is not only 1 goal but as many goals as tasks.

A specific difficulty in the implementation of the feudal MDP compared to classic MDP is that an action *a* taken by agent A_i in state $s \in \{states of agent A_i\}$ can lead to a transition to a state $s' \in \{states of agent A_j\}$ so *s'* is not a state of A_i but has to be considered since it is a destination state and more specifically

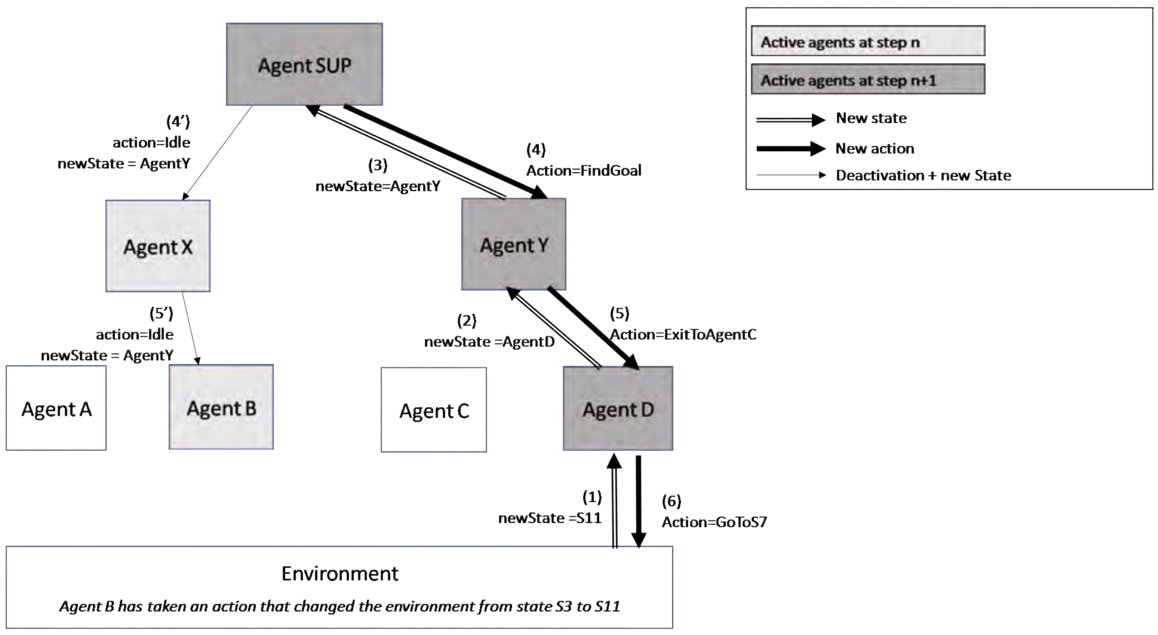

Figure 2: Communication between Agents and Environment in a DEVS hierarchical context.

a goal state associated to a reward for A_i. The complexity of the feudal agent model comes from its interconnections with upper, lower and same level agents, that is why our first application is as simple as a grid world: it made it possible to solve communication and structure issues. The figure 2 represents the state and action flows in the model, numbers indicate the order of the messages.

Let's consider an agent of level L. Its state consists of:

- the list of sub-agents/states it is in charge of,
- the parameters of the Q-value algorithm,
- the list of possible actions it can send to each sub-agent,
- the list of possible tasks it can be assigned to,
- the current task *currentTask*,
- the current state *currentState* = current active sub-agent,
- the last action *currentAction* taken and its start time,
- as many *Q-value* tables as there are tasks.

If the agent interacts with the environment, the possible actions are known and equal to the primitive actions. The tasks it can be assigned to are initialized to *Idle*, *Explore*, *FindGoal*. *Idle* means that the agent is no more active, *Explore* means that the agent should select an action randomly, *FindGoal* means that the goal state belongs to this agent's states and that the agent should select the optimal action to achieve this goal.

For upper level agents, the possible actions and tasks are initialized to *Idle*, *Explore*, *FindGoal*.

As exploration goes on, an agent may find out that its action has led to a state outside the set of states of this agent. On figure 2, *Agent B* discovers that its action has led to state *S11*, belonging to *Agent Y*, then *Agent B* will append the *ExitToAgentY* task to its possible tasks and *Agent X* will append the *ExitToAgentY* action to the possible actions for *Agent B*. Note that the *Explore* action should be selected by the agent of level L as long as the specific *ExitToAgent..* tasks have not been learned by its sub-agent. Finding the appropriate time to start using high level tasks is a well known difficulty of hierarchical learning studied in (Jong et al. 2008). Introducing the high level actions too early may be is useless or even harmful. We did not focus yet on this problem, and chose to get around it by setting time limits to start using *ExitToAgent..* actions.

When it receives a message from a sub-agent with a *newState* and a *goalFound* indicator:

- if the agent was already active (*currentTask* $\neq Idle$ then responsible for the transition), it updates all the *Q-value* tables using the (*currentState, currentAction, newState*) transition (Eq. 7),
- if the *newState* is different from the *currentState*, then the *ExitToNewState* action is added to the list of possible actions for the *currentState* sub-agent and this sub-agent is sent an *Idle* task along with the *newState* so that this sub-agent updates its data according to this last transition.
- if the goal is found then it is reported to the manager with the *goalFound* flag.
- if the *currentTask* is *Idle*, then the agent reports to its manager which will answer with a new task,
- if the *currentTask* is *Explore*, then the agent selects the next action randomly,
- if the *currentTask* is *FindGoal* or *ExitToAgentAi*, then the agent selects the optimal action according to the task to achieve (using the adequate *Q-value* table),

When it receives a *Idle* message from its manager, this message contains a *newState* information:

- a new task *ExitToNewState* is added to the list of possible tasks if it does not already exist and a new *Q-value* table is created,
- a new action *ExitToNewState* is added to the list of possible actions for the *currentState* sub-agent,
- the *Q-value* tables are updated using the (*currentState, currentAction, newState*) transition (Eq. 7),
- the message *Idle* is forwarded downward to this agent's previously active sub-agent.

When it receives a non-*Idle* task message from its manager: (i) if the task is *Explore*, then the agent selects the next action randomly (ii) if the task is *FindGoal* or *ExitToAgentX*, then the agent selects the optimal action according to the task to achieve (using the adequate *Q-value* table).

The next section describes this implementation on the grid world case study.

4 IMPLEMENTATION ON THE GRID WORLD CASE STUDY

The grid world example is a typical MDP use case that can represent the situation of a robot moving inside a room. The room is divided into cells, and at each step, the robot can take one of the four actions: *GoNorth*, *GoEast*, *GoSouth* and *GoWest*. The result of the action is uncertain as the robot goes in the requested direction only 80% of the time, in the other 20% of the time, the action results in a move in one of the 2 directions orthogonal to the requested one. Some cells may be defined as walls or obstacles and therefore are not possible active cells. If the robot cannot move because of a wall, obstacle or border, then it stays in the same cell. When a goal cell is set, the robot is expected to find the shortest path to this goal (considering uncertainty). The feudal hierarchy is built by successive aggregation of adjacent cells as illustrated on the figure 3.

4.1 The Environment

The environment is a 8x8 grid with one *active* cell. It is modelled as a simple atomic model whose state is defined by the id of the *active* cell. The primitive actions are: *GoNorth*, *GoEast*, *GoSouth* and *GoWest*. The resulting active cell is computed according to the rules previously detailed (considering uncertainty and obstacles) and is reported to the agent in charge of this cell: a simple correspondence table is used to identify this agent (Figure 3).

A U-shaped wall is placed to make the problem harder to solve and to highlight the notion of hierarchy on the environment. Indeed, the wall is defined at the cell level and only the agents of level 0 are aware of the wall (unreachable cells), upper level agents do not have access to that level of detail: they discover what states are reachable but do not know whether the path to those states is wide or narrow.

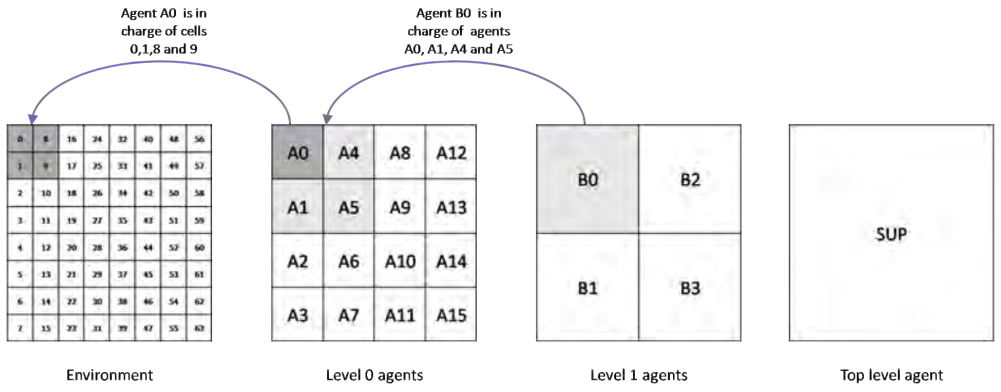

Figure 3: Definition of the hierarchical MDP for the grid world case study.

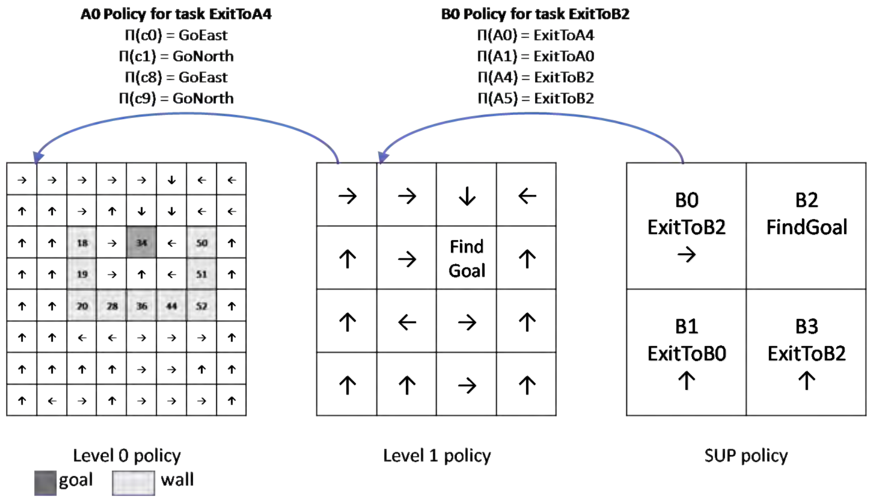

Figure 4: Feudal optimal policy.

4.2 The Agents

Each agent is in charge of 4 sub-agents or cells. With 64 cells in the environment, this leads to 3 levels of agents: Level 0 has 16 agents, Level 1 has 4 and Level 2 has only 1 agent.

The exploration allows each agent to learn all the states it can reach outside the states it is in charge of. Let's consider agent A_5: it is in charge of cells *18, 19, 26* and *27*, its possible actions are *GoNorth*, *GoEast*, *GoSouth*, *GoWest*, the reachable outside states are A_1, A_4, B_1 and B_2. As the agent discovers those reachable states, it starts learning the corresponding tasks $ExitToA_1$, $ExitToA_4$, $ExitToB_1$ and $ExitToB_2$. For the first 2000 steps, as the actions corresponding to these tasks are not learned yet, the agents of level 1 (e.g. agent B_0 for agent A_5) only use the *Explore* action. Then agents of level 1 start using specific actions and learn their own *ExitTo..* tasks. After 4000 steps, the *SUP* agent stops using the *Explore* action and starts using the specific *Exit* actions. The boundaries on the number os steps have been adjusted empirically.

Concerning the implementation of the α parameter, we have used a decreasing α parameter in the form $a/b+k$ where k is the number of transitions for this state-action pair. Concerning the way to balance between exploration and exploitation, we have used the 'optimistic initialization' principle: the Q-table is initialized with a value equal to the maximum expected reward, it makes the agent believe that the space is full of rewards everywhere then encourages the agent to explore all space before settling an optimal policy.

4.3 Simulation Results

The figure 4 represents the feudal optimal policy found after 10000 steps of simulation. The goal has been set in cell 34. Note that the agents have learned all the *Exit* tasks, plus the *FindGoal* task, we have only represented the tasks used to find this goal, i.e. on the right grid, the policy of the SUP agent to fulfill *FindGoal*, on the upper left corner on the middle grid (corresponding to B_0), the policy to fulfil $ExitToB_2$, and on the upper left corner of the left grid (corresponding to A_0), the policy to fulfill $ExitToA_4$. The result is close to the global optimal policy, although one can see some learning mistakes in cells 25 and 23. We think that those mistakes are not due to boundary effect but to bad luck, and might be corrected using a ε-greedy algorithm to balance exploration and exploitation.

The figure 5 represents the mean duration of an episode according to simulation time. We compare the results obtained with the feudal agents and to those obtained with the same environment but with a single learning agent (so-called flat Q-learning). We added the results obtained by feudal agents organized as: 1 SUP agent in charge of 4 B_i agents, each of those in charge of 16 cells, that is one less level of hierarchy. For the 2 levels or 3 levels feudal agents, the initial '0' results correspond to the learning phase.

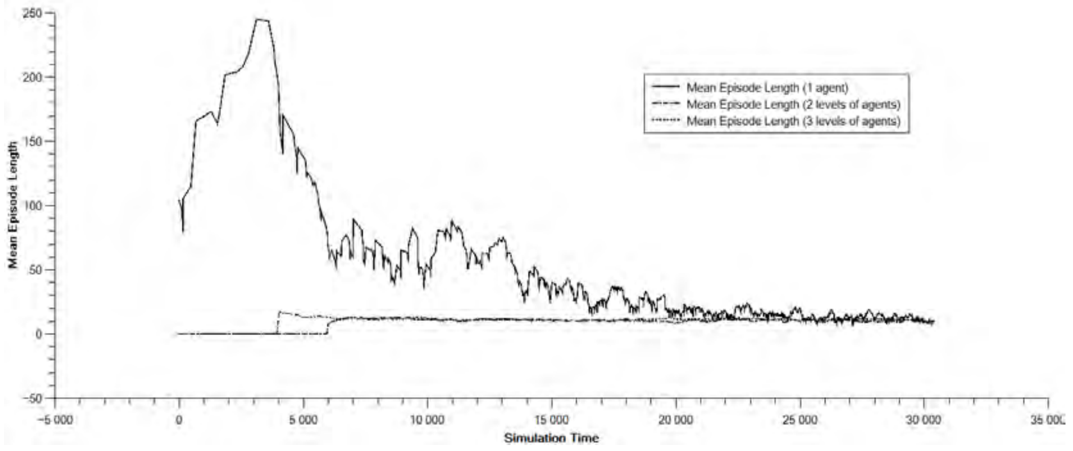

Figure 5: Simulation results for flat Q-learning algorithm with the 8x8 grid example.

We can see that the 3 level feudal system is the fastest to achieve an optimal policy. The 2 level feudal system has comparable results but the learning phase is longer: it is still set empirically, the duration of the phase is increased until it produces good results. This can be explained by the fact that it is harder to find 1 goal among 64 cells than to find 1 exit to a 4-cells grid. The smaller problems can be learned faster. However, we have noticed that the 3 level feudal system is less robust to learning mistakes that the 2 others: a supposed optimal policy is found quickly but then the agent sticks to it with few chances to correct it if it was an effect of bad luck.

4.4 Discussion

Many refinements have been developed to improve the above approaches originated in the 1990's, but this paper does not aim to provide an improved algorithm but to study how the MDP framework can fit within DEVS framework and what advantages could arise from this interaction. We did not use the sub-task approach because its intelligence is entirely in the algorithms used to define the sub-goals and pseudo-rewards and as a consequence using DEVS would not add much interest. On the contrary, the other approaches are based on the exchanges between the agents or HAM where simulation can prove to be more useful. We finally decided to focus on the Feudal MDP framework because of the genericity of the definition of the feudal agents: the same model, with adequate parameters, can be used to describe any agent from any level, taking full advantage of DEVS built-in modular and hierarchical modeling. The implementation of a generic feudal agent in DEVS highlights an interesting aspect of the Feudal MDP: as the lower level agent commanding to the environment uses the same model as an upper level agent commanding to lower level agents, it appears that the set of lower level agents should be processed just as the set of states of the environment. Then the hierarchy on agents can also be seen as a hierarchy on the environment. In other words: the agents of level N actually *are* the environment for the agents of level N+1. With the notations of figure 2, the agent X interacts with an environment that can be in state A or B. The advantages of DEVS in comparison with traditional approaches when dealing with hierarchical MDPs are: (i) no traditional approaches are implemented with a generic view while, due to its modular and hierarchical aspects, the DEVS formalism allows us to propose a family of hierarchical MDPs models (ii) the setting of MDPs empirical parameters is facilitated by the use of DEVS atomic models organized as presented in figure 2.

5 CONCLUSION AND PERSPECTIVE

The paper has presented an approach based on the DEVS formalism that supports modeling and simulation of planning and decision problems with hierarchical MDP. We have pointed out that DEVS brings solution to develop generic concepts for defining hierarchies of agents within the Framework of MDPs and reinforcement learning. Furthermore, we showed the effectiveness of the proposed approach with a typical case study of MDPs. This example provides a basis for further development necessary to exploit the DEVS-oriented hierarchical MDPs to solve complex optimization problems requiring simulations in various application domains. Some of the dimensions requiring such development are: (i) Firmly establishing the generic nature of the approach by an in-depth formalization of the agents and environments within the DEVS formalism. This will require exploiting, and further developing, the extensions of DEVS for Markov M&S which will exploit DEVS's ability to represent complex stochastic environments (Zeigler et al. 2017) (ii) Based on this formalization, exploit, and further develop, M&S tools and environments such as DEVSimPy (Capocchi et al. 2011) and MS4Me (Zeigler and Sarjoughian 2013) to help users write MDP models and simulate them with relative ease and confidence in their computational and representational validity (iii) Addressing some of the theoretical issues underlying the hierarchical architecture developed in this paper. For example, probing the nature of the hierarchical abstractions of the grid world application, characterizing their fundamental structures, and test their generality and applicability in a wide variety of domains. Also extending abstraction to include time granularity (Santucci et al. 2016, Zeigler 2016) (iv) Finally, there are many areas in which the learning aspects of the model can be extended to improve the proposed approach and enable us to compare the results of the simulation with offline calculation of optimal policies. An extensive experimentation and more in-deep discussion will be performed in order to fully evaluate the promising DEVS-based approach for hierarchical MDPs.

REFERENCES

Bellman, R. 1957. "A Markovian Decision Process". *Indiana Univ. Math. J.* 6:679–684.

Capocchi, L., J. F. Santucci, B. Poggi, and C. Nicolai. 2011, June. "DEVSimPy: A Collaborative Python Software for Modeling and Simulation of DEVS Systems". In *Proc. of 20th IEEE International Workshops on Enabling Technologies*, 170–175.

Christopher J. C. H. Watkins, P. D. 1992. *Q-Learning*, 279–292. Springer.

Dayan, P., and G. E. Hinton. 1993. "Feudal Reinforcement Learning". In *Advances in neural information processing systems*, 271–271: Morgan Kaufmann Publishers.

Dietterich, T. G. 2000. "Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition". 13:227–303.

Howard, R. A. 1960. *Dynamic Programming and Markov Processes*. MIT Press, Cambridge, Massachussetts.

Hwang, M. H., and B. P. Zeigler. 2009, July. "Reachability Graph of Finite and Deterministic DEVS Networks". *IEEE Transactions on Automation Science and Engineering* 6 (3): 468–478.

Jong, N. K., T. Hester, and P. Stone. 2008. “The Utility of Temporal Abstraction in Reinforcement Learning”. In *Proc. of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 1*, 299–306: International Foundation for Autonomous Agents and Multiagent Systems.

Nutaro, J. J., and B. P. Zeigler. 2015. “Towards a Probabilistic Interpretation of Validity for Simulation Models”. In *Proc. of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, DEVS ’15, 197–204. San Diego, CA, USA: Society for Computer Simulation International.

Parr, R., and S. Russell. 1998. “Reinforcement Learning with Hierarchies of Machines”. In *Proc. of the 1997 Conference on Advances in Neural Information Processing Systems 10*, NIPS ’97, 1043–1049. Cambridge, MA, USA: MIT Press.

Santucci, J.-F., L. Capocchi, and B. P. Zeigler. 2016. “System Entity Structure Extension to Integrate Abstraction Hierarchies and Time Granularity into DEVS Modeling and Simulation”. *SIMULATION* 92 (8): 747–769.

Seo, C., B. P. Zeigler, D. Kim, and K. Duncan. 2015. “Integrating Web-based Simulation on IT Systems with Finite Probabilistic DEVS”. In *Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, DEVS ’15, 173–180. San Diego, CA, USA: Society for Computer Simulation International.

Sutton, R. S., and A. G. Barto. 1998. *Reinforcement Learning : An Introduction*. MIT Press.

Sutton, R. S., D. Precup, and S. Singh. 1999. “Between MDPs and Semi-MDPs: A Framework for Temporal Abstraction in Reinforcement Learning”. 112 (1): 181–211.

Van Mierlo, S., S. Mustafiz, B. Barocca, Y. Van Tendeloo, and H. Vangheluwe. 2015. “Explicit Modelling of a Parallel DEVS Experimentation Environment”. In *Proc. of the Symposium on Theory of Modeling & Simulation*, DEVS ’15, 860–867. San Diego, CA, USA: Society for Computer Simulation International.

Van Tendeloo, Y. 2014. “Activity-Aware DEVS Simulation”. Master’s thesis, University of Antwerp.

Zeigler, B. P. 1976. *Theory of Modeling and Simulation*. Academic Press.

Zeigler, B. P. 2016. “The Role of Approximate Morphisms in Multiresolution Modeling: Can we Relax the Strict Lumpability Requirements?”. *The Journal of Defense Modeling and Simulation* 0 (0).

Zeigler, B. P., J. J. Nutaro, and C. Seo. 2017. “Combining DEVS and Model-Checking: Concepts and Tools for Integrating Simulation and Analysis”. In *Int. J. Process Modeling and Simulation*, Volume 12: I3M 2014: Special Issue on: New Advances in Simulation and Process Modelling: Integrating New Technologies and Methodologies to Enlarge Simulation Capabilities.

Zeigler, B. P., H. Praehofer, and T. G. Kim. 2000. *Theory of Modeling and Simulation, Second Edition*. Academic Press.

Zeigler, B. P., and H. S. Sarjoughian. 2013. *Guide to Modeling and Simulation of Systems of Systems*. Springer-verlag London.

AUTHOR BIOGRAPHIES

CELINE KESSLER is a PhD student in Computer Science at the University of Corsica (France). His email address is kessler.celine@orange.fr.

LAURENT CAPOCCHI is Associate Professor in Computer Science at the University of Corsica (France). His email address is capocchi@univ-corse.fr.

JEAN-FRANCOIS SANTUCCI is Full Professor in Computer Science at the University of Corsica (France). His email address is santucci@univ-corse.fr.

BERNARD ZEIGLER is Professor Emeritus of Electrical and Computer Engineering at the University of Arizona. His email address is zeigler@rtsync.com.