A Unified Conceptual Framework Integrating UML and RL for Efficient Reconfiguration Design

Amen Ben Hadj Ali and Samir Ben Ahmed
Faculté des Sciences de Tunis, Tunis El Manar University Tunis, Tunisia

Keywords: RCS Design, UML, OCL, Reinforcement Learning, Reconfiguration Models, Q-Learning, Exploration, Reconfiguration Design.

Abstract: The problem of early exploration of various design choices to anticipate potential runtime changes at design time for complex and highly-dynamic Reconfigurable Control Systems (RCS), is still a real challenge for designers. This paper proposes a novel conceptual framework that integrates the benefits of UML-based modeling with Reinforcement Learning (RL) to overcome this difficulty. Our proposal exploits UML diagrams enriched with OCL constraints to describe the reconfiguration controller structure and dynamics using predefined reconfiguration knowledge. On the other hand, the reconfiguration controller is designed as a RL agent (Reinforcement Learning Reconfiguration Agent or RLRA) able to improve its knowledge through online exploration while running a Q-Learning algorithm. The design process we propose starts with an abstract UML-based specification of RCS. Then, a RL-based framework in Python language will be generated from UML/OCL models by applying a generation algorithm. Finally, the resulting framework will be run to allow the RLRA learning optimized reconfiguration policies and eventually improve first design specifications with learning feedback. The learning phase supports both offline and online learning and is based on a Q-Learning algorithm.

1 INTRODUCTION

Manufacturing control systems continue to evolve steadily in the age of the fourth industrial revolution (Industry 4.0) (PI, 2016), where a series of requirements such as autonomy and reconfigurability (Elmaraghy et al., 2021), are imposed on future controllers and control approaches to fit the next generation of production systems based on the concept of Cyber-Physical System (CPS) (Monostori et al. 2016). Reconfiguration allows the control system to switch from one configuration to another, improving the system's efficiency concerning unexpected changes such as environmental disturbances, and unpredictable events, like failures (Koren et al., 1999).

In this research work, with reconfiguration controller, we refer to the software module that, taking as input a representation (configuration) of the controlled system managed by the controller, selects a discrete high-level sequence of reconfiguration actions (a reconfiguration policy) leading to a safe configuration of the whole system.

Within the field of Reconfigurable Control Systems (RCS), designers are faced with two major difficulties: (1) making reconfiguration knowledge explicit through appropriate conceptual models, which is a crucial step for managing reconfiguration requirements (Lepuschitz, 2018). In this context, the additional value of UML-based (Unified Modeling Language) (OMG, 2017) models and model-driven engineering (MDE) is widely recognized, due to the high-level abstraction and the automation of analysis and full code generation that they can provide (Vyatkin, 2013). (2) Moreover, given the challenge of fast-changing dynamic manufacturing environments, it is hardly possible to fully explore all control software configurations. Thus, the use of Machine Learning (ML) techniques is a quite natural and appealing approach. Specifically, Reinforcement Learning (RL) (Sutton and Barto, 2018) is a subfield of machine learning that offers algorithms for learning to control a system by interacting with it and observing feedback (reward). Using this feedback is an efficient means to evaluate how well a controller is performing. This ability is important in the RCS context since it is difficult to write a deterministic

Ben Hadj Ali, A. and Ben Ahmed, S.
A Unified Conceptual Framework Integrating UML and RL for Efficient Reconfiguration Design.
DOI: 10.5220/0012468600003636

In *Proceedings of the 16th International Conference on Agents and Artificial Intelligence (ICAART 2024) - Volume 2*, pages 604-613
ISBN: 978-989-758-680-4; ISSN: 2184-433X

control program that can anticipate all unexpected changes and thus implement a high-quality controller, but it is relatively easy to specify a feedback signal that indicates the best reconfiguration actions to perform. Another advantage of RL is the possibility of extending the design space based on learning feedback.

A possible way to overcome the identified difficulties is by combining the benefits of the UML-based design approach with reinforcement learning capabilities. As far as we know, the present work is original since it represents a first attempt to combine UML and model-driven design principles with RL benefits into a novel unified conceptual framework to address the challenges of reconfiguration knowledge modeling and exploration. The value of such a design approach is that it enables partial reconfiguration knowledge (prior knowledge) modeling and analysis using UML-based models. The rest of the knowledge (e.g., which sequence of reconfiguration actions to select and the order in which to select them), is learned by the run-time RL model.

Clearly, the proposed reconfiguration controller, referred to as RLRA (Reinforcement Learning Reconfiguration Agent) is a decision maker that has to be designed in order to output an "efficient reconfiguration policy" in every situation. In our work, the notion of efficiency is typically obtained as an emerging property coming from the ensemble of different design objectives, namely, safety and optimal reconfiguration time. The proposed RL-based framework for RCS design is a Python environment that is able to run Python 3.6, or above, for the execution of reconfiguration control functions developed in Python language. The idea behind using Python language is to enable the latest advances in machine learning to integrate at the control level with existing industrial standards like the IEC 61499 (IEC, 2005).

In this paper, we explore the use of standard UML (OMG, 2017) models, enriched with OCL (Object Constraint Language) (OMG, 2014) constraints, for the specification of RCS. We shall rely on UML as a modeling language, since many existing tools (such as USE (Gogolla, 2007)) provide a wide variety of analysis capabilities for UML models, including model validation, instance generation, or invariant checking. In addition, high-level UML and OCL models are used in order to encode design objectives (such as safety) during the early design stages into lightweight models with lower development costs than the full implementation of the control system.

Furthermore, for safety-critical systems, including prior knowledge in the exploration process of the RL agent is often used as a solution to avoid risky situations during the exploration (Garcïa, 2015). In this paper, UML/OCL models give an abstract representation of prior knowledge and thus allow to focus exploration of the RLRA's state space, reducing risks as well as the random phase that a RL agent must endure while learning about a new environment. In addition, since a RL agent is effectively operating in a reduced state space, learning will also be faster.

The contributions of this work are threefold: (1) the abstract and formal modeling of reconfiguration knowledge using UML/OCL models and RL mathematical fundamentals thus allowing both early analysis and fast exploration of various design choices for the lower-level implementation. A set of rules is defined in order to allow the generation of RL models from UML/OCL models. (2) In addition, our design process handles learning since the reconfiguration controller is designed as a RL agent (RLRA) and therefore it supports exploitation as well as exploration. In particular, the RLRA implements a Q-Learning algorithm and supports optimized exploration. (3) Furthermore, to enforce generalization and give more flexibility to the design approach, we define a metamodel that abstracts the proposed framework knowledge allowing the designer to integrate new concepts, algorithms, and modeling techniques.

The remainder of the paper is structured as follows. Section 2 presents the fundamentals of the used concepts. In particular, it presents the core of the applied machine learning technique for reconfiguration control design. Section 3 presents a brief review of related work. Section 4 gives an overview of the proposed design process. Section 5 focuses on presenting the main contributions of the proposal. Section 6 presents a case study that will be used to show the applicability of the proposed conceptual framework. Finally, the contributions of the paper and the further challenges are summarized in Section 7.

2 PRELIMINARIES

To provide a comprehensive guide to understanding the remainder of this paper, this section introduces some basic theoretical notions of RL.

The central idea of RL is that the learning agent learns over time by trying the different available actions in different situations and evaluates the outcome of each action, both in terms of immediate

reward (i.e., the action's immediate effect on the environment) and long-term cumulative reward (i.e., the contribution to the learning agent's overall objectives). The basic mathematical model of RL is Markov Decision Processes (MDP) (Bellman, 1957). Fundamentally, an MDP aims to solve a sequential decision-making (control) problem in stochastic environments where the control actions can influence the evolution of the system's state. An MDP is defined as a five-tuple (S, A, R, P, γ) as follows: S is the state space, and A is the action space.

P: S × A × S → [0, 1] gives the state transition probability. $P(s'|s, a)$, specifies the probability of transition to s' by taking action a in state s. R: $S \times A \rightarrow \mathbb{R}$ is the reward function dictating the reward an agent receives by taking action a ∈ A in state s ∈ S, and $\gamma \in [0, 1]$ is the discount factor (Sutton and Barto, 2018).

It is essential to notice that the environment of our RL agent is the reconfigurable system (the controlled system), in contrast to classical RL frameworks, where the environment is represented by the uncontrolled system.

Two main learning strategies are available, exploration and exploitation. Making sure that the agents explore the environment sufficiently is a common challenge for RL algorithms known as the exploration-exploitation dilemma. The ε-greedy policy is a well-known method to address the exploration-exploitation trade-off while training the RL agent. This method, can balance exploration and exploitation and make sure we are never ruling out one or the other. Our exploration strategy uses constraints defined in UML models to give structure to the reconfiguration design space and thereby leverage additional information to guide exploration. Each configuration is considered as a valid constraints' combination defined on reconfigurable active parts (elements) of the control system.

3 RELATED WORK

Solutions and research efforts already exist tackling RCS design using different approaches. In particular, for classical manufacturing control systems, several works (Thramboulidis and Frey, 2011) (Ben Hadj Ali et al., 2012) (Fay et al. 2015) (Ouselati et al., 2016) adapt UML and its extensions (such as SysML and MARTE) for designing and modeling the control logic (Vyatkin, 2013). These works often aim to reduce control software complexity by raising the abstraction level while ensuring automatic generation of PLC (Programmable Logic Controller) standard-compliant code (IEC 61131 and IEC 61499) (Vyatkin, 2013). In addition, more recent research works, such as (Thramboulidis and Christoulakis, 2016) (Schneider et al., 2019) (Müller et al., 2023) (Bazydło, 2023) (Parant, 2023), introduce UML-based solutions to model and design the control part of manufacturing systems compliant with I4.0 and that are considered CPSs in which multiple concurrent software behaviors govern industrial components running on embedded controllers.

As a semi-formal language, UML provides high relevance to handling the semantic gap between system design and the actual features of the control application. However, UML-based design approaches suffer from a lack of precise semantics. For this reason, several researchers propose to combine UML diagrams with formal languages for the model-based design of RCS. The formalization of control model elements is performed using formal languages (such as Petri nets, Timed Automata, etc.) to describe specific reconfiguration requirements and thus guarantee the consistency and the correctness of the specification and code generation by using verification techniques (such as model checking) (Vyatkin 2013) (Mohamed et al., 2021). These approaches allow for verifying that the system behaves correctly for all possible input scenarios by giving a precise description of the possible system behavior. However, most of them are based on an automated transformation from a system description with informally defined semantics and lack learning capabilities. In addition, the reviewed works have in common the exploitation of UML-based metamodels and models to deal with reconfiguration and reconfigurable systems modeling (Mohamed et al., 2021) and therefore allow the automation of several design steps such as validation/verification and code generation. However, they are often static since reconfiguration knowledge that is not anticipated during design time is handled statically by revising (modifying) existing models offline (Ben Hadj Ali and Ben Ahmed, 2023).

Furthermore, several works have proposed many RL agents to model efficient reconfiguration controllers that can learn optimized reconfiguration policies (plans). The optimization goal is therefore formulated using the reward (objective) function of the RL agent (Wuest et al. 2016) (Kuhnle et al., 2020) (Shengluo and Zhigang 2022) (Saputri, and Lee, 2020). Despite learning capabilities, the dynamicity of the reconfiguration space is only partially implemented within these approaches because they mainly focus on exploitation with random exploration. Therefore, an effective conceptual frame-

work, that bridges the gap between UML and RL modeling, is still needed. In the following sections, we will detail our proposal to deal with the identified literature drawbacks.

4 DESIGN PROCESS

To highlight the contributions of this work, we propose in the following section a generic process for RCS design represented as a UML activity diagram (see Figure 1):

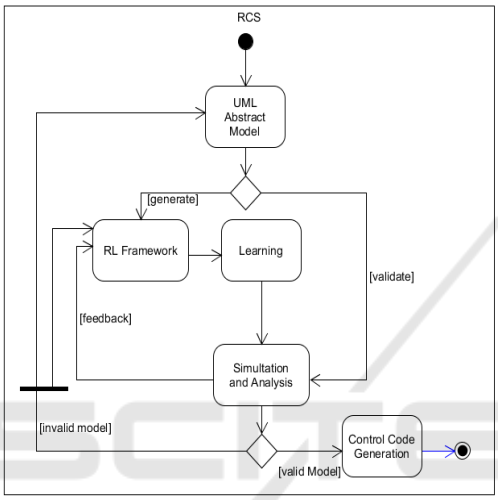

Figure 1: RCS Design process.

a) ***RCS Abstract Specification:*** The representation of reconfiguration knowledge (structures, behaviors, constraints, etc.) through abstract specifications using UML/OCL models.

b) ***RCS Modeling:*** The output of this step is an abstract model expressed using a formal or semi-formal language. In contrast to the majority of reviewed approaches, where the design aim is validation/verification, in this work the design purpose is learning, therefore the obtained model is expressed using RL concepts. Thus, this step aims to generate the code of a RL-based framework which will be used later for the learning step.

c) ***Learning:*** In this proposal, the learning step allows the RLRA to learn optimized reconfiguration policies using a hybrid strategy supporting both exploitation and exploration. In particular, online exploration allows for improving RCS models (UML-based and RL-based) with operation-time learned knowledge.

d) ***Simulation:*** The obtained RCS model can be analyzed using simulation. Therefore, the initial models can be improved or validated and then stored within the Knowledge Base. In our work, the validation of the proposed models is undertaken using the USE (UML-based Specification Environment) tool (USE, 2021).

e) ***Code Generation:*** In this step, the high-level control code is generated using a specific language (such as Java, C/C++, Python, etc.).

The main contributions of this paper focus on the first three steps of the design process and consequently, they will be detailed in the remainder (i.e., Abstract Specification, RL models generation, and Learning).

5 CONCEPTUAL FRAMEWORK

As stated previously, the design process followed in this paper progresses from the state-of-the-art RCS design process by enhancing the reconfiguration controller (RLRA) with learning capabilities. Indeed, our process handles a learning phase that gives the reconfiguration controller the ability to learn optimized reconfiguration rules and also to improve its knowledge through online exploration. Our proposed conceptual framework is composed of five steps (see Fig. 1). The *Simulation* and *Code Generation* steps are out of the scope of this paper. The first three steps will be detailed in the following sections.

5.1 Abstract Specification Using UML

In this paper, the high-level control part of RCS is specified as UML diagrams. Structures are described using class diagrams, whereas behaviors are represented using state diagrams. Such a choice is often sufficient for specifying the dynamics of the control level since UML state machines represent a common tool used to specify the behavior of complex and real-time systems (Harel and Politi, 1998).

Therefore, the first step in our conceptual framework for the design of RCS corresponds to the elaboration of the RCS UML-based specifications according to the metamodel of Fig. 3. The proposed metamodel is structured into three packages corresponding to the respective models of, the state space (CM, i.e., *ConfigurationModel* package), the action space (RM, i.e., *ReconfigurationModel* package) and the RL reconfiguration agent (RLRAM, i.e., *RLRAModel* package).

The presented conceptual elements and their interactions are defined as follows: dynamic reconfigurations are driven by the Reconfiguration Agent (RLRA) which monitors the reconfigurable controlled system and allows to reconfigure its actual

configuration when it detects an internal or external reconfiguration requirement (trigger). In the remainder of this subsection, we will present the basic concepts and operations defined within the knowledge metamodel using UML diagrams and OCL constraints. The central concept in the proposed metamodel is represented by class *ReconfigurableElement.* This class allows for describing any changeable and observable part of the control system, namely the controller (Class *RLRA*), any controlled element (class *ControlledElement*), any reconfiguration constraint (class *Reconfig-Constraint*) and any configuration of the RCS (class Configuration). Each object of this class is reconfigurable since its structure or behavior can change over time. In the following subsections, we focus on presenting the main concepts and operations proposed to describe the controller structure and behavior.

5.1.1 RLRA Structure

The RLRA structure is composed of three components as shown in Figure 2: The *Monitor*, the *Learner* and the *Executor*.

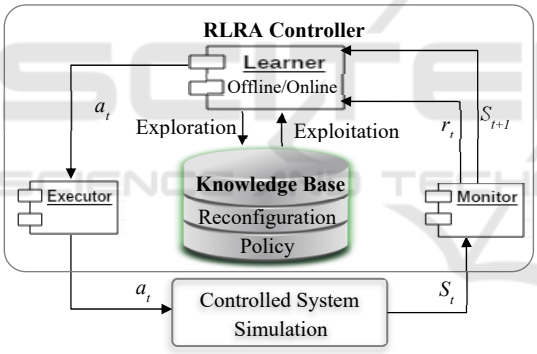

Figure 2: RLRA structure metamodel.

At one moment in time, the RLRA can have only one active configuration (an object of class Configuration). A configuration is a set of objects of the class ControlledElement. For each active object (isActive is True) of the ControlledElement class (CE) (part of the current Configuration of the RLRA), we define a set of constraints and reconfiguration points (RP).

A RP represents a change value Δ_{RP} of the current value of the CE (Δ_{val})that makes the currently observed constraints (applied on the CE) not satisfied.

The operation update for a controlled element and checkRP for a constraint are described in OCL as follows:

```
context ControlledElement::
update(delta: ValueType, ts : Time):
post: value = value@pre + delta
```

```
context ControlledElement::
checkRP(ts : Time): Boolean
  pre: self.isActive
  post:self.constraints->
       exists(c.isValid=#true and
       c.timestamp=ts and self.rps->
       exists(rp.isValid=#true and
       rp = not c and rp.timestamp=ts))
```

In addition, RLRA implements a query operation (checkRP()) that decides if the agent has not yet detected a reconfiguration trigger (internal or external). This operation is described in OCL:

```
context RLRA::
checkRP(ts : Time): Boolean
  pre: self.safetyLevel=#isSafe
  body: self.triggers->
        exists(t|t.timeStamp=#ts and
        t.check())
```

When, a reconfiguration *Trigger* is detected, the RLRA can start *reconfiguring* (the executed action () is *reconfigure*). When the *current context* of the RLRA is known, (predefined =true) then the *Executor* sub-component starts the reconfiguration of the current configuration until it reaches a safe output configuration. This configuration is terminal (isTerminal=true). However, when the *context* is unknown, the *Learning* phase (realized by the *Learner* sub-component) is launched to learn a new (not yet stored in the Knowledge Base) safe output configuration responding to the reconfiguration trigger.

Class Context aims to save the history of the learning phase represented by the knowledge of the different situations the RLRA can have, i.e., the current configuration of the agent when a reconfiguration trigger is raised. The eventual solutions for this context correspond to the reconfiguration policies learned by the agent during the offline or online learning phase. If a given context has at least one solution then it is considered as a known context (predefined=true) and therefore the agent can execute the reconfigure() operation, otherwise it must apply the learn() operation.

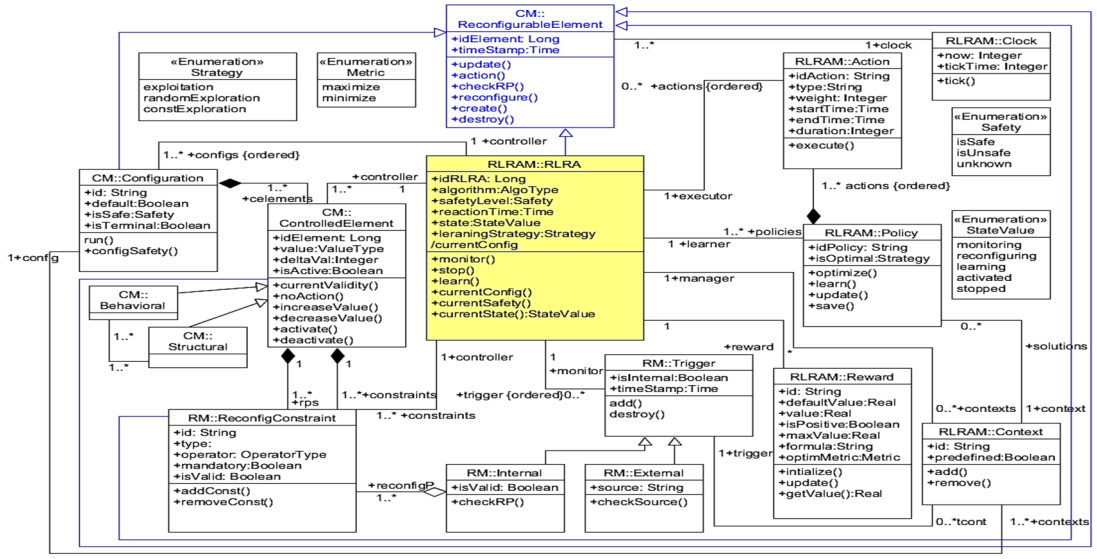

Figure 3: Reconfiguration knowledge metamodel.

5.1.2 RLRA Behavior

The reconfiguration controller is not guided by any predefined plans, and hence it has to decide to take an action at each time. The default state of the RLRA (see Fig. 4) is "monitoring" (the executed action() is *monitor*). In this state, the *Monitor* subcomponent observes the *safetyLevel* of the RLRA and thus checks the reconfiguration points of each observable and reconfigurable element.

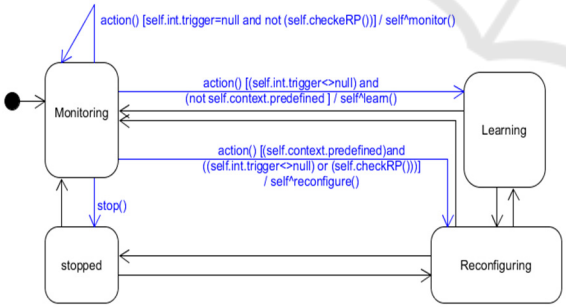

Figure 4: Excerpt of the State Machine for the RLRA behavior.

Furthermore, the controller is associated with a Clock object that models Time. On every tick (whose resolution is defined by the user, for example, in our simulations we have used 0.5 seconds), it invokes operation action() on all active and reconfigurable elements of the control system. For example, as shown in Fig. 4, if the RLRA is monitoring and there is no reconfiguration trigger (internal or external event), the RLRA keeps monitoring; if the reconfiguration point for the agent is reached (checkRP() returns true) or an external trigger is detected, the agent starts reconfiguring or learning; and if both steps generate no safe response to the current reconfiguration request, then the agent and the controlled system are stopped in order to avoid dangerous behaviors.

Whenever a RP is reached an instance of class Trigger is created. Therefore, the behavior of the Monitor can be specified in OCL as the following code fragment shows.

```
context RLRA monitor (ts: Time)body:
begin
 def: iConf : Set(ControlledElement)
       =self.currentConfig()
 def: trig:=self.triggers->
       (t|t.timestamp=ts and t.check())
       t:=new Trigger, ctx:=new Context
  if trig->isEmpty() then
   for ce in iConf do
    for econst in ce.constraints
     for rp in ce.rps
      if econst->intersection(not rp)->
       notEmpty and  rp.isValid and
       rp.timestamp = econst.timestamp
      then
       t.timestamp=ts
        insert(t) into Trigger
       ctx=self.currentConfig()
       ctx.trigger=t
       insert(ctx)into Context
      end
     end
   end
end
```

5.2 RL Models Generation

5.2.1 State Space

In this work, the state space for the RLRA is the set of all possible *configurations* the agent could inhabit. In the proposed RL model, a state (denoted *s*) is an observable combination of both *Structural* and *Behavioral* controlled elements (see Figure 3) corresponding to a possible configuration and represented as a vector: $s=((v_{p1}, v_{p2}, \ldots, v_{pn}), v_{mode})$, where the vector $(v_{p1}, v_{p2}, \ldots, v_{pn})$ represents the respective values of $p_1, \ldots, p_n$ that correspond to the active structural controlled elements composing the current configuration. The vector v_{mode} gives the values of behavioral controlled elements (operations) that describe the operational mode of the current configuration. The set *C* of all possible configurations of the RLRA is separated into pairwise disjoint subsets: the set of safe configurations C_{safe} and C_{unsafe} such as $C \triangleq C_{safe} \cup C_{unsafe}$ and C_{safe} is the set of configurations that satisfy all the system constraints and which are validated through simulation. C_{unsafe} is a possible configuration that can result from the learning process (exploration) and which is not validated or it violates at least one constraint. Furthermore, the default configuration is defined by the designer.

5.2.2 Action Space

The action considered by the RL agent is the reconfiguration control. At each step t, the agent can perform an action a_t from a discrete action space to a given reconfigurable element (property) of the current configuration. Each defined action corresponds to the execution of a given operation of the *ControlledElement* object. As shown in Table 1, action 0 means that the RLRA will maintain its current configuration. Actions 1 and 3 represent positive (i.e., increase the value) and negative (i.e., decrease the value) changes that cannot cause reconfiguration. Action 2 represents the fact that the new value of the property $S_{t+1}(p)$ (after applying the change Δ_{val}) triggers a reconfiguration and thus the controller has to execute an adequate sequence of reconfiguration actions (a policy) and to bring the system to a safe destination configuration in order to respond to this trigger. To prevent the agent from reaching negative values, we clip the minimum value for all structural reconfigurable elements to 0. We also limit the maximum value to some fixed value to avoid dangerously high values.

The transition model $T(s_{t+1}|s_t, a_t)$ for the obtained MDP is deterministic. For action a_t, we map it to different values of changes Δ_{val} as shown in Table 1. For each detected change, we update the property's value using the following rule:

$$S_{t+1}(p) = S_t(p) + \Delta_{val}\, S_t(p).$$

Table 1: Mapping changes to operations and actions.

Action a_t	0	1	2	3
operation	noAction	update/ increase	reconfigure	update/ decrease
Δ_{val}	*0*	>0 and $< \Delta_{RP}$	Δ_{RP}	*<0 and* $\lvert\Delta_{val}\rvert < \Delta_{RP}$

As reconfigurable properties have discrete values we consider Δ_{val}= 1 as the smallest value of observable change.

5.2.3 Reward Function

To achieve our safe reconfiguration goal, we consider three components when designing the reward function R(s, a), each with a specific objective.

a) R_{safety} encodes the objective that the controller should apply valid actions (i.e., actions that maintain the set of reconfiguration constraints satisfied) when updating the value or reconfiguring a controlled element. If the controller applies an invalid action, it receives a penalty of −10.

b) $R_{trigger}$ is the termination reward the controller receives when it reaches a safe configuration that responds to the reconfiguration trigger. If the controller reaches the targeted configuration, it is given a reward of +10. Otherwise, it receives a penalty of −10.

c) R_{time} is the step reward that encourages the controller to minimize the number of reconfiguration actions required to reach the destination. For every state along the controller's path (except the terminal state), the controller receives a penalty of −1. The reward function for the RLRA is given by the following equation:

$$R_{RLRA}(s, a) = R_{safety} + R_{trigger} + R_{time}.$$

5.2.4 Learning Algorithm

The algorithm used to approach the problem is Q-learning (Watkins and Dayan, 1992) which incrementally estimates the action value function Q(s, a). The reconfiguration agent tries to learn the best policy regarding a specific cost function (Q-value function) for possible actions that can be performed. An agent learns a mapping of states to actions based on a learned policy. Q-learning updates the state-action value mapping (i.e., updates its weights) at every time step as follows:

$$Q^*(s,a) \leftarrow Q^*(s_t,a_t) + \alpha[r + \gamma max_a Q^*(s_{t+1},a_{t+1}) - Q^*(s_t,a_t).$$

In order to make sure the action value function converges, we allow the agent to explore by applying the ε-greedy strategy in the offline learning phase. Specifically, the strategy chooses a random action with probability ε (ε ϵ [0,1]). Otherwise, it chooses the greedy action which achieves the maximum action value function for the current state. Since most states are not explored at the beginning, there is much higher uncertainty to start with. Therefore, we set a relatively high ε at initialization and decay it over time. Thus, the agent is encouraged to explore the environment early in the learning process and take full advantage of what it has learnt as the policy converges. Finally, the policy π generated by Q-learning can be expressed as follows using the updated action value function Q:

$$\pi(s) = arg \max_{a \in A} Q(s,a).$$

The learning step is based on the Q-Learning algorithm such that at each time step, the RLRA must decide which combination $s=((v_{p1}, v_{p2}, \ldots, v_{pn}), v_{mode})$ is selected.

The set of proposed generation rules is implemented using the generation algorithm shown in Fig.5

Algorithm. Generation of the RL framework.

```
1. Input: UML/OCL models
2. Output: MDP_RCS= < C, A, r, π, γ >, S_Space C,
   A_Space A, double [] r, double π, γ
3. Set<Class> RCS_structures
4. Set<Region> RCS_SM //the set of regions
   describing the behavior of the RCS
5.  RCS_modes, RCS_regions //the set of SM
   describing the modes of the RCS
6. Set<ReconfigurableElement> VS, VM //the
   set of properties describing the RCS
7. function Generate_StateSpace(Set<ModeSM>
   RCS_modes):S_Space
8. for each class ci in RCS_structures do //
   define  one vector for each class
9. add vector VSi to VS // an ordered set of
      reconfigurable structural elements of
      the RCS
10.   for each value pj in val(pj) do // pj
      is an attribute of the class ci
11.      add a value vpj to VSi // add a
         value for each structural property
12.   end for
13. end for
14. for each ModeSMi in RCS_modes do //
    define one vector for each Mode_SM
15.   add vector VMi to VM // an ordered set
      of reconfigurable behavioral elements
      of the mode SMi
16.   for each opj in VMi  do
17.    add a value vopj // add a value for
       each behavioral property
18.   end for
19. end for
20. for each Combi=(VSi,VMi) //a combination
    of structural and behavioral reconfigu-
    rable properties
21.    add a state Si to C// a configura-
       tion within the state space
22. end for
23.  return C
24. end function
25. function Gener-
  ate_ActionSpace(Set<Region>
  RCS_SM):A_Space
26. for each RE in RCS_model
27.  for i in [0,3]
28.    add ai to A // add a reconfiguration
       action to the action space
29.  for each Transition Ti in RCS_SM
30.    add the preconditions to r // speci-
       fy the reward value for the recon-
       figuration action
31.    add the postconditions to r // spec-
       ify the reward value for the recon-
       figuration action
32.  end for
33.  end for
34. end for
35. return A
36. end function
```

Figure 5: Generation Rules.

In order to assist designers while generating the RL framework, we implemented the set of generation rules within a prototype tool (in Python) that aims to automatize the design process.

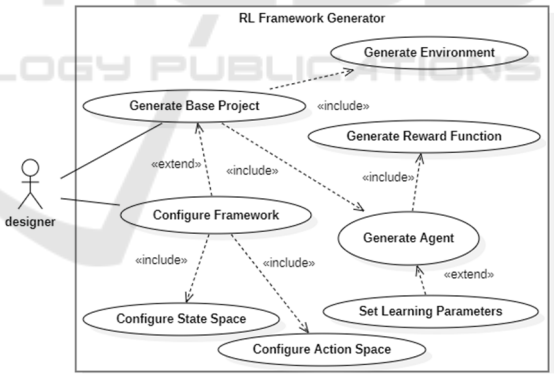

Figure 6: Principal functionalities of RL framework generator.

An overview of the principal functionalities of the RL framework generator is given in Fig. 6.

6 CASE STUDY

As a representative example, we use the FESTO production system (FESTO, 2016), to illustrate the main features of our proposal. The considered RCS complies with all the core reconfiguration character-

istics for manufacturing systems. The FESTO system is composed of three units that cooperate to produce drilled workpieces. We assume in this research work two drilling machines DM1 and DM2 to drill pieces. If one of the drilling machines (DM1 or DM2) is broken, it is replaced by the other unbroken one. In the case where both DM1 and DM2 are broken, then the production system is stopped. The state space of our example is based on the combination of the constraints defined on 5 controlled elements (DM1, DM2, np, p, mode) (np is the number of pieces, p is the periodicity of production and mode corresponds to the operational mode of the system). According to the considerations mentioned before, the modeling of the case study follows the steps of the design process proposed in Section 5. For the evaluation of the RL framework, we compare two scenarios of implementation: we first implement manually the case study with a ε-greedy Q-Learning algorithm without considering conceptual constraints. At a second time we configure the generated framework with the parameters of the same case study (number of states, number of configurations, number of controlled elements, etc.). The RL algorithm is tested using the Python libraries *Simpy* (simulation) and *Scikit-learn* (Q-Learning) (Pyqlearning, 2016). The reconfiguration algorithm is implemented with the following default hyperparameters: the discount factor was $\gamma = 0.8$, the learning rate $\alpha = 0.001$, and $\varepsilon=0.05$. The performance of the reconfiguration algorithm and learning process are compared for different values of ε. For example, a smart agent explores ($\varepsilon=1$) and takes the future reward into account ($\gamma = 0.9$). A greedy agent cares only about immediate reward with $\gamma = 0.01$ and $\varepsilon=0.1$. The initial state is chosen randomly. An episode ends either after the agent reaches the goal or after 100 steps. In general, the RL-agent converges in Scenario 2 faster than in Scenario 1. This can be explained by the reduction of dimension of the design space when we consider the conceptual constraints, which implies the optimization of learning time.

7 CONCLUSION

In this paper, we propose a conceptual framework that integrates the benefits of UML-based modeling with RL concepts to handle the intelligent design of RCS allowing for reconfiguration model improvement through the exploration of run-time knowledge. At the first design step, reconfiguration knowledge is abstracted using UML diagrams. Thereafter, a RL-based model will be deduced from established UML models including state and action spaces. The RL-based framework is generated using a set of generation rules and allows to formulate the reconfiguration control problem as an MDP. Besides, we incorporate the controlled system properties, into the state space and discretize the action space for reconfiguration control. Moreover, we encode safety and reconfiguration time objectives into the reward function. The generated RL framework is then used by the reconfiguration controller which is designed as a RL agent to learn optimal reconfiguration policies by applying Q-learning algorithm. Finally, reconfiguration models can be improved using learning feedback. The proposed design approach progresses from the state of the art of RCS design by studying the coexistence of UML-based design with RL in a unified model. It also opens up further research opportunities. Several aspects will be addressed in future work. Firstly, we plan a complete implementation of our approach (specifically, the RL framework generator), as well as further experimentation and simulation on other case studies. We can also explore deep-learning based RL algorithms to solve scenarios with larger state and action spaces more efficiently.

REFERENCES

Bazydło G. (2023). Designing Reconfigurable Cyber-Physical Systems Using Unified Modeling Language. *Energies,* 16, 1273.

Bellman, R. (1957). A Markovian Decision Process. *Indiana University Mathematics Journal*, 6, 679-684.

Ben Hadj Ali, A. and Ben Ahmed, S. (2023). RLReC: Towards Reinforcement Learning-based Dynamic Design of Reconfiguration Control. In *the proceedings of the 27th International Conference on Knowledge-Based and Intelligent Information & Engineering Systems, KES-2023. Procedia Computer Science*, Volume 225, pages 3670-3680.

Ben Hadj Ali, A., Khalgui, M. and Ben Ahmed, S. (2012). UML-Based Design and Validation of Intelligent Agents-Based Reconfigurable Embedded Control Systems. *Int. J. Syst. Dyn. Appl.,* 1, 17-38.

Elmaraghy, H., Monostori, L., Schuh, G. and Elmaraghy, W.H., (2021). Evolution and future of manufacturing systems. *CIRP Annals*, 70, 635-658.

Fay, A., Vogel-Heuser, B., Frank, T., Eckert, K., Hadlich, T. and Diedrich, C. (2015). Enhancing a model-based engineering approach for distributed manufacturing automation systems with characteristics and design patterns. *Journal of Systems and Software,* 101*,* 221-235.

FESTO. (2016). MPSs—The modular production system.

MPSs—The modular production system. Retrieved October 30, 2022.

Garcïa, J. and Fernández, F. (2015). A comprehensive survey on safe reinforcement learning. *Journal of Machine Learning Research*, 16(1), 1437-1480.

Gogolla M., Buttner F. and Richters M. (2007). USE: A UML-based specification environment for validating UML and OCL, *Sci. Comput. Program*., vol. 69, no. 1-3, pages. 27–34.

Harel, D. and Politi, M. (1998). Modeling Reactive Systems with Statecharts: The STATEMATE Approach; *McGraw-Hill*, Inc.: New York, NY, USA.

IEC. (2005). Function blocks – Part 1: Architecture. Function blocks – Part 1: Architecture. 2005.

Koren, Y., Heisel, U., Jovane, F., Moriwaki, T., Pritschow, G., Ulsoy, G. et al. (1999). Reconfigurable Manufacturing Systems. *CIRP Annals*, 48, 527-540.

Kuhnle, A., Kaiser, J., Theiß, F., Stricker, N. and Lanza, G. (2020). Designing an adaptive production control system using reinforcement learning. *Journal of Intelligent Manufacturing,* 32, 855-876.

Lepuschitz, W. (2018). *Self-reconfigurable manufacturing control based on ontology-driven automation agents.* PhD Dissertation, Technische Universität Wien.

Mohamed, M. A., Kardas, G. and Challenger, M. (2021). Model-Driven Engineering Tools and Languages for Cyber-Physical Systems–A Systematic Literature Review. *IEEE Access*, vol. 9, pages 48605-48630.

Monostori, L., Kádár, B., Bauernhansl, T., Kondoh, S., Kumara, S., Reinhart, G. et al. (2016). Cyber-physical systems in manufacturing. *CIRP Annals*, 65(2), 621–641.

Müller, T., Kamm, S., Löcklin, A., White, D., Mellinger, M., Jazdi, N. and Weyrich, M. (2023). Architecture and knowledge modeling for self-organized reconfiguration management of cyber-physical production systems. *International Journal of Computer Integrated Manufacturing*. 36:12, 1842-1863.

OMG. (2014). Object Constraint Language Specification, version 2.4 (omg.org/spec/OCL/2.4/PDF).

OMG. (2017). Unified Modeling Language, version 2.5.1 (omg.org/spec/UML/2.5.1/PDF).

Oueslati, R., Mosbahi, O., Khalgui, M. and Ben Ahmed, S. (2016). A Novel R-UML-B Approach for Modeling and Code Generation of Reconfigurable Control Systems. In *the proceedings of the 11th International Conference on Evaluation of Novel Software Approaches to Software Engineering – ENASE.* SciTePress, pages 140-147.

Parant, A., Gellot, F., Zander D., Carré-Ménétrier ,V. and Philippot, A. (2023). Model-based engineering for designing cyber-physical systems from product specifications. *Computers in Industry*, Volume 145.

PI. (2016). Plattform Industrie 4.0–Aspects of the research roadmap in application scenarios. Plattform Industrie 4.0–Aspects of the research roadmap in application scenarios. Retrieved October 30, 2022.

Pyqlearning. (2016). pyqlearning. Pyqlearning. Retrieved October 30, 2022.

Saputri, T. and Lee, S. (2020). The Application of Machine Learning in Self-Adaptive Systems: A Systematic Literature Review. *IEEE Access, 8*, 205948-205967.

Schneider, G.F., Wicaksono, H. and Ovtcharova, J. (2019). Virtual engineering of cyber-physical automation systems: The case of control logic. *Adv. Eng. Inform.*, 39, 127–143.

Shengluo Y. and Zhigang X. (2022). Intelligent scheduling and reconfiguration via deep reinforcement learning in smart manufacturing. *International Journal of Production Research*, 60:16, 4936-4953,

Sutton R. S. and Barto AG. (2018). *Reinforcement learning: an introduction,* 2nd ed. MIT Press, Cambridge, USA.

Thramboulidis, K. and Christoulakis, F. (2016). UML4IoT—A UML-based approach to exploit IoT in cyber-physical manufacturing systems. *Comput. Ind*., 82, 259–272.

Thramboulidis, K. and Frey, G. (2011). Towards a model-driven IEC 61131-based development process in industrial automation. *Journal of Software Engineering and Applications*, 4(04), 217.

USE, (2021), https://sourceforge.net/projects/useocl/

Vyatkin, V. (2013). Software engineering in industrial automation, State-of-the-art. *IEEE Transactions on Industrial Informatics*, 1234–1249.

Watkins, C.J.C.H. and Dayan, P. (1992). Q-learning. *Machine Learning 8*, 279–292.

Wuest, T., Weimer, D., Irgens, C. and Thoben, K.D. (2016). Machine learning in manufacturing: Advantages, challenges, and applications. *Prod. Manuf. Res*., 4, 23–45.