An Intelligent Approach to Support Software Architecture Decision-making in the Context of Software Architecture Evaluation

Verónica Bogado[1,2]
[1]*CIT Villa María (CONICET-UNVM)*
Carlos Pellegrini 211, Villa María, Córdoba, Argentina
[2]*Departamento Ingeniería en Sistemas de Información FRVM, UTN*
Av. Universidad 450, X5900 HLR Villa María, Córdoba, Argentina
vbogado@frvm.utn.edu.ar

Eva Villarreal Guzmán
Departamento Ingeniería en Sistemas de Información
FRVM, UTN
Av. Universidad 450, X5900 HLR Villa María, Córdoba, Argentina
villarrealguzman@gmail.com

Silvio Gonnet, Horacio Leone
INGAR (CONICET – UTN)
Avellaneda 3657, S3002GJC Santa Fe, Argentina
{sgonnet,hleone}@santafe-conicet.gob.ar

Abstract

Software Engineering needs novel tools to pursue further the goals of achieving software quality, facing the changing role of software. In this context, Software Architecture plays a key role because it directly affects the final quality. Software Architecture Evaluation validates if the architecture achieves the quality requirements, and triggers a set of design decisions. The decision-making is a very complex process driven by several human factors. It is argued that Artificial Intelligence-based practices can assist this process. In this work, an Artificial Intelligence-based approach for assisting architects in the design decision-making process driven by quality attributes is proposed. This first version combines quality-attribute models and an intelligent agent to support software architecture evaluation. It applies Reinforcement Learning tools to obtain a sequential architectural pattern application policy by simulation. A case study and a set of experiments illustrate the proposal with patterns commonly used in software industry.

Keywords: Software Architecture Evaluation, Design Decision-making, Artificial Intelligence.

1. Introduction

Software is a transversal system, which impacts in the activities of any organization including companies, governments, and society in general. In the last few years, the demand from users and developers for software, which delivers greater functionality, reliability, performance, usability, and a lot of other attributes, spurs innovation and new paradigms. These challenges require a quick response in terms of methodologies and tools to conceive high-quality software.

In this context, the role of software quality becomes a fundamental issue for the software lifecycle [1], where software architecture design is a central practice in software development, due to the growing system complexity and the impact on the quality. Software Architecture (SA) is a means by which quality attributes are achieved, and risks and costs in complex technological projects are managed. The SA restricts the level of quality attributes; it is a bridge between requirements and detail design or implementation [2] and it is even a Design Plan for developing the product [3].

Software architectural design involves analysis, synthesis, and evaluation. In this process, architects must make decisions for architectural patterns, tactics, and gross decomposition of functionality in order to reason about the advantages and disadvantages of potential solutions [4]. It is argued that Artificial Intelligence-based practices can assist developers to search in the design space more effectively. Also, they have shown to be useful to support the design decision-making process [4].

In the present work, an approach based on Artificial Intelligence (AI) for assisting architects in the design decision-making process driven by quality attributes is proposed. This first version of the approach merges

quality-attribute models and an intelligent agent to support software architecture evaluation. Unlike other proposals, this approach is based on a quality model (driven by metrics) and applies Reinforcement Learning tools to obtain a sequential architectural pattern application policy by simulation. This policy allows analyzing in real-time the impact of the application of a sequence of patterns on the software quality when a software architecture is being evaluated. The proposal is centered on a quality and quantitative approach based on metrics that are indicators of quality attributes visible at runtime. This perspective allows integrating metric-software architecture evaluation approaches with a design support based on quality indicators. Therefore, agent learning is reached by the experience of applying a pattern and the impact on the quality of a particular system, using this knowledge in new projects.

2. Related Work

In software architecture design, analysis, synthesis, and evaluation are very close, requiring an iterative and incremental process. Software industry needs to improve the link between documentation and validation (through the SA as its means), requiring a measurable approach. Some industrial studies have shown how non-functional requirements influence architectural decisions and the importance of Performance efficiency, Reliability, and Usability on the software product [5]. Moreover, the importance of expressing non-functional requirements in a measurable form with the main purpose of validating them early and avoiding a time-consuming testing or even impossible has been emphasized [5].

Software architecture evaluation implies analyzing different quality attributes (part of non-functional requirements). In this process, a distinction can be made between external attributes that are visible at runtime, and internal attributes that are focalized on static aspects. Several approaches propose evaluating quality attributes based on quantitative/qualitative analysis such as [6], [7], [8], [9], [10]. In addition to architecture evaluation techniques, some works have focalized on improving the architecture specification for the assessment [11]. A late architecture evaluation has been proposed for the analysis of several quality attributes giving the possibility of making more assertive decisions. Other approaches attempt to provide a trade-off analysis among quality attributes that are visible at runtime, and the software is analyzed including quality and functionality [13]. In all cases, after software architecture evaluation, if the architecture was not validated (quality attributes were not fulfilled), design decisions should be made. It is important to highlight that non-functional requirements drive several types of decisions such as architectural patterns, implementation strategies, and technological platforms [5].

Software architecture decision-making becomes difficult when it involves many requirements from multiple perspectives and there are multiple solutions that satisfy the same requirements, and each of these solutions is likely to have a trade-off regarding quality attributes [4]. In addition, architects do not follow a systematic software architecture decision-making technique to make decisions. Instead, their decisions are mostly based on personal characteristics such as intuition and experience, following informal but structured approaches for decision-making; nevertheless, these approaches look like some of the existing systematic techniques. Finally, there are several factors, including time, money and organizational practices that make it difficult for decision-makers to conduct extensive analysis before making a decision [14].

Over the last years, AI techniques have been used in supporting the tasks of the process of developing software. In particular, Bayesian networks, Fuzzy Logic, among other practices were used to requirement analysis; Case Based Reasoning (CBR) and Constraint programming have been used to design systems; and Genetic Algorithms (GAs) have been applied to coding and testing. Specifically, in software architecture design, AI techniques use quality attributes to define a goodness function over the space of possible architectures; so, Genetic Algorithms or similar algorithms search in the space of possible hierarchical decompositions of a system [15].

There is clearly a growing trend of Intelligent Engineering. Machine Learning can increase the power of software systems and help developers to learn about problems and domains. In particular, Reinforcement Learning (RL) introduces exploration and exploitation to exhibit the intelligent learning behavior that is expected in many real-life and complex problems [16]. Architectural decisions can be anything (pattern, tactics, restrictions, etc.). Some authors suggest that the order in which decisions are taken may have an impact on the result (or, rather, the problem that is being solved) and the first decision may be particularly important [17]. In this way, RL practices allow developers to model and simulate this kind of problem [16].

3. Intelligent Architectural Decision-Support

Software architecture design is a key process of software development and for the success of the projects in software companies. This process involves a set of activities that are complex because of the nature of software design and the complexity of deciding for one alternative over other solutions, even more considering

that the quality of the solution must achieve the user requirements.

The general activities proposed in [18] are Analysis, Synthesis, and Evaluation (Figure 1). This work is focused on the last two activities. The architectural synthesis proposes architecture solutions to a set of requirements, thus it moves from the problem to the solution space. The architectural evaluation ensures that the architectural design decisions made are the right ones, i.e., it ensures that the SA achieves the quality requirements. There is a direct relation between architectural synthesis and evaluation. In consequence, it is important to pay explicit attention to decisions in architecture assessments [17]. If the SA did not fulfill the quality attributes in the measure required, new architectural decisions should be made.

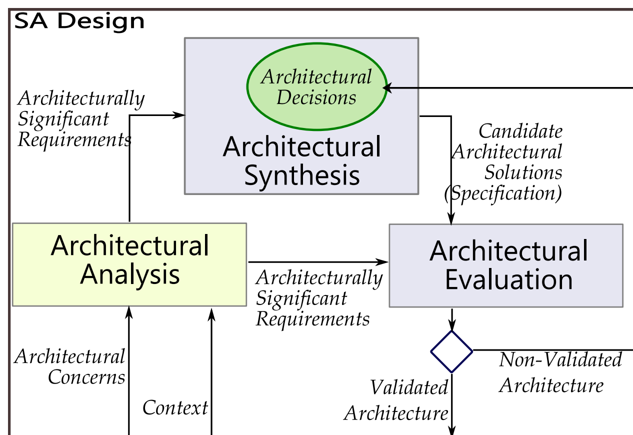


Figure 1. A general model of software architecture design, adapted from [18].

Although tools for SA specification exist, these tools do not support architects in making informed decisions driven by quality attribute measures. This work proposes an agent (design assistant) for recommending the use of some patterns to satisfy quality attribute scenarios, but considering functional scenarios too, i.e., the assignment of responsibilities to components.

3.1. Architecture Analysis and Specification

The first entries are the Functional Requirements (FRs) and Non-Functional Requirements (NFRs), particularly quality requirements (QRs) related to Quality Attributes (QAs). The FRs describe the system scenarios, which define the responsibilities of each software component. Quality Requirements specify the responses of the software to realize business goals [2]. Note that it is important to clearly specify the value of each quality requirement to be validated quantitatively. A good template for doing this is Quality Attribute Scenarios [2].

Several factors are synthesized to obtain the software architecture specification using Use Case Map (UCM) notation. UCM represents a SA and system scenarios [19]. This visual notation allows architects to model complex dynamic systems making the work of analyzing the SA easier. The main reason for choosing this notation is that UCM builds functional scenarios together with software structures by means of overlapping paths composed of responsibilities and other elements, all in the same model. This notation represents units that have presented at runtime (dynamic view of the software). This view allows architects to analyze quality attributes that are visible at runtime, i.e., during the software operation.

In particular, the basic elements of this notation are [19]:

- *Responsibility*: it is high level abstraction interface or interaction. In a software architecture, the responsibility is an action that a component is responsible for executing.
- *Team*: it is used to represent both kinds of software components, simple or composite ones. It is a high level entity that can group other smaller entities in more detail.
- *Start Point*: it is the beginning of a scenario (or path) by means of a stimulus.
- *End Point*: it indicates the end of a scenario (or path).
- *Path*: it is a system scenario, connects start points, responsibilities, and end points.

Nowadays, software companies have to build a database about the quality information of the products developed during the execution of projects in order to improve the product quality and, in consequence, the software development process. On the other hand, the use of patterns is very beneficial to improve the quality of design solutions and to speed up the software development process. In the first version of the present approach, an architectural decision is given by the application of an architecture pattern because it implies selecting one of alternative solutions. The architect makes a decision associated with the pattern in a specific context using implicitly quality knowledge learned from previous applications. For example, an architect might consider two alternative patterns: Model-View-Controller or N-Tier; each one has impacts on attributes such as modifiability, reliability, and performance. The decision is driven not only by structural features of the pattern but also by implicit knowledge of the impact of the pattern on the final system quality. In this way, it is mandatory that companies can capture this know-how to make it an asset of the organization. Addressing this information in a software assistant would provide an architectural decision mechanism for the software architecture evaluation that is independent of the person, keeping this

knowledge in the organization. Another important issue is that architectural patterns can be catalogued, which makes the automation of the pattern selection process easier [2].

3.2. Architecture Evaluation

Software architecture evaluation needs quality information. Metrics and indicators play an important role for a more objective assessment; so, architects can make well-founded strategic decisions. There are several practices to make the evaluation, but due to the quantitative approach of this proposal, simulation (such as [13] or similar) is suggested to train and consult the agent. The simulation provides the quality indicators used by the intelligent agent to propose adjustments in the SA by a pattern-based improvement suggestion. Here, it is important to consider the concept of *trade-off*, which means that the improvement of one quality comes at the cost of degrading another, as it is the case of modifiability versus performance. In this work, several indicators are considered, which allow architects to analyze partially the following quality attributes: performance, reliability, availability, and the impact on the usability.

3.2.1. Quality Metrics for Trade-off: The considered characteristics (attributes) and subcharacteristics follow the standard ISO25000 [20]are: Performance Efficiency (Time-behavior), Reliability (availability), and their influence on quality in use for primary users. These characteristics and the defined metrics to measure them are chosen taking into account the following considerations: the availability of human resources to collect data, the ease of data collection, the number of users of the information products utilizing the indicator, the repeatability and reproducibility of quality measure elements, and the target entity about which information is kept [20].

Equation (1) shows the definition of the metric *System Turnaround Time* taken as indicator of Time-behavior and Equation (2) defines the metric *System Failures* taken as indicator of availability, from the analysis of information about reliability.

$$STT = ((rtt_{1,1}+ rtt_{1,2} +\ldots+ rtt_{1,n}) +\ldots+ (rtt_{m,1}+ rtt_{m,2} +\ldots+ rtt_{m,n}))/m. \quad (1)$$

Where *rtt* is the responsibility turnaround time, *n* is the total number of responsibilities that are involved in the system scenario, and *m* is the total number of requests sent by users to be processed.

$$SF = ((rf_{1,1}+ rf_{1,2} +\ldots+ rf_{1,n}) +\ldots+ (rf_{m,1}+ rf_{m,2} +\ldots+ rf_{m,n}))/m. \quad (2)$$

Where *rf* is the responsibility failure, *n* is the total number of responsibilities that are involved in the system scenario, and *m* is the total number of requests sent by users to be processed. A responsibility can fail only once per request, assuming that when a failure occurs in an execution, the system fails to deliver and continues the execution of the next responsibility, i.e., *rf*= 0 (without fail) or 1 (fail). A fail could bring problems to the successor responsibilities due to the causal relationship among the responsibilities.

3.3. Intelligent Agent for Design Decisions

RL is inspired by behaviorist psychology and it is concerned with how software agents must take actions in a dynamic environment maximizing some notion of cumulative reward [21]. A *RL-based Agent* is capable of learning from the interaction with an environment exploring and exploiting (Figure 2) the knowledge driven by a specific goal. The *Agent* exploits what it already knows in order to obtain a reward, but it can also explore in order to make better action selections in the future [16]. In the architectural design decision-making problem, the agent needs to know about software architectures specified using UCM notation (environment that represents software dynamics), explores new Design Decisions (patterns), and exhibits intelligence in known or new scenarios (UCM with quality data). ε-greedy is a simple exploration method, where the agent chooses the action that it believes to have the best long-term effect with probability $1 - \varepsilon$ and it chooses an action uniformly at random, otherwise [21]. ε is a real number, whose possible values are between 0 and 1 inclusive.

The environment is usually formulated as a Markov Decision Process (MDP) [21], [16]. In this design decision problem, the environment represents the dynamics of the software through its executed and evaluated architectural design solution (Figure 2). The quality indicators obtained during the evaluation are used as *reward* (*r*) associated to the application of the selected pattern (*action a*) at a decision time *t*. This *r* feedbacks the agent, so when it perceives a similar state, the pattern with the better cumulative reward will have more probability of being selected.

A *state* (*s*) captures the software architecture specified as UCM at time *t* (Figure 2). An example of a state is illustrated in Figure 3 as s_1. The UCM model is composed of dynamic software elements including simple and composite components (Dynamic structure), functional elements that represent the system scenarios (or paths), and measurable quality data (metadata) related to the pattern in the context of application. In this case, the collected data is related to turnaround time and failures at low level (responsibility).

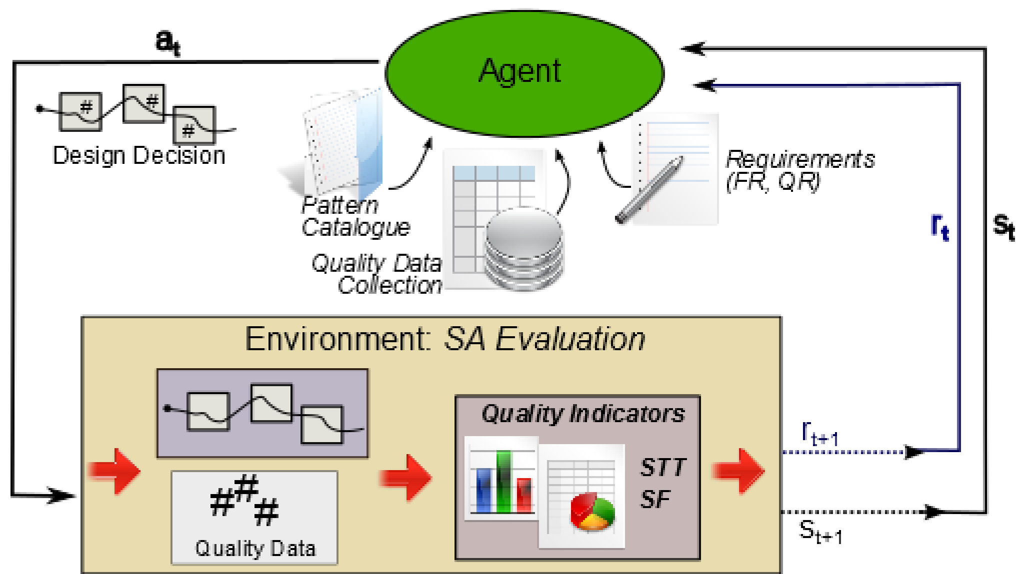


Figure 2. Software architecture decision-making using RL approach.

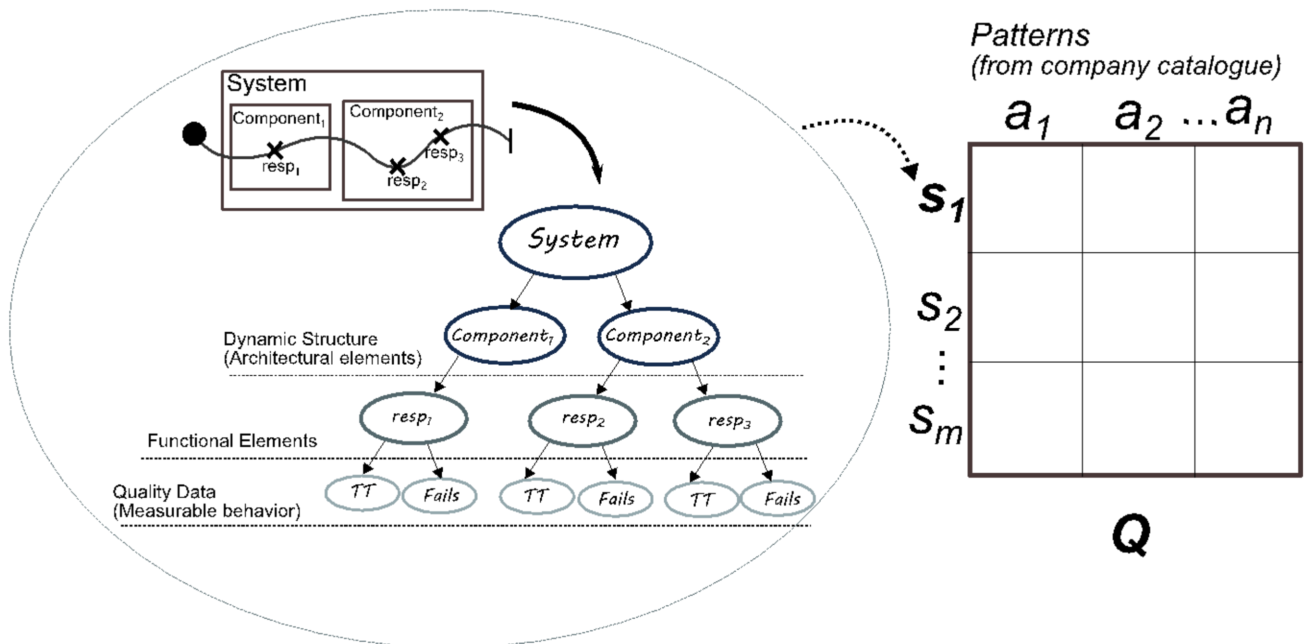


Figure 3. Sample of a state (*s*), action (*a*), and *Q*.

The learning results in the selection of an action or a sequence of actions [16]. An *action* (*a*) is the application of a pattern on the model leading to another UCM that includes the selected pattern in the context. This pattern is chosen from a company catalogue conformed by the experience of the application of the pattern in previous projects, i.e., quality data collection (or *Pattern Catalogue* in Figure 2).

The *Agent* receives a feedback from the environment at time *t* (Figure 2). The *reward* (*r*) (Equation(3)) was defined for a trade-off of the mentioned quality attributes. So, the first term of the Equation (3) considers the performance indicator *STT*, the requirement (*TTReq*), i.e., the desired value, and the priority of this indicator over the global system. The second term considers the reliability (availability) indicator *SF*, the corresponding requirement (*SFReq*), and the defined priority for this requirement.

$$reward = (1 - STT/TTReq)*1/TTReqPriority + (1 - SF/SFReq)*1/SFReqPriority. \quad (3)$$

The priority has to be defined with a descending scale. Two examples of scales commonly used in software companies are defined in Table 1.

Table 1. Quality requirement priority scales.

Name	Value	Meaning
Scale 1	1-*High*	Critical requirement; it is mandatory for the next release.
	2-*Medium*	Eventually necessary requirement; it could wait until a later release.
	3-*Low*	Improvement; it would be nice to include it in the next release, but if there are enough resources to implement it.
Scale 2	1-*Essential*	It must be satisfied to make the product acceptable.
	2- *Conditional*	It would enhance, but the product is not unacceptable if absent.
	3- *Optional*	It is useful, but the product is acceptable if absent.

Summarizing, Architectural Design Decision-making process (Figure 4) in software architecture evaluation context using UCM notation to specify software architectures can be seen as a sequence of UCMs. A monolithic design, where no pattern is applied or where no decision has been taken yet, is the first UCM which must be validated against quality requirements (Evaluation). Following this quality goal, a set of design decisions can be made passing from a UCM to another, which includes the applied design decision. This process can be modeled as a MDP (Figure 4), where there is an initial state and in each state an action that has quality data can be performed (architectural pattern application). Furthermore, a reward, which includes the quality indicators, feedbacks the decision passing to another state until the goal state is achieved, i.e., quality requirements are fulfilled.

The following two learning mechanisms have been implemented in this approximation of the proposal, applying an ε-greedy exploration-exploitation policy, where each episode consists of an alternating sequence of states and state–action pairs as it is shown in Figure 4 [21].

- **Q-Learning:** is an off-policy temporal difference (TD) control algorithm and in its simplest form is defined by the next update rule:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma max_a Q(s_{t+1}, a) - Q(s_t, a_t)].$$

The learned *action-value* function (*Q* matrix in Figure 3) directly approximates q^*, the optimal action-value function, which defines π^* (optimal policy) independently of the exploration-exploitation policy being followed (ε-greedy in this case). The policy still has an effect in the sense that it determines which *state–action* pairs are visited and updated. However, all that is required for correct convergence is that all pairs continue being updated.

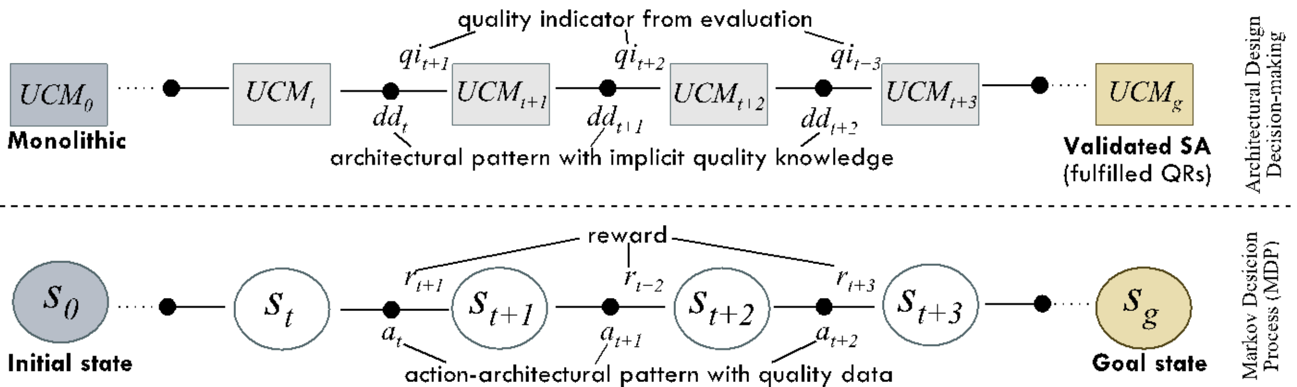


Figure 4. Architectural design decision-making as a MDP.

Figure 5 shows the Q-learning algorithm incorporated within the Agent, where the learning happens throughout each episode (occurs over a period of time) by observing a state s, doing an action a, and obtaining a reward r from the environment as consequence of its performance in it. In this case, the Agent learns the optimal policy, i.e., the optimal pattern application sequence independently of the pattern most commonly used by the company, the architects, or the agent in this case.

```
Initialize Q(s, a) arbitrarily, Q(terminal-state, ·) = 0

Repeat (For each episode):
  Initialize s

  Repeat(For each step of episode)
    Choose a from s using policy derived from Q
    Take action a
    Observe reward r, s'
    Q(s, a)←Q(s, a) + α[r + γmax_A Q(s',A) − Q(s, a)]

    s←s'
  until s is terminal

   (A: set of possible actions in state s', policy: e.g. ε-greedy)
```

Figure 5. Q-learning: Off-policy TD control algorithm.

- **Sarsa:** is an on-policy temporal difference (TD) control algorithm, whose update rule is:

$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\ [r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)].$

Figure 6 shows the algorithm used by Sarsa learning mechanism.

```
Initialize Q(s, a) arbitrarily, Q(terminal-state, ·) = 0

Repeat(For each episode)
  Initialize s
  Choose a from s using policy derived from Q

  Repeat(For each step of episode)
      Take action a
      Observe reward r, s'
      Choose a' from s' using policy derived from Q
      Q(s, a) ← Q(s, a) + α [r + γQ(s', a') − Q(s, a)]
      s←s'
      a←a'
  until s is terminal

                                   (policy: e.g. ε-greedy)
```

Figure 6. Sarsa: On-policy TD control algorithm.

This mechanism uses state-action-reward-state-action experiences to update the Q-values (Q matrix in Figure 3). This updating is done after every transition from a nonterminal state s_t. If s_{t+1} is terminal, then $Q(s_{t+1}, a_{t+1})$ is defined as zero. As in all on-policy methods, q_π is continually estimated for the behavior policy π, and at the same time it changes π toward greediness with respect to q_π. With this learning mechanism, the Agent learns a policy that consists of the most frequently chosen pattern application sequence in the course of the software architecture design process,i.e., the most repeated set of design decisions, which have been made after the architecture evaluation.

4. Implementation

Software architecture design requires computational tools that can assist the decision-making process. Recent design tools check conditions in the object-oriented models being developed, but very few tools pay attention to software architecture design, and even less to the importance of this intermediate product on the quality of the final product.

In this work, a prototype with the simulation environment and the agent was designed and implemented using JAVA programming language.

This prototype of the design decision-making tool has two main parts (Figure 7): *Agent* and *Environment*. The agent (*RLAgent*:*Agent*) has two learning mechanisms (*Q:LearningEngine* and *Sarsa*:*LeaningEngine*) for resolving the architectural problem. Both Q learning and Sarsa are implemented conforming the learning Engine of the *Agent*. A third algorithm, Value Iterator (*VI:ValueIterator*) was implemented to validate the knowledge generated by using the other mechanisms. The environment (*EnvironmentSimulator*) gives life to the UCMs. It simulates the software operation following the parameters set by the architect in the UCM metadata and calculates the quality indicators.

The *Agent* observes the Environment and obtains a state (UCM) from it. The *Agent* makes a design decision and the simulator reproduces the execution of the software by means of the UCM. Then, the *Environment Simulator* returns the quality indicators, which have been taken dynamically. Both, the *Agent* and the *Environment*, require the UCMs to represent the states and to choose the pattern as an action. *jUCMNav* is the editor used for the specification of the UCM models, which include software architecture alternatives [22].

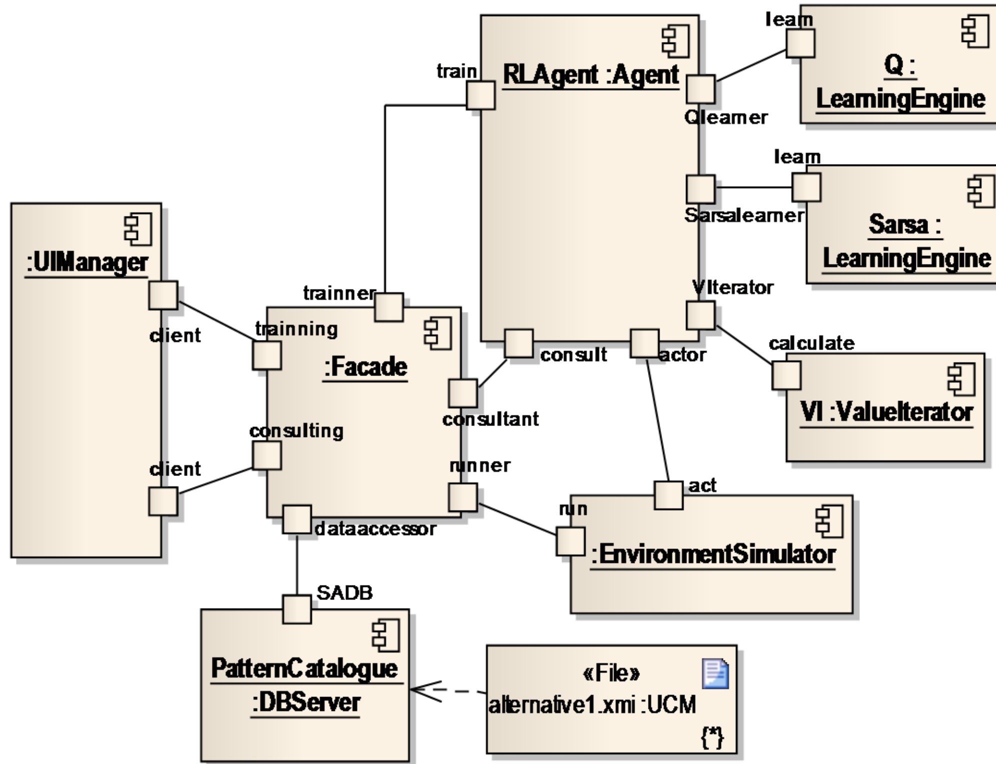


Figure 7. Architecture: component view of the RL-Agent prototype.

This tool runs under the development environment *Eclipse.* A *Facade* structures the tool and provides a simplified interface to reduce the complexity of the agent, the environment, and the *Pattern Catalogue* of the company. It makes the *Agent* interface easier or simpler, the library of UCMs (*PatternCatalogue:DBServer*) more readable, and reduces dependencies among different components or clients (*UIManager*). The *UIManager* provides a graphical user interface to show the results, which allows the architect to analyze the learning and the correct application of the knowledge generated by the Agent.

The software company has to feedback the database of UCMs (*PatternCatalogue*) with complete software architecture models including quality information needed for the simulation of the system and to calculate the indicators.

5. Results and Discussion

A case study and a set of experiments have been developed with the purpose of validating the proposal. The system is a management software commonly used by software factories, or other organizations, to control their system licenses. This license manager software (LM), provided by a software factory, controls where and how their software products are able to run (more detail can be found in [13]).

5.1. Architecture and quality requirements

The initial specification of the system considers the FRs for building a monolithic architecture that contains a scenario. The quality requirements (two examples are in Table 2) define the response measures, which are used to evaluate the architecture and to make the design decisions (Agent).

Table 2. Quality requirements specified using Quality Attribute Scenarios [2].

ID	Stimulus	Source of Stimulus	Artifact	Environment	Response	Response Measure	Priority
QR01	Request	User	*LMSytem*	Normal	Authenticated license	Turnaround time<10 sec	*Medium*
QR02	Fail to respond to a request	Responsibility	*LMSytem*	Normal	Recorded fail	Total failures < 1	*High*

jUCMNav is used for the specification of the UCMs [22]. This tool allows architects to graphically specify the architecture components, the functionality by means of the paths, and the quality data by means of defining metadata (as can be seen in Figure 8). These last ones are obtained from data collection (historical information of previous projects). The UCM specifications are saved as *xmi* files.

5.2. Patterns and quality data

A set of patterns is part of the convention to work in each project of the company. Also, in the design process, the criterions for choosing a pattern are based on an implicit knowledge of the quality impacts on the system. Companies would be capable of distinguishing and separating the quality information corresponding to each pattern application. This good practice allows companies to be independent from the architect (person) and to share this knowledge among the team members and among different projects. The architectural patterns allow architects to manage and improve the software architecture design, reducing the complexity of the decision task. In this case, the patterns used to test the agent learning (conforming the catalog for the project) are [23], [24], [25]:

- *Monolithic*: there are no architecturally separate components and the functionality is all interwoven. It does not promote any quality attributes and it can even inhibit some of them.
- *Pipe-Filter*: it transforms input data. It has two types of components, Filter and Pipe. The Filter transforms the data. The data flows from one Filter to another one across the Pipes.
- *Client-Server*: the system is organized as a set of services that are provided by components called servers, and other components called clients access to those services.
- *Broker*: it is responsible for coordinating communication, forwarding requests or transmitting results.

5.3. Evaluation and design decisions

The *xmi* files (UCMs) are loaded in the tool that contains the IA agent. The UCMs are mapped to a Markov model by an automatic transformation (from model to model directly). Then, the RL algorithm is chosen between Q-Learning and SARSA. The parameters of the two algorithms are set in the following form: ε= 0.2, γ= 0.2, α=0.6, and *episodes*=1000.

The evaluation is run simulating the software execution, giving life to the UCM model (system dynamic view). The behavior of the responsibility is simulated following the input information (means, probability distribution, etc.). The quality indicators are obtained and used to calculate the reward for each state-action. The agent tries to apply patterns, whose rewards allow the fulfillment of the QRs, or to get as close as possible to their fulfillment (r>=0).

The optimal policy is given by the best action that could be chosen in each state. In Figure 8, the found policy applying Q-learning algorithm is highlighted (orange). The learning curves are shown in Figure 9 under this first scenario. The first learning curve corresponds to the application of the Q-learning mechanism stabilized before the 100 episode for this example. The learning with Sarsa was similar. Sarsa starts with worst rewards, but it arrives a little more quickly to a feasible solution, and its average cumulative reward along the episodes is stabilized closer to 0 (fulfilled QRs). No algorithms could achieve the QRs, but both found a good sequence of design decisions that are near to the expected quality measures. This policy is similar for the two cases. Finally, Value Iteration (VI), which is a Dynamic Programming (DP) algorithm [21], is implemented to validate the results and the learning of Q-learning agent. The last curve shows the error between the Q-learning and VI and how it is minimized over the iterations.

The agent learned about software architectural decisions through experiments. The policy found using both algorithms, Q-learning and Sarsa, was the same, obtaining a sequential pattern application: *Monolithic* ⇨ *Client-Server* ⇨ *Broker* as the best combination of patterns for this example under the conditions given by the environment. The state which had the best value was *Client-Server and Broker.* This result indicates that it was the state in which the quality requirements were closer to be reached. In consequence, taking into account the results in this example, the optimal policy coincides with the most chosen solution.

It is important to note that the coincidence in the results of both learning mechanism is perhaps because of the number of states, i.e., the MDP is small in this example. If a more complex combination of patterns and quality information were taking into account, the policy could be different due to the way the future reward is found in the Q function. In Q-learning, it’s simply the action with the highest value the one which can be chosen from a given state, and in SARSA it’s the value of the actual action that was taken in the given state. This means that Q-learning simply assumes that an optimal policy is being followed while SARSA takes into account the control policy by which the agent is moving, and incorporates that into its update of action values. However, in both cases, there is also a chance that some random action will be chosen; this is the built-in exploration mechanism of the agent.

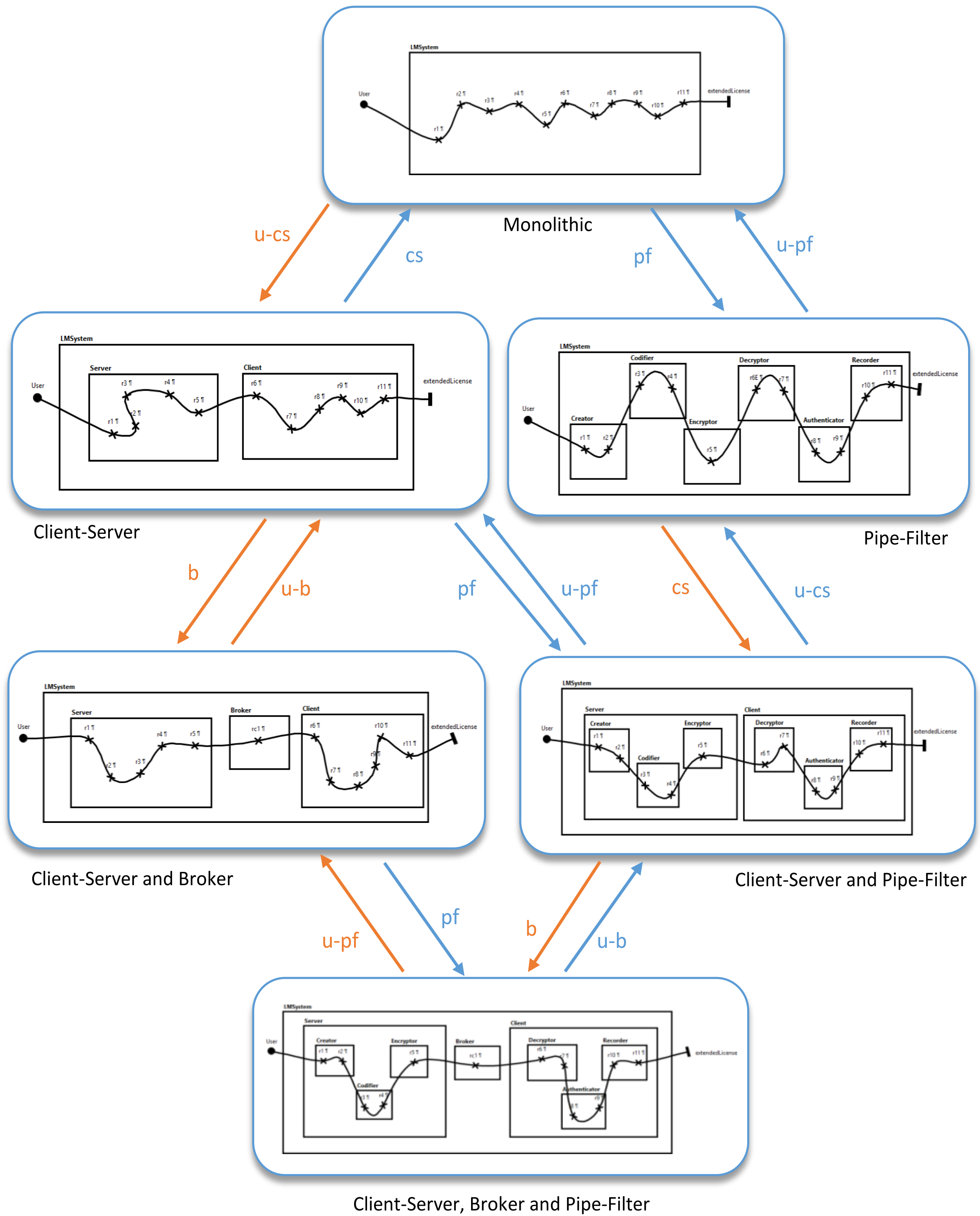
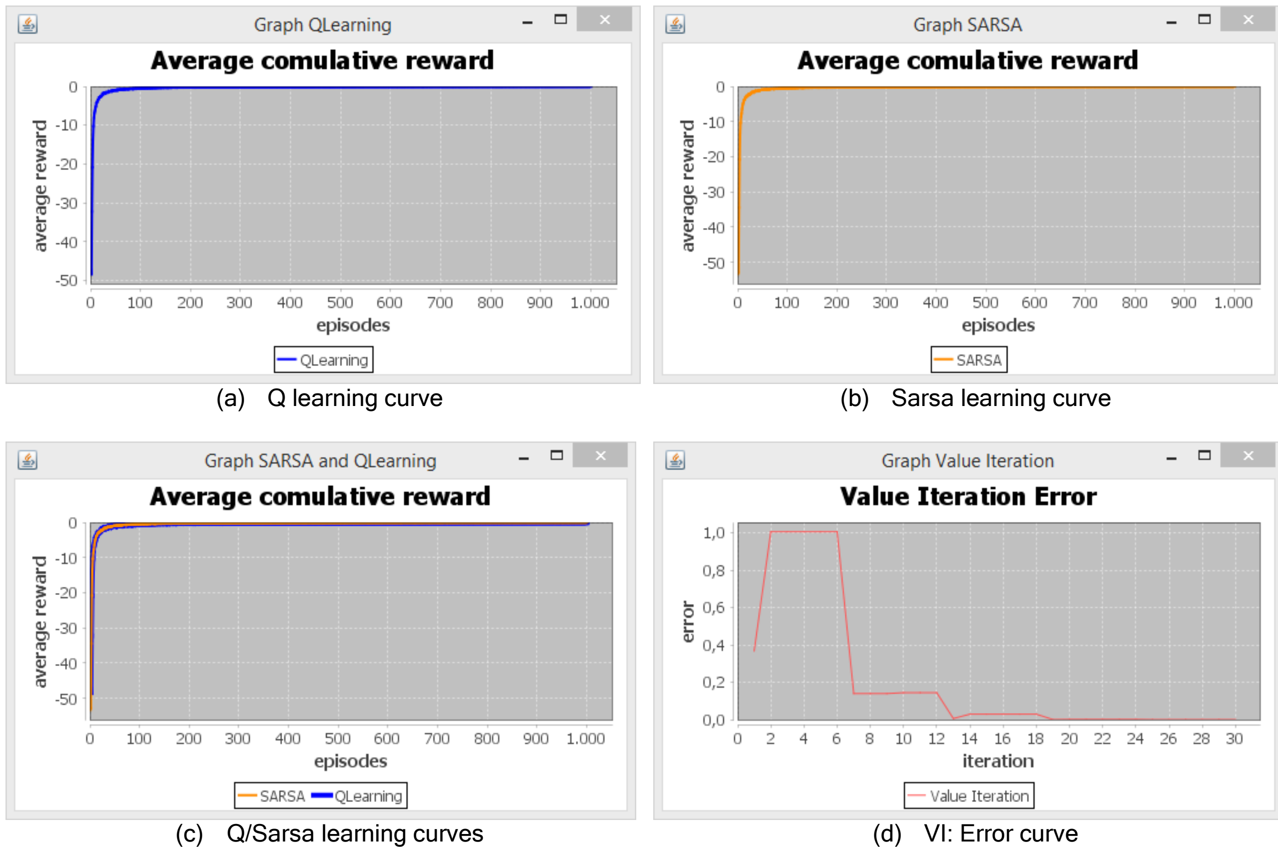


Figure 8. State transition diagram with possible actions for *LMSystem*.



(a) Q learning curve (b) Sarsa learning curve

(c) Q/Sarsa learning curves (d) VI: Error curve

Figure 9. Learning curves of experiments (Scenario 1).

A second scenario was defined to test the correct learning of the agent introducing changes in the environment during the simulation. Firstly, Pipe-Filter pattern time means were modified by duplicating them. This directly affected the *STR* indicator; then, the reward was more negative. Secondly, in another experiment, Pipe-Filter pattern failure rates were modified in a similar form (affecting the *SF* indicator). Finally, Pipe-Filter quality data (means, failure rate) were manipulated to force this option to be the worst action among all the possible alternatives. The results were as expected. The agent learned that this state is not an alternative under these new conditions, choosing always the other alternatives. In other words, this state had a lesser Q-value and the worst reward (-2.9964).

A third scenario was focalized on the quality requirements. The experiment introduced a modification in the response measures, duplicating the values to relax them. Figure 10 shows the learning curves obtained through the experiments. In this case, the requirements were not achieved, but a good solution was found, where the learning curves were stabilized very close to 0. The convergence of the learning curves was quicker than in the previous experiments (scenario 1). It is also important to see the error curves in Figures 9 (d) and 10 (d), where the error is stabilized after the iteration 18 in the first case and after the iteration 13 in the second case.

Software architecture design is a very complex process. Particularly, software architecture evaluation involves several issues such as software structure, software quality (including requirements, metrics, and indicators), and design decisions. So, the decision-making becomes a multifaceted activity. A complex state is necessary to better represent the domain, which involves: structural (pattern and other components), behavior (responsibility), and quality data (measurable form).

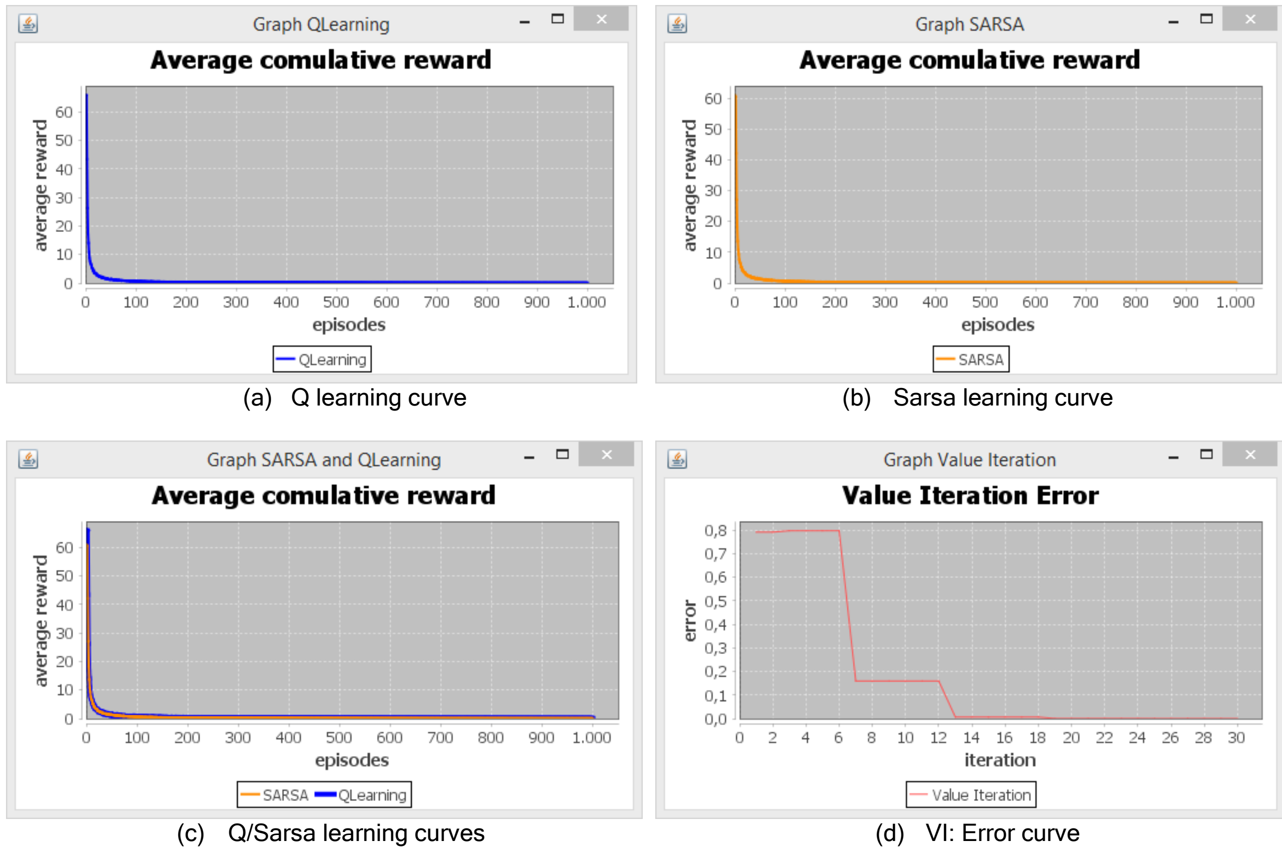
The UCM notation allows architects to define all mentioned aspects in the same model, providing a complete dynamic view of the software in an early stage of the development. This model is executed by simulation, and a set of quality indicators are used to define a reward that feedbacks the agent. Furthermore, the quality requirements define the learning goal and the learning parameters of the algorithms can affect the results too.

All these variables can be analyzed and, in consequence, they have effects on the learning and on the solutions proposed by the Agent. The advantage is that this RL-approach let the agent try and learn from the environment (software architecture and quality indicators obtained from the evaluation) as it continues evolving.

Reinforcement learning differs from typical supervised learning in the fact that correct input/output pairs are never presented, and no sub-optimal actions are explicitly corrected. In addition, there is a focus on on-line performance, which involves finding a balance between exploration (uncharted solutions) and exploitation (current knowledge) [21].

With Q-learning mechanism, the Agent can learn an optimal policy no matter what the agent does, as long as it explores enough; while with Sarsa, the Agent can learn the better solutions that have been chosen previously.

In this work, the initial state is a monolithic architecture, where no pattern is applied. But, the Agent could learn from any state due to it tries to achieve a quality goal, i.e., a trade-off among the quality requirements specified in relation to quality attributes (performance and availability-reliability).



(a) Q learning curve

(b) Sarsa learning curve

(c) Q/Sarsa learning curves

(d) VI: Error curve

Figure 10. Learning curves of experiments (Scenario 3).

6. Conclusion and Future Work

In software companies, making decisions is surely the most important task for software architects and it is often a very difficult one. This work presented a proposal based on artificial intelligence techniques and methodologies for assisting architects during software architecture design with the purpose of giving a concrete support to software architecture evaluation. With this proposal, the architect is not only able to analyze several quality attributes with the same evaluation model, but he is also able to acquire an integrated view including functional aspects in the SA evaluation. The main issue is the metric-based quality approach over which design decisions can be taken.

This work proposed a perspective of a software architecture based on quality models. With this perspective, the agent is capable of perceiving the software structure, behavior, and quality impact of

applying a pattern from previous experiences. The agent learns from the interaction with the environment (SA alternative and the quality indicators from the evaluation). Nowadays, design tools check conditions in detailed models, but do not focus on either on the software architecture or in the architecture evaluation. This proposal can help in the activities of evaluating and improving the software architectures by the simulation of software operation, obtaining quality indicators and suggesting patterns (design decisions) according to the quality requirements.

Finally, a case study was developed using a traditional software architecture and historical data from a company project database. Two RL algorithms (Q-learning and Sarsa) were implemented and Value Iteration was used to validate the values returned by the other two algorithms. It is important to note the complexity of software architecture design-making process, even more considering a metric-based approach and a software quality model.

As future work, it is interesting to train the agent with more complex systems, including other architecture patterns, new quality attributes such as security, and/or new metrics for performance, availability, and reliability. Hierarchical RL could be applied in order to improve the decision-making process and its combination with DEVS-based software architecture evaluation [13]. Finally, the development of a software tool that can be integrated into a software development environment used by architects is another future issue.

7. Acknowledgements

This work was supported by the Consejo Nacional de Investigaciones Científicas (PIP 112 20110100906) y Técnicas (CONICET), Universidad Tecnológica Nacional (PID UTI3803TC and PID UTN3581).

The authors gratefully acknowledge the contributions of the training program “Prácticas Profesionalizantes” of Tecnicatura Superior en Traducción, Instituto Técnico Superior Victor Mercante, carried out at Universidad Tecnológica Nacional, Facultad Regional Villa María, for their linguistic revision.

8. References

[1] Nielsen, P. D., "Software Engineering and the Persistent Pursuit of Software Quality", *CrossTalk-Journal of Defense Software Engineering*, May/June 2015.

[2] Bass, L., Clements P., and Kazman, R., *Software Architecture in Practice,* Addison-Wesley, MA, 2013.

[3] Hofmeister, C., Nord, R.; Soni, D., *Applied Software Architecture,* Addison-Wesley Professional, 2000.

[4] Meziane, F., and Vadera, S., *Artificial Intelligence Applications for Improved Software Engineering Development: New Prospects,* Information science reference, NY, 2010.

[5] Ameller, D., Ayala, C., Cabot, J., and Franch, X. "Non-functional Requirements in Architectural Decision Making", *IEEE Software*, March/april 2013, pp. 61-67.

[6] Clements, P., Kazman, R., and Klein, M., *Evaluating Software Architectures: Methods and Case Studies,* Addison-Wesley, MA, 2002.

[7] Wen-Li Wang, Dai Pan, and Mei-Hwa Chen, "Architecture-based Software reliability modeling", *J. Sys. Software*, 79, 1, 2006, pp. 132-146.

[8] Sharma, V. S., and Trivedi, K. S., "Quantifying software performance, reliability and security: an architecture-based approach", *J. Sys. Software*, 80, 4, 2007, pp. 493-509.

[9] Fukuzawa, K. and Saeki, M., "Evaluating Software Architecture by Coloured Petri Nets", in *Proceedings 14th International Conference on Software Engineering and Knowledge Engineering,* Italy, 2012, pp. 263-270.

[10] Rathfelder, C., Klatt, B., Sachs, and K.; Kounev, S., "Modelling Event-based Communication in Component-based Software Architectures for Performance Predictions", *Softw. Sys. Model.*, 13, 4, 2013, pp. 1291-1317.

[11] Brosch, F.; Koziolek, H.; Buhnova, B.; Reussner, R. Architecture-based reliability prediction with the Palladio Component Model, IEEE T. Software Eng. 38, 6 (2012).

[12] Christensen, H.; Hansen, K. An empirical investigation of architectural prototyping. J. Syst. Software 83, 1(2010), 133-142.

[13] Bogado, V., Gonnet, S., and Leone, H., "Modeling and Simulation of Software Architecture in Discrete Event System Specification for Quality Evaluation", *SIMULATION*, 90, 3, 2014, pp. 290-319.

[14] Dasanayake, S., Markkula, J., Aaramaa, S., and Oivo, M., "Software Architecture Decision-Making Practices and Challenges: An Industrial Case Study", in *Proceedings 24th Australasian Software Engineering Conference*, Adelaide, Australia, 2015.

[15] Ammar, H., Abdelmoez, W., and Hamdi, M. S., "Software Engineering Using Artificial Intelligence Techniques: Current State and Open Problems", in *Proceedings 15th International Conference on Computer and Information Technology*, Bangladesh, 2012.

[16] Kulkarni, P., *Reinforcement and Systemic Machine Learning for Decision Making*, IEEE/Wiley, US, 2012.

[17] Van Vliet, H., and Tang, A., "Decision Making in Software Architecture", *J Sys Software* 7, 2016, pp. 1-7.

[18] Hofmeister, C., Kruchten, P., Nord, R.; Obbink, H., Ran, A., and America, P., "A General Model of Software Architecture Design derived from Five Industrial Approaches", *J Sys Software*, 80, 1, 2007, pp. 106-126.

[19] ITU-T Z.151 Series Z, *Languages And General Software Aspects for Telecommunication Systems*, Formal description

techniques (FDT) – User Requirements Notation (URN)-Language definition, ITU, 2012.

[20] ISO/IEC 25000. Systems and software engineering--Systems and software Quality Requirements and Evaluation (SQuaRE), 2014.

[21] Sutton, R., Barto, A., *Reinforcement Learning: an introduction*, MIT Press, MA, 1998.

[22] jUCMNav, URL: http://jucmnav.softwareengineering.ca. Accessed: 07-2016.

[23] Fowler, M., Catalog of Patterns of Enterprise Application Architecture, URL: http://martinfowler.com/eaaCatalog/, Accessed: 07-2016.

[24] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M. Pattern-oriented Software Architecture, Volumen 1: A System of Patterns, John Wiley & Sons Ltd, 1996.

[25] Buschmann, F., Henney, K., Schmidt, D. C., Pattern-oriented Software Architecture, Volumen 5: On Patterns and Pattern Languages, John Wiley & Sons Ltd, England, 2007.