Learning Recovery Strategies for Dynamic Self-heal in Reactive Systems

Mateo Sanabria
Systems and Computing Engineering Department, Universidad de los Andes, Colombia
m.sanabriaa@uniandes.edu.co

***Abstract*—Self-healing applications generally depend on a set of predefined instructions that the system must follow in order to recover from a failure state. Such actions are triggered from predefined hooks in the program. Moreover, self-healing strategies detect failure states based on message response times, or metrics that are not expressive enough to detect different types of failures. Such strategies are usually applied in the context of distributed systems, where the detection of failures is constrained to communication problems, and resolution strategies often consist of replacing complete components. However, current complex systems may reach failure states at a fine granularity that were not anticipated by developers. For example, value range changes for data streaming in IoT systems. To counter these problems, we propose a self-healing framework that learns recovery strategies to heal fine-grained system behavior at run time. We demonstrate and evaluate our healing strategies in a new domain, reactive systems. Our proposal uses monitor predicates to define satisfiability conditions of the system state. Such monitors have functional expressivity and can be defined at run time to detect failure states. Once failure states are detected, we use a Reinforcement Learning-based technique to learn a recovery strategy based on users' corrective atomic actions. Finally, to execute the learned strategies we define them as Context-oriented Programming variations that activate at run time whenever the failure state is detected, overwriting the base system behavior with the recovery strategy for that state. We validate the feasibility and effectiveness of our framework through a prototypical reactive application for tracking mouse movements in different scenarios. Our results demonstrate that with just the definition of monitors, the system is indeed able to recover from failure states without a predefine strategy.**

***Index Terms*—Self-healing systems, Context-oriented Programming, Functional-reactive programming, Reinforcement learning**

I. Introduction

Self-healing systems are of used to enable systems to recover autonomously from failure states, whenever these appear. To enable self-healing in a software system we need to address several challenges [1]. (1) We need to build the means to *monitor* the system state, (2) use *fault analysis* to detect the root cause of faults based on the monitored state, (3) *make a decision* about the detected fault, whether it is blocking, and (4) execute the *recovery* actions to take the system back to a correct state. To execute the aforementioned process, software systems must be equipped with the tools and capabilities for each of the tasks. To capture the system state and to detect faults therein, self-healing systems specify hooks in the system as control points to observe the system state. Such hooks are also used to plug the corrective actions (*e.g.,* alternative execution paths, introductions of new/fixed modules) [1]. Such characteristics of self-healing systems can be problematics, as to diagnose and heal from failures these need to be anticipated by developers at design time. This implies that self-healing systems require a pre-defined set of instructions to recover from prescribe failures. This characteristic accounts for a low flexibility of self-healing systems, only offering a solution of anticipated unknowns. Unknown failures are not covered, leading to system failures still. Additionally, there are concerns regarding the core properties of self-healing systems. For instance, it is not directly evident whether local changes can assure global healing [2].

We note that the application of self-healing has been applied mostly in the field of distributed system, taking advantage of the characteristics of such systems with defined metrics that point to possible failure states during the execution. However, the use of self-healing in other system types has been seldomly studied [3, 4]. In line with this observation, we recognize there is a wide variety of research and application opportunities of self-healing systems. For example, data streaming and big data [5], web services aiming to ensure the quality of service [6], inside operating systems to avoid faults requiring a system restart [7], complex reconfigurable network environments [8].

This paper proposes an alternative to deal with the implicit definition challenges of designing self-healing systems. Our proposal breaks the assumptions previously made for monitoring, fault analysis, and recovery action execution about the pre-definition of healing strategies (Section III). To do this, we first introduce flexible dynamic monitors, that can be defined and modified at run time. Moreover, monitors are flexible to define any type of condition, as predicates, with respect to any system variable, avoiding fix monitoring points. Second, upon detecting failures, we use an algorithm based on Reinforcement Learning (RL) options to learn the corrective atomic actions taken by a human actor, in order to restore the system from the failing state. Once a resolution strategy is learned, using Context-oriented Programming (COP), the system dynamically

adapts the execution path whenever the state leading to the error is detected by the monitor. The adapted behavior corresponds to the learned recovery strategy. Furthermore, our proposal seeks to broaden the application of self-healing systems to a new domain, that of reactive systems [9]. The objective behind this, is to evaluate the feasibility to apply the process of self-healing systems in a domain in which systems are in continuous execution, and in which there is not a predefined set of metrics to assess the system performance across multiple systems.

Therefore, our framework is realized on top of three main concepts to achieve run-time learning and definition of recovery strategies: (1) reactive systems, (2) RL options, and (3) Context-oriented Programming

To validate the feasibility and applicability of our approach we present a proof-of-concept application taken from the reactive systems literature. The application consist of a simple GUI application to react to mouse movement events. We introduce monitors into the application to observe if a given property is observed. The results of our evaluation show that, using our framework, we are able to detect over 70% of all failure states, effectively generating and learning a recovery strategy of each of them.

II. Background

Before going into the details of our proposal, this section sets the definitions of the three main concepts behind our solution: reactive programming, and automated generation of adaptations based on RL and COP.

A. Reactive Systems

There are two main approaches to address the complexity of reactive applications: Event-based languages and languages with direct representation of reactive values.

Reactive values [10] propose as main abstraction values that vary continuously over time. Those values, called *behaviors*, may also depend on other time-varying entities, creating, in this way, a dynamic dependent object. Thus, whenever a *behavior* is updated, all *behaviors* depending on it are also updated.

Note that writing an application using such an approach allows the programmer not to update values explicitly; instead, the values are updated automatically, the language runtime should take care of the complexity of updating values asynchronously.

REScala [11] is presented as a reactive language that provides a robust event system with seamless integration support for reactive values, promoting a mixture of Object-Oriented Programming (OOP) and Functional programming (FP). REScala supports Signals (a concept for expressing functional dependencies among values in a declarative way), and events for continuous or discrete-time changing values. A Signal represents the state in the application, whereas an event holds a value that changes when it is fired, representing actions in the application. REScala exposes a series of abstractions for signals and events, uniformly applying over them.

In order to propagate changes through a reactive system, REScala provides observers which attach a handler function to the event. Every time the event is fired, the handler function is applied to the current value of the event.

The following code snippet show and example of an observer. Line 1 associate the signal *counting* with a handler function. The *counting* is a integer signal that holds the learning steps that the monitor has done. Notice that, the handler function (lines 2-3) changes the value of the learning flag variable when the learning step limit is reach.

```
1 val learningStageObserver = counting observe
2 (count => { if (count > learningStepsLimit)
3     learningFlag = false})
```

Observers are the base for failure detection inside the monitors. In this case, the handle function is the predicate that defines the failure states. Whenever the predicate evaluation is true, the monitor executes the healing strategy.

B. Context-Oriented Programming

Traditional OOP method dispatch depends on message passing and the receiver's action, while COP considers the context of the system to dispatch according to it. COP allows modeling the variability required by complex adaptive systems based on behavioral variations that depend on context [12]. Behavioral variations are defined by *layers*, that are abstractions defining partial method definitions. *layers* can be dynamically activated or deactivated depending on the context currently executing.

Complex real-world self-adaptive systems consider several requirements. For instance, distribution leads to various contexts in different concurrent components. Correspondingly, if several components create different contexts, they may trigger behavioral changes in others. Thus, behavioral variation activation performed by asynchronous communication is desirable. Additionally, considering a highly dynamic environment, performing behavioral changes without leading to inconsistent/erroneous behavior is also a primary concern.

ContextErlang [13], is a COP language that copes with the requirements mentioned above. ContextErlang is based on context-aware agents. These agents have behavioral units that can be dynamically activated on the agents, *variations*. The mechanism to activate such variations is via message passing to the agent. This mechanism leads to asynchronous activation required in real-world applications.

One of the advantages of ContextErlang is that it is presented with a core calculus that specifies its exact

semantics based on the actor concurrency model. The advantage of the language's semantics, is that the language capabilities can be easily replicated in other actor languages, as for example Scala. ContextScala is a COP language implemented on top of the Akka framework in Scala, based on ContextErlang.

The following shows the example of managing mouse movement (the move behavior) using ContextScala. We define two variations of the move behavior to manage the direction of the movement, Up (Lines 1-3) and Down (Lines 4-6). Note form the example, that the definition of variations is defined reusing the main modularity abstraction of Scala, classes.

```
1  class Up extends Variation[Up] {
2     def moved(): Unit = println("Movement Up")
3  }
4  class Down extends Variation[Down] {
5     def moved(): Unit = println("Movement Up")
6  }
```

Variations are made dynamically available in the environment through their activation. In ContextScala this is managed by an instance of the ContextAgent (an Akaa actor), which is in charge of variation activation by means of message passing to the actor. *ContextAgent* deals with context variations. Through the definition of the variation list, as shown in the following code snippet, *e.g.,* Line 1. Upon a method call (the moved method in our example), the implementation corresponding to the activated variation is executed. In our example, Line 2 executes the definition of the Up() variation, and Line 4 Executes the definition of the Down() variation as follows:

```
1  ContextAgent ! SetActiveVariations(List(Up()))
2  ContextAgent ! moved()
3  ContextAgent ! SetActiveVariations(List(Down()))
4  ContextAgent ! moved()
5  Movement Up
6  Movement Down
```

C. Reinforcement Learning Options

RL is a learning technique that learns optimal actions for specific environmental conditions by trial-and-error based on interactions with the environment.

At each time-step, Q-learning agent perceives the environment and maps it to a state s_i from its state space S. It then selects an action a_i from its action set A and executes it. The agent receives a reward r_i from the environment when it transitions to the next state, based on which it updates the suitability of taking action a_i in state s_i. The agent's goal is to learn a policy (*i.e.,* the most suitable action for each state) to maximize the long-term cumulative reward. The learning rate α determines to what extent new experience overwrites previously learned ones, and the discount factor γ determines how much the future rewards are discounted for agents to prioritize immediate actions but still be able to plan the best long term actions.

$$\underbrace{Q(s_t,a_t)}_{\text{New Q-Value}} \leftarrow Q(s_t,a_t) + \underbrace{\alpha}_{\text{Learning rate}} [\underbrace{r_{t+1}}_{\text{Reward}} + \underbrace{\gamma}_{\text{Discount rate}} \overbrace{\max q'(s',a')}^{\text{Maximum Q-Value in the next state}} - Q(s_t,a_t)] \quad (1)$$

Early interactions with the environment are focused on environment exploration, *i.e.,* actions are picked randomly and uniformly from actions available in a given state. At the same time, after an agent has had a chance to learn the quality of actions, later stages are focused on exploitation –that is, mainly executing those actions known to lead to the highest long-term rewards.

Numerous ways to build options from primitive actions exist, from those that are manual [14] or require domain knowledge [15], to learning options [16], identifying sequences that lead to the fulfillment of subgoals [17], or automatically generating all action combinations but using another layer of learning to narrow down the complete set of generated macro-actions to the most helpful ones [18]. Options can also be learned from generated behavior histories by extracting the most commonly used sequences of primitive actions [19]. Option learning in Auto-COP is most closely related to the techniques used in subgoal fulfillment [17] and behavior histories [19].

Temporally extended actions (*e.g.,* macro actions) are used in Artificial Intelligence applications to ensure robustness and build in prior knowledge into applications, from early work on building and executing robot plans [20] to recent applications in deep learning [21]. In particular, in RL, macro-actions are used to speed up learning or to minimize the periods of suboptimal performance during exploratory interaction with the environment. We consider RL-based macro-action techniques, called options [22], to be suitable for learning and integrating sequences of actions into COP-based adaptive systems. Adaptations in COP respond to changes in the context, akin to the way actions in RL are learned and taken in response to observed environment conditions. In this proposal, we use Q-learning inside the monitor to define the learning strategy.

III. Self-Healing in Reactive Systems

This section introduces our proposal for a self-healing framework based on monitors that observe the system's usual behavior, and can recover the system from unannounced run-time failures by learning the appropriate corrective actions on-line.

The critical element to achieving this reactive self-healing framework are monitors. Handling the failure detection, triggering learning, and executing healing actions at run time. Monitors run alongside the system

watching its behavior, and executing healing strategies based on the execution context. Note, however, that monitors do not follow specific predefined healing strategies. Instead, they learn from the natural system behavior executed by experts after an error detection, triggered by the invalidation of predicate statements defined in the monitor. To do this, there must be a predefined set of atomic actions to execute the base behavior of the system. The healing strategy will be learned from combinations of these actions. Notice that having predefined atomic actions is not a specific healing strategy since the monitor is smart enough to compose these actions in different manners based on the context. Additionally, the framework uses the reactive paradigm to handle the event propagation between the base application and the monitor layer.

A. Error Detection

As mentioned before, monitors are a critical element in the framework. This entity is supposed to supervise any part of a (reactive) system. Thus, as many monitors as needed could be instantiated in the application to achieve the optimal self-healing strategy. Fig. 1 shows the monitor's internal structure and its interaction with the system, the internal components of the monitor are: the fault detection system, the learning model, and the variation manager. The fault detection system interacts with the learning model to learn the specific behavior watched by the monitor; once the learning stage is finished, the variation manager generates the corrective action sequence as a variation to be used by the system at run-time, at a later stage.

Note that for this process to work, it is crucial that atomic actions are defined within the application. Monitors will use these atomic actions to learn and define specific healing strategies.

An instance of a monitor requires a specification of a set of relevant variables for the monitor (*i.e.,* observable variables), and a predicate that can be verified using such variables. The monitor verifies behavior based on REScala's Observers; thus, the monitoring behavior is fired with every update in the signal/event associated with it. In addition, monitors' abstraction allows refining the predicates dynamically, if the acceptable conditions of the behavior are to change at run time. This is achieved by adding more complex statements to the original by modifying existing statements.

B. Recovery Strategy Generation

REScala observers drive the learning model inside monitors. Therefore, each time an observer triggers an invalid state event, the learning process starts. The learning process follows the regular RL process with an exploration phase to learn the corrective behavior for an invalid state, and an exploitation phase after convergence is reached, to use the learned actions as a

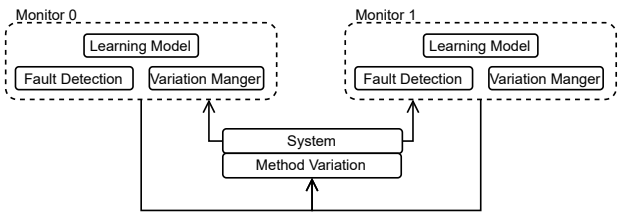


Fig. 1: Monitor Architecture

recovery strategy for the given state. Note, in this process we can have a continuous learning process in which new recovery actions can be learned even after an strategy has been generated already.

In Section II-C we positioned RL RL is presented as a suitable alternative to integrate with COP systems. In the combination of the two strategies to adaptive systems, we use a Q-learning agent to learn the atomic action sequences from monitors. Fig. 2 sketches the learning process. In the figure, colored balls represent events caught by the monitor, each color representing a different events. At a given moment in time, the system state is captured by the monitor and used to evaluate the defined predicates. This process is executed continuously, given the reactive nature of or target systems. Finally, each atomic action must be associated with a scalar reward in accordance to the monitor's predicate.

The first step in the learning process is to define the reward for the fired event; then, this value and the state of the system at the moment of the event are used to update the learning monitor state, using the update rule, defined in Reduction rule (1). To achieve this, the monitor's learning module uses a map in which keys are the system states that the monitor has recorded, and its value is a map containing the list of possible actions to execute associated with the accumulated reward for the action at that particular state. Every event fired adds to the learning step counter.

During exploitation, the monitor generates a map containing the fault states and the learning reaction to those states. Each state's reaction chooses to use the best-rewarded action within the final learning state monitor. That action leads to the next state. If the new state is defined within the learning monitor state, the process is repeated (otherwise, the reaction finishes), and the process finishes when no faulty state is reached. Thus, a map of reactions is generated containing faulty states and the corresponding reaction to get from it to a valid state. This list is then used to build the required variations of the system's behavior at run time, to recover from the detected failure state whenever it occurs again. The precision for these reactions depends entirely on the exploration phase in the learning stage.

Notice that the effectiveness of the healing strategies produced by the monitors depends on the expressive-

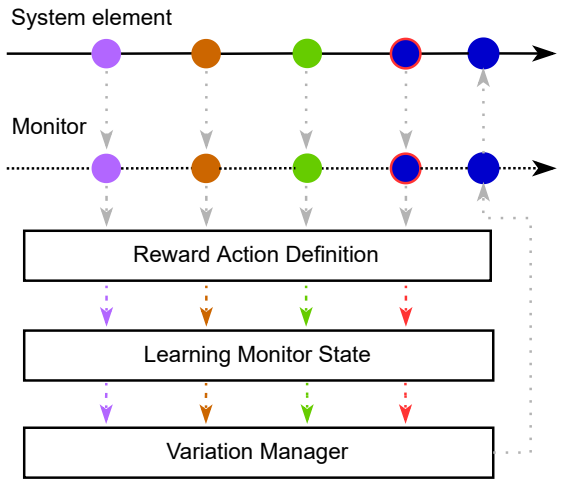


Fig. 2: Monitor learning stages

ness of the predicate and the exploration of the possible states within the learning stage. Hence, if the monitor only learns from local states, the healing strategies would not work in those states that were not reached during the learning. Further, if the expressiveness of the predicate does not allow the monitor to identify the desired faulty states, the learning stage would not take into account those, and the reactions will be incorrect.

C. Healing From Error States

The healing strategy does not rely on predefined actions for particular program. Instead, it depends on nested atomic actions created at run time, based on the learned atomic actions to reach a valid state.

The detection of failure states and the association between such states and atomic actions allows the monitor to know the correct (expected) behavior of the system at any given state. For instance, whenever a state is reached in which the monitor's predicate is violated, the system should automatically execute the learned list of atomic actions to recover from the detected failure.

The healing process is as follows: Whenever an event fires the predicate if the state is defined in the reaction map, the monitor interrupts system's base execution and proceeds to execute the reaction. Once the reaction is finished, the expected behavior control execution is returned to the element. For instance, in Fig. 2 the whole healing process is sketched. The red state represents a faulty state; once the previously mentioned process occurs, the monitor proposes a healing reaction that is executed if it were the expected system behavior.

The reaction is a list of variations in which each variation calls to the next variation. Thus, the execution of the reactions consist of setting the list of variation to the variation manager and then calling the system behavior, which now corresponds to that of the first variation; this will lead to execute all of the variations. Note, that after the execution of the healing strategy the system reverts to the base behavior without reactions, as disposed by COP

IV. Validation

This section shows the feasibility of self-healing applications defined without specific hooks in the program to trigger healing, or preconceived recovery strategies, applied to the new domain of reactive systems. For this purpose we use a proof-of-concept prototypical application for reactive systems, taken from the REScala exemplars.[1] This example consist of a graphical application using the mouse position as a reactive value; the GUI instance detects the mouse movement and displays a point that follows the cursor in a circular pattern.

A. ASCII Mapping Tessellation

On top of the example provide by REScala, we define several tessellations of the GUI surface based on the sum or subtraction of the coordinated values module ASCII. The example consider the mouse position as signal of points (x, y). The mouse signal updates each time the mouse is moved inside the GUI application. Four signals are created based on the mouse signal position: *sum*, *sumMod*, *asccivalue*:

```
val sum: default.Signal[Int] = Signal[Int] {
  (mouseX() + mouseY()) % 255
}
val sumMod: default.Signal[Int] = Signal[Int] {
  if (sum() < 10) sum() + 33 else sum()
}
val asciiValue: default.Signal[Char] = Signal[Char] {
  sumMod().toChar
}
```

The signals create a mapping between the coordinates of the GUI and the printable ASCII characters. Thus, signals *sum* and *sub* maps mouse coordinates into an integer lower than 255, the signal *sumMod* defines values for printable ASCII characters and the signal *asciiValue* is a signal of those characters.

We define a range of ASCII values to be the faulty states. The code below shows two examples of predicates to define two different tessellation patterns. Notice that the predicate expressiveness is based in the user implementation, any of the built-in functionality of REScala could be used to define it. For instance *predicateOne* uses just simple logical operation whereas *predicateTwo* uses functions to know if the value of the signal is prime:

```
val predicateOne =
 (x: Signal[Int]) => Signal[Boolean] {
   (40 < x() and x() < 50) or 200 < x()
 }
val predicateTwo =
 (x: Signal[Int]) => Signal[Boolean] {
   isPrime(x())
 }
```

[1] https://github.com/rescala-lang/REScala/tree/master/Code/Examples

The tessellations generated when using the predicates: *predicateOne*, *predicateTwo*, respectively are display in Fig. 5. For these, the blue region represents the faulty states. Thus, whenever the monitor detects the mouse in the blue region, it should move it to any white region. Additionally, notice that the tessellations are significantly different in how the faulty states are distributed through the states. In Fig. 5b the faulty state are more uniformly distributed than in Fig. 5a where the faulty state are concentrated in three particular regions.

The monitor watches the behavior of the mouse and detects the faulty states using the corresponding predicate and an Akka actor. The system is updated with each mouse interaction. This means that the signals and the monitor are update with each mouse movement. Hence, the detection is driven by an observer attached to the monitor's observable, the mouse position. In Fig. 3 the observer learning process behaves based on the learning stage. If the learning stage is active: the step function is in charging of calculate the reward for the atomic action taken, and update the Q-Values, Line 3. As mention in Section II-C, a set of possible actions must be set. In this case, the actions are the movements made by the mouse: Up, Down, Left, Right. The reward is a binary function assigning 1 when the predicate is true (*i.e.,* the mouse is in a valid position) and 0 when the predicate is false. Once the learning stage is finished, the final Q-Value map is used to generate the reaction, Lines 4-9.

The function inside the monitor in charge of activating the recovery strategy is base on the context, given by the system state, as presented in Fig. 4. At Line 1, the attribute *context* represents the value of the predicate, attribute *e* is the state of the mouse, the position. When the mouse is in a valid position teh base system behavior is trigger (*i.e.,* the regular mouse movement) in contrast when the mouse position invalidates the predicate the corresponding context is activated trigger the reaction as in Lines 2-8.

B. Experimental Design

In our case study, we consider two tessellations to avoid bias in our solution; the scenarios correspond to the tessellations generated when using the predicates: *predicateOne*, *predicateTwo*, respectively. This scenario considers a square GUI of side 100 pixels. Two different experiments were designed to test the monitor failure detection and the monitor failure healing strategies. An external automatic agent is meant to move the mouse uniformly through the whole GUI for both experiments. For each experiment five iteration where done.

The first experiment counts the number of times the monitor detects a failure; later, this will be compared with the failures it was supposed to detect. Once the learning state is finished, the second experiment takes place; the agent moves the mouse on faulty regions to test how many healing strategies generate reactions leading to no faulty regions. Through all the cases, the learning steps are set to 100000.

C. Results

In Table I the result of the experiments are disposed, additionally, the Table II displays the mean and the standard deviation for the results. The description of the information displayed per row in Table I is presented bellow:

Fault detected
: Shows the number of faulty states that the monitor detected through the exploration stage.

Fault proportion
: Displays the proportion of faulty states against the number of steps.

Correct strategies
: From the list of strategies generated after the learning stages this number represents the healing strategies that leads to a no faulty state.

Healing effectiveness
: Exhibits the proportion of correct strategies against the number of states that are supposed to be heal. Notice, the number of states to be heal is a fix number for each tessellation thus for *predicateOne* this number is 1174 and for *predicateTwo* this number is 2090.

The data presented in the first two rows of Table I allows the measurement of the monitor failure detection. Having a square GUI of size 100, the *predicateOne* tessellation has 1174 faulty states and 8826 no faulty states, meaning an objective target ratio of 13.3% whereas the *predicateTwo* has 2090 faulty states and 7910 no faulty states, meaning a target ratio of 26.4%.

Notice that for both experiments, the standard deviation is low, as shown in Table II, showing the precision of the monitor when detecting faulty states. On the other hand, the accuracy of the faulty detection is slightly off compared to the target ratio and the average for the experiments. There is a difference of almost 5% for the *predicateOne* tessellation and almost 6% for the *predicateTwo* tessellation. This puts in evidence some bias on the behavior of the external agent when executing the experiment iteration.

The information displayed in the last two rows of Table I allows the measurement of the monitor failure healing strategies effectiveness. For instance, the mean value shown in Table I for correct strategies in the tessellation of *predicateTwo* is just slightly off the target value. Additionally, the standard deviation is quite low, meaning that the monitor is precise and accurate when generating and executing healing strategies.

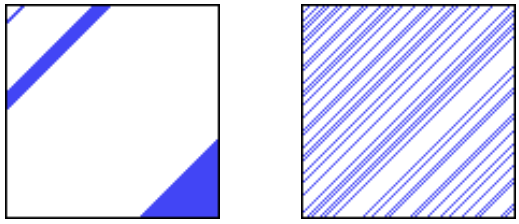
At first glance, the result for the *predicateOne* tessellation seems contradictory to the mentioned results. For example, iteration 3 shows healing effectiveness of barely 23%. The mean value for this measure is just

```
val DetectionObservable = observable observe { State =>
 if (learningFlag) {
      step(State.oldPosition, State.newPosition)
 } else if (tableVariationCreationFlag) {
   agentData.Q.foreach {
     case (key, value) =>
     StateOfVariations.updateState(key, updateCurrentVariationState(key, value, List()))
   }
   tableVariationCreationFlag = false
 }
}
```

Fig. 3: Monitor learning process subsystem

```
def takeAction(contex: Boolean, e: MouseMoved): Unit = if (contex) {
 val currentAgent = AgentLocation(e.point.x, e.point.y)
 val currentVariationList: List[QMove] = StateOfVariations.getState.getOrElse(currentAgent, List())
 if (currentVariationList.nonEmpty) {
   val castedVariationList = CastingDefinition.QMoveListToDefinitionList(currentVariationList)
   actor ! SetActiveVariations(castedVariationList)
   actor ! moved(e.point.x, e.point.y)
 }
} else {
 currentObservable.mouse.mouseMovedE.fire(e.point)
}
```

Fig. 4: Monitor reaction activation subsystem



(a) Tessellation induced by *predicateOne* in which are three specific faulty regions

(b) Tessellation induced by *predicateTwo* in which faulty regions are spread through the GUI

Fig. 5: Areas in which errors arise, in blue, for the two scenarios

about 55.7%, still with a low standard deviation. As noticed in Fig. 5, both tessellations cover the GUI surface with a different distribution of faulty states. Notice that the *predicateOne* tessellation has three specific regions containing faulty states making the generation of healing strategies quite challenging. In contrast, the tessellation induces by *predicateTwo* allows a better propagation of the reward due to the many no faulty regions surrounding the faulty regions. This is not a limitation of the monitor learning subsystem; instead, it shows that the learning steps should be defined by the user base on the complexity of the system and its interaction with the faulty states observed by the monitors.

V. Related Work

This sections makes a distinction between the self healing related approaches and the existing COP solutions using RL.

A. Self Healing

Early attempts to define a self-healing system have been made. For instance, a self-stabilizing system where the system reaches a legitimate state in a finite number of steps regardless of its initial state, [2]. Following similar definitions, several approaches have been proposed to define self-healing systems. For example, a classification by area of research is presented in Psaier *et al.* [3]. Here the authors identify various implementations of self-healing systems according to their application domain: embedded systems, operating systems, reflective middleware, and web services, among others.

Nevertheless, they all share critical general concepts, whatever the field. Koopman [23] presents a set of general concepts that all self-healing systems have in common, regardless of their application domain. In this sense, the system must have a fault model with complete knowledge of the faults expected to self-heal, having some information regarding fault duration, fault manifestation, and fault source. Besides, the system must be complex enough to respond to the fault, meaning that the proper mechanism for fault detection, response, and recovery must be considered. Furthermore, self-healing approaches must evaluate the system completeness. For example, it must account for how the system behaves over time and handle the various changes in the system architecture that may occur at run time and architectural completeness. Finally, Koopman [23] proposes that the design context shapes in a particular way the self-healing capabilities.

	PredicateOne					PredicateTwo				
	1	**2**	**3**	**4**	**5**	**1**	**2**	**3**	**4**	**5**
Faults detected	18122	18390	18127	18404	18048	20786	20652	20594	20712	20805
Faults proportion	18.1%	18.3%	18.1%	18.4%	18.0%	20.7%	20.6%	20.4%	20.7%	20.8%
Correct strategies	649	797	309	686	843	1937	1786	1677	1808	1842
Healing effectiveness	55.%	67.8%	23.6%	58.4%	71.8%	92.6%	85.4%	80.2%	86.5%	88.1%

Table I: Experimental iterations results

	PredicateOne		PredicateTwo	
	μ	σ	μ	σ
Faults detected	18218.1	148.71	20709.8	82.4
Correct strategies	656.8	187.26	1810	84.16

Table II: Statistical measures of the experiments results

Concerning the architectural completeness, the approach presented by Dashofy *et al.* [24] uses software architecture. In this respect, the authors state that before any self-healing system could occur, a significant infrastructure must be put in place to support fault detection and repair. Specifically, the system must be built using a framework that provides: run-time adaptation, a language to express the repair plan, and an agent to execute the repair.

Usually, self-healing systems are tightly integrated with the application. Granting the systems the ability to deal with failures at detection. Nonetheless, externalization mechanisms are proposed [25], where the self-healing system is untangled from the base application using an architectural model approach to design the monitoring, problem detection, and repair; i.e., the self-healing system works on top of the base application understanding what the running system is doing in high-level terms. Following this idea, Al-Zawi *et al.* [26] proposes that these systems are suitable for continuous learning. The system could learn from its run-time behavior to improve its capabilities. Specifically, the case study of this paper presents a client-server application in which the self-healing system is based on a PRNN (Pipelined Recurrent Neural Network) where the parameters are the server status and the bandwidth level. Another example of learning capacity for self-healing systems [27] uses a Multivariate Diagram and a Naive Bayes Classifier to determine severity levels and infer possible consequences. In this approach, the model continuously changes its parameters based on the healing process. Reinforcement Learning RL has also been used for this application. For instance, Schneider *et al.* [4] shows self-healing systems' capabilities in the area of LTE. This is implemented using a RL scheme. In particular, the actor-critic approach was chosen for its capability of learning from experience and its low computational complexity. Similarly, Razavi *et al.* [28] proposes a self-optimization solution of coverage and capacity in LTE networks using fuzzy RL while operating complete autonomous in a fully distributed environment. In addition, the authors show that the solution has both self-configure and self-healing capabilities. Additional research efforts exist to define self-healing systems over networks [29–31], using similar techniques to the ones described before.

B. COP + Learining

Our self-healing approach incorporates the Q-learning reinforcement learning approach, enabling the system to define healing decisions automatically. Then, when the system detects a failure state, this healing strategy is executed automatically using context-aware variations of the defined methods.

Similarly, Cardozo *et al.* [32] proposes not to have predefined adaptation using a proof-of-concept to illustrate this idea while showing the challenges of implementing such systems. For instance, the systems should be capable of integrating new elements, data sources, atomic actions, and goals at run time. Correspondingly, Cardozo *et al.* [33] proposes Auto-COP, which uses RL to build action sequences based on previous instances of the system execution.

As shown in Section II-B COP is used to generate dynamic behavioral responses to the system execution context; nevertheless, due to different simultaneous sensed situations, many adaptations could be applicable, which leads to conflict in systems' execution. An automated conflict resolution mechanism [34], where W-Learning (an RL algorithm) is used to capture the relationships between simultaneously proposed adaptations over time, updating their appropriateness as the system progresses.

Based on the revision of the current state of the art, it is possible to conclude that:

- All self-healing systems must present the same general properties.
- Learning exists to try to manage self-healing systems, but there is no consensus, though RL looks promising.
- The application domain seems to be restricted for distributed systems.

We found that there is something missing in state-of-the-art, which is having self-healing systems with no previous knowledge of the possible errors in centralized systems. Finally, to the authors' best knowledge, there is no work done on self-healing systems on centralized

systems or reactive applications, nor is there any work that has been implemented with REScala.[2]

VI. Conclusion and Future Work

There are two principal concerns when designing a self-healing system: detection and healing. Some challenges are the definition of faulty states, the definition of hooks for adaptation, and the definition of healing strategies. This paper proposes a framework to deal with the complexity of real word self-healing reactive applications. The framework proposes a *monitor* which are abstractions capable of dealing whit the challenge presented above, encapsulating the self-healing complexity from detection to healing. The abstraction brings customizations and modularity to the design of self-healing applications allowing its implementation alongside the system. *Monitors* run alongside any element inside the application in order to achieve the best self-healing performance.

The key features of this proposal are (1) the whole self-healing process is delegated to the *monitors* from failure detection to healing (2) the monitor does not require any predefined healing strategies; instead, through learning from the system's behavior it is capable of generating its healing strategies base on atomic actions Using a prototypical reactive application for tracking mouse movements, we demonstrate the effectiveness of the monitors for learning healing strategies executing when faulty states are detected.

The decrease in complexity and the difficulty of detecting and handling failures in massive reactive systems, like streaming companies, makes this proposal appealing since monitors could be placed inside any existing services, thus releasing the developers from the tedious process of defining complex healing strategies. We demonstrate the feasibility of our approach with a prototypical reactive application for tracking mouse movements in which several signals are generated using the mouse movement information. Further, two predicates define failure states that induce tesselations inside the GUI in which the mouse is moving. In this application, monitors' healing strategies must bring back the mouse to a no-fault region whenever the monitors' detection system detects a failure region. The atomic actions correspond to atomic mouse movements: Up, Down, Right, Left. We show that monitors are a feasible alternative to deal with the detection and healing inside reactive applications through several scenarios.

In future work, we propose the definition of communication strategies between monitors to improve the global healing strategies using the local strategies without the need for any centralized orchestration. Additionally, the design of the distributed reactive monitors also looks promising.

[2]https://www.rescala-lang.com/

References

[1] G. D. Rodosek, K. Geihs, H. Schmeck, *et al.*, *Self-healing systems: Foundations and challenges.*, Self-Healing and Self-Adaptive Systems, 2009.

[2] E. W. Dijkstra, *Self-stabilizing systems in spite of distributed control*, Commun. ACM, vol. 17, pp. 643–644, 1974.

[3] H. Psaier and S. Dustdar, *A survey on self-healing systems: Approaches and systems*, Computing, vol. 91, pp. 43–73, 2010.

[4] C. Schneider, A. Barker, and S. Dobson, *A survey of self-healing systems frameworks*, Software: Practice and Experience, vol. 45, pp. 1375–1398, 2015.

[5] B. Dundar, M. Astekin, and M. S. Aktas, *A big data processing framework for self-healing internet of things applications*, Intl. Conf. on Semantics, Knowledge and Grids, IEEE, 2016, pp. 62–68.

[6] H. Naccache and G. C. Gannod, *A self-healing framework for web services*, IEEE Intl. Conf. on Web Services (ICWS 2007), IEEE, 2007, pp. 398–345.

[7] F. M. David and R. H. Campbell, *Building a self-healing operating system*, Third IEEE Intl. Symp. on Dependable, Autonomic and Secure Computing (DASC 2007), IEEE, 2007, pp. 3–10.

[8] A. Trehan, *Algorithms for self-healing networks*, CoRR, vol. abs/1305.4675, 2013.

[9] Z. Wan and P. Hudak, *Functional reactive programming from first principles*, Proc. of the Conf. on Programming language design and implementation, 2000, pp. 242–252.

[10] C. Elliott and P. Hudak, *Functional reactive animation*, Intl. Conf. on Functional Programming, 1997.

[11] G. Salvaneschi, G. Hintz, and M. Mezini, *Rescala: Bridging between object-oriented and functional style in reactive applications*, Proc. of the 13th Intl. Conf. on Modularity, 2014, pp. 25–36.

[12] R. Hirschfeld, P. Costanza, and O. Nierstrasz, *Context-oriented programming*, Jour. of Object technology, vol. 7, pp. 125–151, 2008.

[13] G. Salvaneschi, C. Ghezzi, and M. Pradella, *Contexterlang: A language for distributed context-aware self-adaptive applications*, Science of Computer Programming, vol. 102, pp. 20–43, 2015.

[14] S. Elfwing, E. Uchibe, K. Doya, *et al.*, *Multi-agent reinforcement learning: Using macro actions to learn a mating task*, 2004 IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566), IEEE, 2004, pp. 3164–3169.

[15] A. McGovern and R. S. Sutton, *Macro-actions in reinforcement learning: An empirical analysis*, Computer Science Department Faculty Publication Series, p. 15, 1998.

[16] J. Randlov, *Learning macro-actions in reinforcement learning*, Advances in Neural Information Processing Systems, vol. 11, 1998.

[17] M. Stolle and D. Precup, *Learning options in reinforcement learning*, Intl. Symp. on abstraction, reformulation, and approximation, Springer, 2002, pp. 212–223.

[18] M. Pickett and A. G. Barto, *Policyblocks: An algorithm for creating useful macro-actions in reinforcement learning*, ICML, 2002, pp. 506–513.

[19] S. Girgin and F. Polat, *Option discovery in reinforcement learning using frequent common subsequences of actions*, Intl. Conf. on Computational Intelligence for Modelling, Control and Automation and Intl. Conf. on Intelligent Agents, Web Technologies and Internet Commerce (CIMCA-IAWTIC'06), IEEE, 2005, pp. 371–376.

[20] R. E. Fikes, P. E. Hart, and N. J. Nilsson, *Learning and executing generalized robot plans*, Readings in Knowledge Acquisition and Learning: Automating the Construction and Improvement of Expert Systems. Morgan Kaufmann Publishers Inc., 1993, pp. 485–503, ISBN: 1558601635.

[21] S. Zhang and S. Whiteson, *DAC: the double actor-critic architecture for learning options*, CoRR, vol. abs/1904.12691, 2019.

[22] R. S. Sutton, D. Precup, and S. P. Singh, *Intra-option learning about temporally abstract actions.*, ICML, 1998, pp. 556–564.

[23] P. Koopman, *Elements of the self-healing system problem space*, 2003.

[24] E. M. Dashofy, A. Van der Hoek, and R. N. Taylor, *Towards architecture-based self-healing systems*, Proc. of the first workshop on Self-healing systems, 2002, pp. 21–26.

[25] D. Garlan and B. Schmerl, *Model-based adaptation for self-healing systems*, Proc. of the First Workshop on Self-Healing Systems, Association for Computing Machinery, 2002, pp. 27–32, ISBN: 1581136099.

[26] M. M. Al-Zawi, A. Hussain, D. Al-Jumeily, *et al.*, *Using adaptive neural networks in self-healing systems*, 2009 Second Intl. Conf. on Developments in eSystems Engineering, 2009, pp. 227–232.

[27] Y. Dai, Y. Xiang, and G. Zhang, *Self-healing and hybrid diagnosis in cloud computing*, Cloud Computing, Springer Berlin Heidelberg, 2009, pp. 45–56, ISBN: 978-3-642-10665-1.

[28] R. Razavi, S. Klein, and H. Claussen, *Self-optimization of capacity and coverage in lte networks using a fuzzy reinforcement learning approach*, IEEE Intl. Symp. on Personal, Indoor and Mobile Radio Communications, 2010, pp. 1865–1870.

[29] A. Saeed, O. G. Aliu, and M. A. Imran, *Controlling self healing cellular networks using fuzzy logic*, IEEE Wireless Communications and Networking Conf., 2012, pp. 3080–3084.

[30] T. Angskun, G. Fagg, G. Bosilca, *et al.*, *Self-healing network for scalable fault-tolerant runtime environments*, Future Generation Computer Systems, vol. 26, pp. 479–485, 2010.

[31] R. Kawamura, K.-I. Sato, and I. Tokizawa, *Self-healing atm networks based on virtual path concept*, IEEE Jour. on Selected Areas in Communications, vol. 12, pp. 120–127, 1994.

[32] N. Cardozo and I. Dusparic, *Generating software adaptations using machine learning*, 2018.

[33] ——, *Auto-cop: Adaptation generation in context-oriented programming using reinforcement learning options*, CoRR, vol. abs/2103.06757, 2021.

[34] N. Cardozo, I. Dusparic, and J. H. Castro, *Peace corp: Learning to solve conflicts between contexts*, Proc. of the 9th Intl. Workshop on Context-Oriented Programming, 2017, pp. 1–6.