Autonomic Reactive Systems via Online Learning

Sanjit A. Seshia
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
E-mail: sseshia@eecs.berkeley.edu

***Abstract*— Reactive systems are those that maintain an ongoing interaction with their environment at a speed dictated by the latter. Examples of such systems include web servers, network routers, sensor nodes, and autonomous robots. While we increasingly rely on the correct operation of these systems, it is becoming ever harder to deploy them bug-free.**

We propose a new formal framework for automatically recovering a class of reactive systems from run-time failures. This class of systems comprises those whose executions can be divided into rounds such that each round performs a new unit of work. We show how the system recovery and repair problem can be modeled as an instance of an online learning problem. On the theoretical side, we give a strategy that is near-optimal, and state and prove bounds on its performance. On the practical side, we demonstrate the effectiveness of our approach through the case study of a buggy network monitor. Our results indicate that online learning provides a useful basis for constructing autonomic reactive systems.

I. Introduction

Reactive systems are those that maintain an ongoing interaction with their environment at a speed dictated by the latter. Examples of such systems include many kinds of networked and embedded systems such as web servers, network routers, sensor nodes, and autonomous robots.

Even as we increasingly rely on the correct operation of such systems, it is becoming ever harder to make them entirely bug-free before deployment. There are three major challenges. First, traditional offline verification and testing tools are unable to keep up with growing system complexity. Second, the environments in which embedded and networked systems operate is often unpredictable and it is hard to model them precisely for verification and testing. Finally, with the advent of software-as-a-service and increasing use of reconfigurable hardware, both software and hardware systems change much faster than before, so that a component that worked earlier might no longer do so after an update.

There is therefore a need to construct *autonomic reactive systems* that can (1) monitor themselves online and detect failures arising from design and program faults (bugs), (2) diagnose and recover from such failures, as well as (3) learn from failures so as to repair themselves over a period of time. This paper is mainly concerned with the third item. Specifically, *we propose and demonstrate a formal, algorithmic approach to constructing autonomic reactive systems*. This work complements a growing body of work on the first two items, surveyed below. While our approach is generally applicable, it is particularly suited to a class of systems whose executions can be divided into *rounds* such that each round performs a new unit of work. We make the following novel contributions:

- *Online learning as a basis for self-repair:* The area of *online learning* studies problems of repeatedly making choices so as to optimize the average cost of those choices in the long run. We show how the problem of repairing a system can be formulated as an instance of the *multi-armed bandit problem* [1], [2], a classic online learning problem. We give a learning strategy customized to our problem context, and prove that this strategy is near-optimal.
- *Exploiting parallelism for pro-actively exploring repair space:* With the advent of multi-core processors, there is likely to be far more parallelism available on-chip than most current software applications can exploit. Some of this parallelism can be used for self-repair. We give a framework for leveraging available parallelism for pro-actively exploring the space of options to repair a fault. Currently, the search for a suitable repair is only performed after the failure occurs, typically manually. This process is far too slow for online self-repair. Instead, we propose a duplicate version of the system be run alongside the main copy. This duplicate version morphs itself in each round so as to explore the consequences of a particular "fix" to the code, even when no failure has occurred. The data thus collected can be used to guide the online learner in selecting a good repair action when a failure is eventually encountered.
- *Fault models as a basis for defining the repair space:* The space of possible variants of a system obtained by self-morphing can be huge; in some cases, it can be exponential in the size of the system description. We restrict and define the space of system variants by using a fault model. While such an approach reduces the space of possible repairs, it also makes it more likely that the bug cannot be fixed within this space. Diagnostic information about the cause of a failure that might be used to guide system repair can also be imprecise. We therefore use a performance metric called *regret* that rates a repair strategy in terms of how well it performs with respect to the *best repair in hindsight*. This metric allows us to give theoretical guarantees about our repair strategy even when a perfect fix does not exist.
- *Demonstration on case study of network monitor:* We demonstrate the practicality of our theoretical results by

applying it to a simplified network monitor described in a recent book on Network Algorithmics [3]. Experimental data indicates that the repair strategy we give based on online learning converges to the correct repair while suffering far fewer failures on average than picking a fix at random.

While the presented approach is mainly focussed on system self-repair "in the field," we note that it could be applied offline as well. For example, it could be used as a "repair assistant" to suggest potential bug fixes to programmers for a system under test.

Related Work. Several projects have recently addressed the problem of surviving failures arising from software bugs. Rinard et al. have introduced *failure-oblivious computing* [4], which seeks to execute through buffer overflow problems by returning artificial values for reads that are out of bound. Although potentially unsafe, they demonstrate the utility of being able to execute through an error rather than terminating with an exceptional condition reporting the error. The *reactive immune system* [5] similarly proposed to execute through errors by returning a speculative error code on failure. Qin et al. [6] present Rx, a system for recovering from common bugs in commodity software, such as web servers, such as buffer overruns, races, and memory corruption. The main idea is to change the controllable part of the environment, namely the operating system, by providing safe versions of system and library calls for use during rollback and recovery. However, these approaches do not generalize beyond the considered bug classes and there is no use of online machine learning.

Researchers have successfully demonstrated the use of practical machine learning and statistical monitoring to detect and diagnose system failures (e.g., [7], [8]). Our work complements these efforts by starting where they finish, viz., by performing online system repair.

Easwaran et al. [9] discuss *steering*, a technique for predicting failures and taking evasive action in advance, in the context of discrete event systems. The authors build upon previous work on run-time verification [10], [11] and instead focus on constructing a steerer whose lookahead is more than enough to compensate for communication latency between the steerer and the system. Unlike our work, there is no use of online learning, and the results are restricted to finite-state systems.

Model-based methods involve the synthesis of fault diagnosis and recovery components from high-level descriptions [12], [13]. An example of a model-based system is Livingstone developed by Williams and Nayak [14]. Their work primarily focuses on device faults (such as in sensors and actuators), as opposed to the design/program errors that are the subject of this paper. Demsky et al. [15], [16] present the concept of *data-structure repair*, which uses a model-based approach to maintaining invariant properties of data structures at run-time in the face of errors introduced by buggy code. Joshi et al. [17] give a model-based approach to recovering distributed systems from failures based on partially observable Markov decision processes.

Several projects have explored ways to design computers to recover quickly from transient run-time failures, mainly targeted towards Internet services; an example being the work on reboot-only systems [18], Our work is more theoretical and focussed on deterministic errors.

II. Framework

This section defines the terminology and notation used in this paper, and gives an overview of our approach.

A. Terminology

We use the taxonomy for dependable and secure computing given by Avizienis et al [19].

A *system* is an entity that interacts with other entities (systems), including hardware, software, human beings, and the physical world. These other systems form its *environment*.

The focus of this paper is on making systems robust from bugs that a designer/programmer might introduce into the system description, especially those bugs that are deterministic in nature whose effects cannot be eliminated just by rollback and replay. In fact, the deterministic nature of such bugs makes it likely that systems running identical code will fail in the same (correlated) way, and those correlated failures could even be triggered by a malicious party.

A *failure* occurs in a system when its behavior deviates from the specification. The part of the system state that exhibits this deviation is the *error*. The adjudged or hypothesized cause of an error is called a *fault*.

The system $\mathcal{M}$ is formally modeled as a transition system, represented as a tuple $(\mathcal{S}, \mathcal{A}, \delta, I)$, where

- $\mathcal{S}$ is the set of system states;
- $\mathcal{A}$ is the set of system actions;
- $\delta \subseteq \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$ is the transition function that describes the next state of the system that results from performing an action in the current state; and
- I is the set of initial states of the system.

The above system model is standard. Formally, an action is a predicate over two states (current and next). Examples of actions include system calls, message sends and receives, incrementing a hardware counter, setting a bit, etc.

The only assumption we make about the system $\mathcal{M}$ is that it obeys a *state-renewal* property. Specifically, after startup and initialization, the system *behavior* (also termed as *run* or *execution*) can be divided into *rounds*, each of which begins in a "valid" start state and is of finite, but arbitrary, length. A valid start state is defined by a state invariant I_{start}. This is typical of many reactive systems, whose runs are infinite, but are composed of terminating sub-computations performed within a non-terminating sense-compute-respond loop. One ways to view this assumption is that $\mathcal{M}$ always generates *some* output, even if that is not the *correct* output.

Formally, each minimally correct behavior (trace) of the system must be a sequence of states and actions taking the form

$$s_0\ a_1\ s_1\ a_2 \ldots a_{i_1}\ s_{i_1}\ a_{i_1+1}\ s_{i_1+1} \ldots a_{i_2}\ s_{i_2}\ a_{i_2+1}\ s_{i_2+1} \ldots \ldots$$

where

- $s_0 \in I$;
- $s_i = \delta(s_{i-1}, a_i)$;
- $\forall j \in 1, 2, 3, \ldots, s_{i_j} \in I_{start}$.

The finite sub-trace starting at s_{i_t} and ending before $s_{i_{t+1}}$ is termed as *round* t. We make a distinction here between the initial state of the system (s_0) and the state in which the system starts its state-renewal behavior (s_{i_1}) to model system "boot-up."

The reason we require the state-renewal behavior is so that errors do not accumulate to the extent that the system becomes unrecoverable. By requiring the system to return to a valid start state in each round, we place a minimal correctness requirement on each execution. We note that each round can be viewed as performing a "unit of work." An example of a system exhibiting this behavior is a network packet processing system performing a task such as a packet forwarding, where a new packet is processed in each round of execution. Similarly, reactive sensor-actuator control programs operate in a sense-compute-respond loop, the start of a new iteration corresponds to the program returning to the head of the loop, so I_{start} is simply a predicate on the program counter.

Note that even systems that do not exhibit the above behavior could potentially be viewed in this way. Each round could be a finite, but arbitrary-length prefix of a run of the system that prematurely ends in failure, with system reboot leading to the start of a new round. We can potentially model micro-reboot based approaches [18] in this manner.

Note that we are making no assumptions about the environment, including about the distribution of inputs it provides or as to its adversarial nature.

B. Overview of Approach

Figure 1 depicts the form of the self-repairing system in our approach. There are three main components.

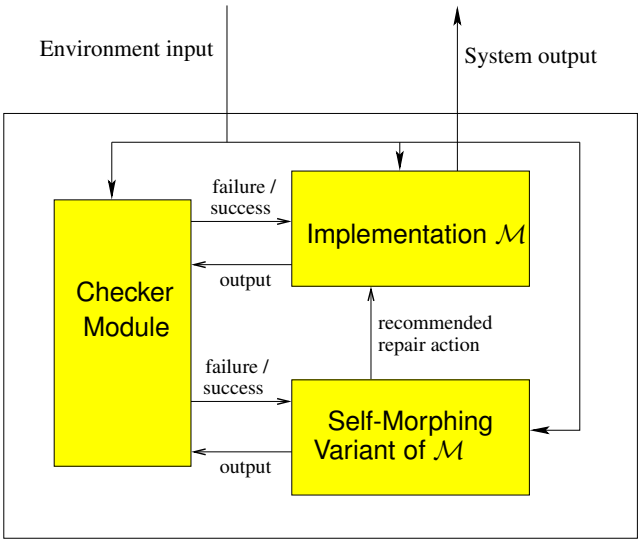


Fig. 1. **Components of a self-repairing system based on online learning**

The implementation module $\mathcal{M}$ is the original version of the system that is to be deployed and/or tested. It receives input from the environment and generates output.

The checker module verifies that the output generated by $\mathcal{M}$ is correct. The checker module can be a simpler (unoptimized), but reliable version of the implementation that is either formally verified to be correct or otherwise assumed to be the specification; in this case, it may be unable to keep up with $\mathcal{M}$ but can randomly pick the rounds in which it checks $\mathcal{M}$'s output. The checker can alternatively be a collection of run-time monitors that check continuously that system invariants and assertions are satisfied. The status of the check (either "failure" or "success") is returned to $\mathcal{M}$ so that it may take corrective action. The vast body of literature on run-time verification [10] can be leveraged in the design of the checker module.

The third component is a variant of $\mathcal{M}$ that is instrumented so as to modify its transition function δ in accordance with a *fault model*. In each round t, this variant, denoted $\overline{\mathcal{M}}$, also runs on the environment input. However, its output is generated solely for verification by the checker module. Based on the failure/success status returned by the checker module, $\overline{\mathcal{M}}$ decides whether and how to modify its transition function for the next round $t+1$. The algorithm that $\overline{\mathcal{M}}$ uses to make this decision is termed as a *repair strategy*.

The repair strategy must pick a *repair action*, i.e., a fix to $\mathcal{M}$, from a space of possible repair actions of size m. This space is defined by a fault model. For example, one possible fault model is that a single variable is initialized to the wrong constant from amongst a set of m constants. With programmer assistance in writing "patterns" of repairs, or by use of static analysis tools, the space of actions could be narrowed down to a tractable size. We will assume in this paper that the value of m is small (polynomial in the system description); techniques to ensure this are left to future work.

A *correct* or *perfect* repair is one which, when applied to the system, ensures that the system will satisfy its specification for all inputs. If the fault model is imprecise, it is likely that a correct repair will not exist. Therefore, we define our measure of performance as a relative measure that compares against the number of failures suffered if the best possible fix under the assumed fault model had been initially applied to $\mathcal{M}$. This relative measure is called *regret*. We elaborate on this point in the next section.

If $\mathcal{M}$ suffers failure, it repairs its code by loading an update recommended by $\overline{\mathcal{M}}$ according to the *repair strategy* of the latter. The advantage is that $\mathcal{M}$ can self-patch itself based on the "experience" gained by $\overline{\mathcal{M}}$ over many rounds of execution, without having to wait a long time for a human supervisor to intervene. However, this approach is only as effective as the repair strategy employed by $\overline{\mathcal{M}}$.

As a sidenote, we observe that the approach can also be generalized to a setting where $\overline{\mathcal{M}}$ and the checker are running at a server physically separate from $\mathcal{M}$, and possibly in an offline diagnosis and repair mode. All the results we discuss in the subsequent sections also carry over to this setting.

III. Online Learning: Formulation and Theoretical Results

In this section, we consider how the online repair process can be optimized by use of *online learning*. In particular, one can view the repair problem as one of *learning from mistakes*. We leverage existing results in computational learning theory (e.g., [2], [20]). System rounds correspond to the "trials" in which learning proceeds.

A. The Multi-Armed Bandit Problem

The problem we face is to pick a repair action based on the history of how candidate repair actions have performed on past inputs received from the environment. This problem maps naturally to an online learning problem referred to in the literature as the *multi-armed bandit problem*. In the bandit problem, originally proposed by Robbins [1], a gambler plays on a set of m slot machines over several rounds, choosing in each round a specific slot machine to play on. At the start of each round, he chooses a machine (a "one-armed bandit") and pulls its arm. He receives a reward at the end of the round. The gambler's aim is to maximize the sum of the rewards he receives over a sequence of rounds. Rewards can be non-positive, thus modeling costs.

The mapping to our situation is easily seen. The gambler corresponds to the the module $\overline{\mathcal{M}}$, and the slot machines correspond to repair actions. If the chosen action correctly avoids the error in that round, it receives a reward of 1, otherwise 0.

In our setting, the rewards depend largely on the inputs selected by the environment. In the basic bandit problem, the rewards for the arms are assumed to be drawn independently from a fixed, but unknown, distribution. However, as Auer et al. [2] argue, in many computer systems settings an alternative formulation is desirable. They term this formulation as the *adversarial bandit problem*, which is essentially the same as the problem stated above except that *no* statistical assumptions are made about the rewards. It is only assumed that each slot machine is initially assigned an arbitrary and unknown sequence of rewards, one at each time step. Thus, we use the adversarial bandit problem formulation given by Auer et al. [2] as the starting point for devising a learning-based repair strategy.

Our measure of performance is relative to the best repair action that could have been chosen from amongst the m possibilities and applied to $\mathcal{M}$ initially. This measure is termed as *regret*, and is formally defined as follows:

Definition 1 *The* regret *of a repair strategy σ is the difference between the expected number of failures of a system that uses σ and the number of failures suffered when using, from the start, the best repair action (receiving highest reward, in hindsight).*

Note that the expectation in the above definition is taken over the random choices made by the repair strategy, not by the environment in choosing its inputs.

The bandit problem illustrates well the trade-off between *exploitation* and *exploration*. On the one hand, the gambler needs to try out all the arms enough (exploration) in order to discover whether one of them can get a high cumulative reward. On the other hand, too much exploration will not allow the gambler to exploit the best arm often enough.

In our context, we leverage parallelism and use $\overline{\mathcal{M}}$ to proactively perform exploration of the repair space before $\mathcal{M}$ actually encounters a failure. In other words, $\mathcal{M}$ gets to exploit the results of exploration performed by $\overline{\mathcal{M}}$.

B. Repair Strategies

Suppose the system is run for T rounds of execution. In the worst case, if there exists an input that causes the system $\mathcal{M}$ to fail, an adversarial environment could always pick that input to feed to $\mathcal{M}$. In other words, $\mathcal{M}$ can end up suffering T failures.

We now present two strategies that achieves $o(T)$ regret when $m \ln m = o(T)$. That is, if the number of actions m is small enough that the quantity $m \ln m$ is asymptotically strictly less than the number of rounds T, sub-linear regret can be achieved. The first strategy, termed Exp3, was given by Auer et al. [2]. The second, termed CE3, is a variant of Exp3 that uses a cost-based model rather than a reward-based one. As we explain later in this section, CE3 is more suitable for practical implementation, and is a contribution of this paper.

In the sequel, we refer to repair actions simply as "actions," and identify them by their indices, which range from 1 to m.

The Exp3 Strategy. Auer et al. [2] present the Exp3 strategy (which stands for "exponential-weight algorithm for exploration and exploitation"). A core idea is to use weights to track the performance of actions over a sequence of rounds. A larger relative weight for an action indicates a history of better performance for that action.

A description of this strategy is given in Figure 2(a). At the start of each round t, Exp3 draws an action i_t according to a probability distribution $p_1(t), p_2(t), \ldots, p_m(t)$ over the actions. This distribution is a mixture of the uniform distribution and one that assigns to each action a probability mass based on the weight associated with that action. The mixture is done based on a parameter $\gamma \in (0, 1]$. Intuitively, the use of the weight-based probability mass is in order to facilitate exploitation, while the uniform distribution is mixed in so as to ensure exploration of the action space.

If the chosen action i_t causes the system to suffer a failure, its weight (and those of all the other actions) remains the same for the next round $t+1$. However, if the system works correctly with action i_t, the weights are increased by an exponential factor that reflects the estimated reward. Thus, in this case, the weights can only increase, never decreasing below 1. Note further that if, after a point, the chosen action does extremely well (the algorithm settling into exploitation mode), its weight will increase to $+\infty$.

Auer et al. [2] prove that the Exp3 strategy can achieve a $o(T)$ regret if $m \ln m = o(T)$, as stated in the following

Strategy Exp3

Parameter: $\gamma \in (0, 1]$
Initialization: $w_i(1) = 1$, for all $i = 1, 2, \ldots, m$.

At each round: $t = 1, 2, 3, \ldots$:

1) For each $i = 1, 2, \ldots, m$, let
$$p_i(t) = (1-\gamma)\frac{w_i(t)}{\sum_{j=1}^{m} w_j(t)} + \frac{\gamma}{m}$$
2) Draw action i_t from the set $\{1, 2, \ldots, m\}$ randomly according to the probabilities $p_1(t), p_2(t), \ldots, p_m(t)$.
3) Receive reward R_{i_t} given by
$$R_{i_t} = \begin{cases} 0 & \text{if action } i_t \text{ fails,} \\ 1 & \text{otherwise.} \end{cases}$$
4) For $j = 1, 2, \ldots, m$, set
$$\hat{R}_j(t) = \begin{cases} R_j(t)/p_j(t) & \text{if } j = i_t, \\ 0 & \text{otherwise.} \end{cases}$$
$$w_j(t+1) = w_j(t) \cdot \exp\left(\frac{\gamma}{m}\hat{R}_j(t)\right)$$

(a) **Exp3**

Strategy CE3

Parameter: $\gamma \in (0, 1)$
Initialization: $w_i(1) = 1$, for all $i = 1, 2, \ldots, m$.

At each round: $t = 1, 2, 3, \ldots$:

1) For each $i = 1, 2, \ldots, m$, let
$$p_i(t) = (1-\gamma)\frac{w_i(t)}{\sum_{j=1}^{m} w_j(t)} + \frac{\gamma}{m}$$
2) Draw action i_t from the set $\{1, 2, \ldots, m\}$ randomly according to the probabilities $p_1(t), p_2(t), \ldots, p_m(t)$.
3) Receive cost C_{i_t} where
$$C_{i_t} = \begin{cases} 1 & \text{if action } i_t \text{ fails,} \\ 0 & \text{otherwise.} \end{cases}$$
4) For $j = 1, 2, \ldots, m$, set
$$\hat{C}_j(t) = \begin{cases} C_j(t)/p_j(t) & \text{if } j = i_t, \\ 0 & \text{otherwise;} \end{cases}$$
$$w_j(t+1) = w_j(t) \cdot \exp\left(-\frac{\gamma}{m}\hat{C}_j(t)\right).$$

(b) **CE3**

Fig. 2. **Two Repair Strategies.** The weight w_i tracks how repair action i performs over multiple rounds. In the case of CE3, the weights can only decrease.

statement of their theorem.

Theorem 1 *(Auer et al. [2]) For any $T > 0$, suppose Exp3 uses the input parameter $\gamma = \min\{1, \sqrt{\frac{m \ln m}{(e-1)T}}\}$. Then, Exp3 achieves a regret that is $O(\sqrt{Tm \ln m})$.*

Auer et al. [2] also prove a lower bound on the regret achievable by any strategy, which we state in the following theorem.

Theorem 2 *(Auer et al. [2]) For any $m \geq 2$ and $T > 0$, there exists a distribution over the assignment of rewards such that the expected regret of any strategy is $\Omega(\sqrt{mT})$.*

Notice the $\sqrt{\ln m}$ gap between the bounds. However, the gap is small enough that it is not significant in practice. The regret achieved by Exp3 is thus near-optimal.

There are however some problems with a practical implementation of the Exp3 strategy that we must consider:

1) *Time/Space complexity:* Exp3 requires $O(m)$ space to store weights. The time required to sample according to the probabilities p_i can however be reduced to $O(\log^* m)$, using a clever sampling and weight maintenance algorithm given by Matias et al. [21]. For large action sets, the space expense can be significant; however, as noted earlier, we leave that to future work.
2) *Unnecessary updates to weights:* Note that the p_i probabilities will change while going from round t to round $t+1$ only if the weight of action i_t changes. The weight w_{i_t}, in turn, only changes if i_t succeeds. Thus, even after Exp3 converges to a subset of correct actions, assuming they exist, the weights will continue to increase to $+\infty$, thus incurring an unnecessary overhead.
A secondary point is that an implementation using floating-point arithmetic would suffer overflow-related inaccuracies, in spite of any re-normalization techniques one might apply. These inaccuracies would have an adverse impact on probability estimates, and thus on the convergence of the algorithm. Actions that do well continue to get selected in subsequent rounds, making overflow highly likely. If instead the weights of actions that repeatedly perform badly are decreased, underflow to zero can occur, but that is less likely and more easily handled through re-normalization because badly-performing actions are unlikely to be chosen in the future.

In order to eliminate the latter problem, we have designed a variant of Exp3 that uses a cost-based model and decreases weights rather than increasing them. Auer et al. [2] recommend that costs in $[0, 1]$ can be modeled as negative rewards in $[-1, 0]$ and handled by Exp3 after translation to

$[0,1]$. That approach does not resolve the above problem.

Cost-Based Exp3 Strategy. We present a cost-based variant of the Exp3 strategy, which we term CE3.

Figure 2(b) describes this strategy. As can be seen from the figure, there are two main points of difference:

1) In CE3, the chosen action i_t incurs a cost of 1 when it fails, otherwise no cost is incurred. Exactly the opposite happens in Exp3.
2) The weight of the chosen action needs to be updated only if it fails. In that case, the weight is decreased by an exponential factor.

Thus, the cost-based strategy CE3 avoids the problems with Exp3 except for the $O(m)$ space cost per round.

Kleinberg [22] has also given a cost-based variant of Exp3 that differs from CE3 in the weight update rule, but which also updates a weight of an action only when it fails.

In spite of the changes, CE3 also yields similar guarantees on the regret as Exp3. We state this in the following theorem:

Theorem 3 *For large enough $T > 0$, suppose that $m \ln m \leq (e-1)T$ and that CE3 uses the input parameter $\gamma = \sqrt{\frac{m \ln m}{(e-1)T}}$. Then, CE3 achieves a regret that is $O(\sqrt{Tm \ln m})$.*

Observe that if $m \ln m = o(T)$, CE3 achieves a regret of $o(T)$.

The proof of the above theorem, given below, uses a similar technique as the proof given for Theorem 1 by Auer et al. [2], but there are also some subtle differences.

Proof: We use the following facts which can be easily derived from the description of CE3 in Figure 2(b).

$$\hat{C}_i(t) \leq \frac{1}{p_i(t)} \leq \frac{m}{\gamma} \tag{1}$$

$$\sum_{i=1}^{m} p_i(t)\hat{C}_i(t) = p_{i_t}(t)\frac{C_{i_t}(t)}{p_{i_t}(t)} = C_{i_t}(t) \tag{2}$$

$$\sum_{i=1}^{m} p_i(t)\hat{C}_i(t)^2 \leq \hat{C}_{i_t}(t) = \sum_{i=1}^{m} \hat{C}_i(t) \tag{3}$$

Let $W_t = \sum_{i=1}^{m} w_i(t)$. Note that $W_1 = m$, while $W_t \geq W_{t+1}$ since the weights can only decrease.

For an (arbitrary) action j, since $w_j(T+1) \in [0,1]$ and $w_j(T+1) \leq W_{T+1}$, we have

$$\begin{aligned} \ln \frac{W_1}{W_{T+1}} &\leq \ln \frac{W_1}{w_j(T+1)} \\ &= \ln W_1 - \ln w_j(T+1) \\ &= \ln m - \frac{\gamma}{m}\sum_{t=1}^{T}(-\hat{C}_j(t)) \end{aligned}$$

Thus, we have the following upper bound inequality:

$$\ln \frac{W_1}{W_{T+1}} \leq \ln m + \frac{\gamma}{m}\sum_{t=1}^{T}\hat{C}_j(t) \tag{4}$$

We next derive a lower bound on $\ln \frac{W_1}{W_{T+1}}$.

$$\begin{aligned} \frac{W_{t+1}}{W_t} &= \sum_{i=1}^{m}\frac{w_i(t+1)}{W_t} = \sum_{i=1}^{m}\frac{w_i(t)}{W_t}\cdot\exp\left(-\frac{\gamma}{m}\hat{C}_i(t)\right) \\ &= \sum_{i=1}^{m}\frac{p_i(t)-\frac{\gamma}{m}}{1-\gamma}\cdot\exp\left(-\frac{\gamma}{m}\hat{C}_i(t)\right) \end{aligned}$$

Using the inequality $e^{-x} \leq 1-x+(e-2)x^2$ (for $0 \leq x \leq 1$) and Inequality 1 we get

$$\frac{W_{t+1}}{W_t} \leq \sum_{i=1}^{m}\frac{p_i(t)-\frac{\gamma}{m}}{1-\gamma}\cdot\left(1-\frac{\gamma}{m}\hat{C}_i(t)+\frac{(e-2)\gamma^2}{m^2}\hat{C}_i(t)^2\right)$$

Expanding the above expression, and simplifying while using Equation 2 and Inequality 3, we get

$$\begin{aligned} \frac{W_{t+1}}{W_t} &\leq 1-\frac{\gamma}{m(1-\gamma)}C_{i_t}(t)+\frac{(e-1)\gamma^2}{m^2(1-\gamma)}\sum_{i=1}^{m}\hat{C}_i(t) \\ &\leq \exp\left(-\frac{\gamma}{m(1-\gamma)}C_{i_t}(t)+\frac{(e-1)\gamma^2}{m^2(1-\gamma)}\sum_{i=1}^{m}\hat{C}_i(t)\right) \end{aligned}$$

where the last inequality is obtained using the fact $1+x \leq e^x$ for all $x \in \mathbb{R}$.

Taking natural logarithms on both sides, we obtain

$$\ln \frac{W_t}{W_{t+1}} \geq \frac{\gamma}{m(1-\gamma)}C_{i_t}(t)-\frac{(e-1)\gamma^2}{m^2(1-\gamma)}\sum_{i=1}^{m}\hat{C}_i(t)$$

Adding the inequalities for $t = 1, 2, 3, \ldots, T$, we get

$$\ln \frac{W_1}{W_{T+1}} \geq \frac{\gamma}{m(1-\gamma)}\sum_{t=1}^{T}C_{i_t}(t)-\frac{(e-1)\gamma^2}{m^2(1-\gamma)}\sum_{t=1}^{T}\sum_{i=1}^{m}\hat{C}_i(t)$$

Note that $\sum_{t=1}^{T} C_{i_t}(t)$ is the cumulative cost incurred by the CE3 algorithm, which we will denote by C_{CE3}. Thus, rewriting the above, we get the desired lower bound inequality:

$$\ln \frac{W_1}{W_{T+1}} \geq \frac{\gamma}{m(1-\gamma)}C_{\text{CE3}}-\frac{(e-1)\gamma^2}{m^2(1-\gamma)}\sum_{t=1}^{T}\sum_{i=1}^{m}\hat{C}_i(t) \tag{5}$$

Combining Inequalities (4) and (5), and taking expectation on both sides while noting that $E[\hat{C}_i(t)] = C_i(t)$ (by Equation 2), we get

$$\begin{aligned} &\ln m + \frac{\gamma}{m}\sum_{t=1}^{T}C_j(t) \\ \geq\ &\frac{\gamma}{m(1-\gamma)}E[C_{\text{CE3}}]-\frac{(e-1)\gamma^2}{m^2(1-\gamma)}\sum_{t=1}^{T}\sum_{i=1}^{m}C_i(t) \end{aligned}$$

Since j is an arbitrary action, the above equality also holds for $j = b$, where b is the best action in hindsight over the T rounds of execution. The quantity $\sum_{t=1}^{T} C_j(t)$ is thus the cost incurred by the system using the best action, which we will denote as C_{best}. Further, switching the order of summation on the RHS and since $C_i(t) \leq 1$, we obtain

$$\begin{aligned} \ln m + \frac{\gamma}{m}C_{\text{best}} &\geq \frac{\gamma}{m(1-\gamma)}E[C_{\text{CE3}}]-\frac{(e-1)\gamma^2}{m^2(1-\gamma)}mT \\ &= \frac{\gamma}{m(1-\gamma)}E[C_{\text{CE3}}]-\frac{(e-1)\gamma^2}{m(1-\gamma)}T \end{aligned}$$

Multiplying throughout by $\frac{m}{\gamma}$, and using $\frac{1}{1-\gamma} \geq 1$, we get

$$\begin{aligned} \frac{m \ln m}{\gamma} + C_{\text{best}} &\geq \frac{1}{1-\gamma} E[C_{\text{CE3}}] - \frac{(e-1)\gamma}{1-\gamma} T \\ &\geq E[C_{\text{CE3}}] - (e-1)\gamma T \end{aligned}$$

Rewriting, we obtain an upper bound on the regret of CE3:

$$E[C_{\text{CE3}}] - C_{\text{best}} \leq \frac{m \ln m}{\gamma} + (e-1)\gamma T \tag{6}$$

It is easily seen that for $\gamma = \sqrt{\frac{m \ln m}{(e-1)T}}$, CE3 achieves a regret that is $O(\sqrt{Tm \ln m})$.

■

Uniform Random Strategy. For comparison in our experimental evaluation, we also define the following simple randomized strategy:

> *UR:* Pick a repair action from the set of m actions uniformly at random.

It is easy to show that, in the worst case, if there is no correct repair under the fault model, then the UR strategy can suffer $\Theta(T)$ failures over T rounds. The key insight is that UR does not track past performance of actions; it can pick an imperfect action infinitely often and the environment can force that action to fail each time.

Theorem 4 *The expected regret of the UR strategy for a worst-case environment over T rounds is $\Theta(T)$.*

IV. Case Study: A Network Monitor

We now consider how the CE3 strategy performs in practice. Our case study is a simplified network monitor, a packet filtering system to detect malicious traffic, taken from Varghese's book on Network Algorithmics [3].

The network monitor seeks to drop packets that match a syntactic pattern of malicious behavior while forwarding all other packets. The data payload of a packet is viewed as a URL of length L that could potentially have malicious code embedded in it. We view this payload as a sequence of characters (bytes) of length L, with the last character being a null character. For each possible character c, there is an associated threshold τ_c that specifies that c must appear no more than $\tau_c L$ times in the packet. A malicious packet is defined as one that violates this threshold restriction for some character.

Varghese steps the reader through several implementations of this network monitor, from very simple and slow, to a cleverly optimized version. For our case study, we chose the most highly optimized version as the implementation under online test, and refer to it as Impl. The second-most optimized variant is considered to be the correct specification and is used to check the output of Impl; we refer to it as Spec. We begin by describing these versions in Section IV-A.

A. Two Versions

Both Spec and Impl are finite-state machines that make use of lookup tables (arrays).

Figure 3(a) depicts the operation of Spec. Spec makes use of two lookup tables. The first, `thresh_arr`, stores thresholds for each of the 256 possible characters in the form of shifts rather than as floating-point fractions. The second table is an array `count_arr` of length 256 storing for each character c a count of the number of times c was encountered in this packet. In each round, Spec begins by initializing `count_arr` with zeroes, one entry per cycle. Thus, Spec takes 256 cycles just to initialize its `count_arr`. After initialization, it proceeds to read and process the bytes in the packet, one byte per cycle. For each character c that is read, the entry `count_arr`$[c]$ is incremented. As the data payload is of length L, this process takes L cycles. By calculating the ratio of the count over the threshold on-the-fly while reading each character, a final pass through the count array to check thresholds is avoided. Thus, the decision on whether to drop or forward the packet is made in $256 + L$ cycles.

Impl runs much faster than Spec, in $1 + L$ cycles. Its operation is shown in Figure 3(b). The key insight used in Impl is to delay initialization of `count_arr` at the cost of using slightly more storage. Briefly, Impl initializes only one entry in `count_arr` per round of execution (packet). However, it also maintains a register to keep track of a "global generation count," which is the current round number (modulo maximum storable value in the register), as well as an array of generation counts for the last time each character's entry in `count_arr` was updated. If the generation count of the currently read character does not match the global generation count, that character's `count_arr` entry is set to 1 (initialization of 0 plus the first occurrence), otherwise, it is incremented as usual. By maintaining a large enough generation register, Impl can process each packet in $1+L$ cycles while avoiding wraparound issues with the global generation count.

For further details on the above case study, we refer the reader to the book [3].

B. Fault Injection and Verification

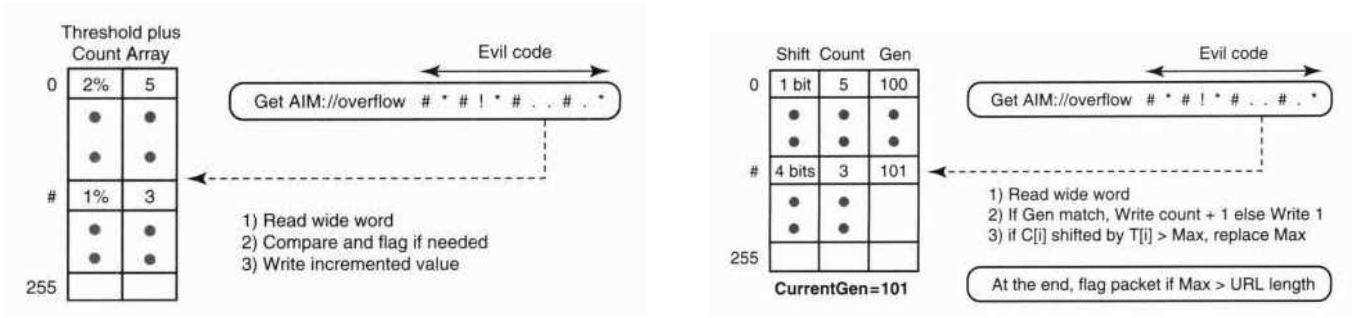
For our experimental evaluation, we created three buggy versions of Impl, each leading to packets being incorrectly classified.

The first buggy version was obtained by injecting a fault into a single assignment in the code for Impl. Instead of setting the `count_arr` entry to 1 when a generation count mismatch is detected, we set it to 0. Thus, an off-by-one error is introduced leading to a malicious packet being incorrectly forwarded.

The second buggy version was created similarly, this time for when the generation count matches up. Instead of incrementing the `count_arr` entry for character c by 1 when c is encountered, we leave the count unchanged.

Finally, the third buggy version was created by injecting both of the above faults.

We first attempted to formally verify the correctness of Impl, checking that it generated equivalent results to Spec for a



(a) Checker Module Spec

(b) Optimized Implementation Impl

Fig. 3. **Two implementations of a simplified network monitor.** Figures reproduced from [3].

packet. The state-of-the-art Cadence SMV model checker [23] was used for this purpose. However, SMV ran out of 2 GB of memory even for a greatly simplified version with $L = 8$, only 8 possible characters, and a 3-bit generation count register.

We then used simulation to test Impl against Spec on randomly generated packets, where the characters in the packet were selected uniformly at random and the two variants run in tandem for several thousand rounds. This random testing successfully found mismatches between the answers generated by Impl and Spec, but diagnosis and repair remained to be done.

C. Online Error Detection and Repair

Consider using Spec to check online the results generated by Impl. Since Spec runs 255 cycles slower than Impl on each packet, we cannot check the output of Impl on every packet. However, this application is one in which a few failures can be tolerated (amounting to a few dropped packets), but it would be unacceptable to drop several thousand benign packets. Thus, for example, we can use Spec to check every other output generated by Impl.

While Spec can be used to detect errors, it cannot take over from Impl on detecting a failure without a significant performance loss – it would need to drop every other packet until Impl is repaired. Our goal in this experiment was to investigate using the CE3 strategy given in the preceding section to evolve Impl towards a correct system over several rounds of execution.

We created an overall network monitor comprising three modules executing in parallel: Impl, Spec, and $\overline{\text{Impl}}$, where $\overline{\text{Impl}}$ is a self-morphing version of Impl. Using backward slicing from the statements in the code that make the "drop"/"forward" decision on a packet, we identified that the updates to the count array were the only statements that could affect the decision, since the threshold array entries remain constant. These update statements to the count array are exactly the two statements in which faults were injected, as described in Section IV-B.

The fault model assumed in the design of $\overline{\text{Impl}}$ is that at most one update statement is wrong. We deliberately made this choice so that the fault model is imprecise if both faults are injected, and we can measure the impact of that imprecision. Thus, in each round, $\overline{\text{Impl}}$ changes one of the two update statements in its code, choosing from a space of $2L$ candidate repair statements (actions). This space comprises statements that set the count array entry to an arbitrary integer constant in $[0, L-1]$ as well as those that increment the entry by any integer in $[0, L-1]$.

The above set of repair statements is small enough that the space requirements for implementing CE3 in $\overline{\text{Impl}}$ are only linear in the length of the packet. For the first two buggy versions with a single injected fault, it includes the correct repair. Thus, it forms a useful space of possible repairs for the buggy versions under investigation in this case study.

Note that our approach requires a network monitor to use two additional cores for Spec and $\overline{\text{Impl}}$ for every core used for Impl. We believe that the parallelism available in the future can well support this redundancy. In fact, the Cisco Silicon Packet Processor already has an array of 188 programmable RISC cores [24].

D. Experimental Results

We implemented a simulator for the system comprising Impl, Spec, and $\overline{\text{Impl}}$, as described above. To experiment with this system at a few different scales, we parameterized the number of characters allowed in the URL (denoted N), varying it over the set $\{64, 128, 256\}$. To simplify the design of the overall system, we set L to be equal to N. Thus, Impl needs $N+1$ cycles to process a single packet, while Spec needs $2N$ cycles.

Each simulation of an execution of the system was run for one million rounds, where each round comprises $2N + 2$ cycles. Thus, Impl processes two packets in each round, the first of which is also processed by Spec in order to check the result of Impl. Packets were generated uniformly at random as described earlier. Spec uses the first $2N$ cycles to process the packet received at the start of the round, and the remaining 2 cycles to check the results of Impl and $\overline{\text{Impl}}$; in the latter case, it updates the weight of the repair action if needed. The self-morphing variant $\overline{\text{Impl}}$ also processes only the packet received

Repair Strategy	N = 64		N = 128		N = 256	
	Mean	Std. Dev.	Mean	Std. Dev.	Mean	Std. Dev.
None	423.30	10.55	1020.00	23.76	2163.63	42.54
CE3	29.73	20.53	85.42	57.01	348.35	224.21
UR	254.70	293.63	464.55	440.34	992.65	1081.26

(a) Fault 1

Repair Strategy	N = 64		N = 128		N = 256	
	Mean	Std. Dev.	Mean	Std. Dev.	Mean	Std. Dev.
None	471.50	11.02	1134.25	25.28	2423.18	47.58
CE3	36.61	27.93	75.62	52.46	330.33	199.41
UR	301.04	302.70	479.97	444.48	864.81	963.73

(b) Fault 2

Repair Strategy	N = 64		N = 128		N = 256	
	Mean	Std. Dev.	Mean	Std. Dev.	Mean	Std. Dev.
None	470.95	18.05	1123.42	32.39	2427.99	211.15
CE3	817.20	49.41	2239.26	94.55	6336.40	221.54
UR	6535.87	366.94	30917.90	1307.95	120458.61	3607.66

(c) Both Faults

Fig. 4. **Experimental data obtained by simulations with randomly generated packets.** The numbers shown in the tables are the measured average and standard deviation of the number of failures suffered by Impl while using the indicated repair strategy. In the case of Tables (a) and (b), these also coincide with the regret as there exists a valid repair under the fault model.

at the start of the round, using the remainder of the round to run CE3 to select the repair action for use in the next round.

In all, we performed 27 experiments. In each experiment, we picked a combination of a buggy version of Impl (out of the 3 described in Section IV-B), one of the 3 possible values of N, and one of 3 possible repair strategies (CE3, UR, or no repair). For each experiment, we ran 100 simulations, each with a different random seed. At the end, we measure the mean and standard deviation of the number of failues of Impl obtained with each repair strategy.

Our results are shown in Figure 4. We note that for a precise fault model (i.e., when only a single fault is injected) the CE3 strategy helps to reduce the number of failures of Impl as compared to doing no online repair at all; the improvement ranges from 15 times fewer failures for $N = 64$ to about a factor of 7 fewer failures for $N = 256$. However, when both faults are injected, the CE3 strategy does a bit worse, but is still within a factor of 3 of the number of failures suffered by the unrepaired system. We believe that this factor can be brought down significantly as the number of rounds increases.

We note that for a single fault injection (Figures 4(a) and (b)) the CE3 repair strategy does much better than the UR strategy for all combinations of buggy implementations and N. Both the average and the standard deviation of the number of failures suffered by Impl under the CE3 strategy are significantly less than under the UR strategy, ranging from a factor of 10 for $N = 64$ to a factor of about 3 for $N = 256$.

Significantly, notice from Figure 4(c) that when both faults are injected, there is no single correct repair statement (both updates to the count array must be repaired). In this case, the CE3 strategy does much better than UR, suffering 20 times fewer failures on average for $N = 256$.

Overall, we can conclude that if we search for a repair action under an accurate fault model, the CE3 strategy can do much better than either leaving the system unrepaired or picking an action completely at random (UR).

V. Conclusion

In summary, this paper makes a first step towards the construction of autonomic reactive systems based on online learning. We have given a framework for leveraging parallelism to pro-actively explore the space of repairs even before a failure is encountered. A mapping to the well-studied multi-armed bandit problem has been given, along with a novel cost-based strategy for the same. An experimental evaluation with a simplified network monitor shows that our repair strategy can be effective.

Finally, we also mention that the applications of online learning and the CE3 strategy presented herein go well beyond repairing faults; for example, it can be used in auto-tuners to improve the performance of a system as it runs.

ACKNOWLEDGMENTS

The author is grateful to Avrim Blum, Randal Bryant, Edward Lee, Sharad Malik, and George Varghese for helpful discussions. The author acknowledges the support of the Gigascale Systems Research Focus Center, one of five research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program. This research was also supported in part by the National Science Foundation and a grant from Microsoft Research.

REFERENCES

[1] H. Robbins, "Some aspects of the sequential design of experiments," *Bulletin of the American Mathematical Society*, vol. 55, pp. 527–535, 1952.

[2] P. Auer, N. Cesa-Bianchi, Y. Freund, and R. E. Schapire, "The non-stochastic multiarmed bandit problem," *SIAM Journal on Computing*, vol. 32, no. 1, pp. 48–77, 2002.

[3] G. Varghese, *Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices*. Morgan Kaufmann Publishers, 2005.

[4] M. C. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee, "Enhancing server availability and security through failure-oblivious computing," in *6th Symposium on Operating System Design and Implementation (OSDI)*, 2004, pp. 303–316.

[5] S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis, "Building a reactive immune system for software services," in *Proceedings of the USENIX Annual Technical Conference*, 2005, pp. 149–161.

[6] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou, "Rx: Treating bugs as allergies - a safe method to survive software failures," in *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, 2005, pp. 235–248.

[7] B. R. Liblit, "Cooperative bug isolation," Ph.D. dissertation, University of California, Berkeley, Dec. 2004.

[8] E. Kiciman, "Using statistical monitoring to detect failures in internet services," Ph.D. dissertation, Stanford University, September 2005.

[9] A. Easwaran, S. Kannan, and O. Sokolsky, "Steering of discrete event systems: Control theory approach," in *Proc. Workshop on Run-time Verification (RV)*, 2005.

[10] N. Delgado, A. Q. Gates, and S. Roach, "A taxonomy and catalog of runtime software-fault monitoring tools," *IEEE Transactions on Software Engineering*, vol. 30, no. 12, pp. 859–872, December 2004.

[11] K. Havelund and G. Roşu, "An overview of the runtime verification tool Java PathExplorer," *Formal Methods in System Design*, vol. 24, no. 2, pp. 189–215, 2004.

[12] B. C. Williams, M. D. Ingham, S. H. Chung, and P. H. Elliott, "Model-based programming of intelligent embedded systems and robotic space explorers," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 212–237, 2003.

[13] J. de Kleer and J. Kurien, "Fundamentals of model-based diagnosis," in *SafeProcess 2003*, 2003.

[14] B. C. Williams and P. P. Nayak, "A model-based approach to reactive self-configuring systems," in *AAAI/IAAI, Vol. 2*, 1996, pp. 971–978.

[15] B. Demsky and M. C. Rinard, "Automatic detection and repair of errors in data structures," in *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 2003, pp. 78–95.

[16] ——, "Data structure repair using goal-directed reasoning," in *27th International Conference on Software Engineering (ICSE)*, 2005, pp. 176–185.

[17] K. R. Joshi, W. H. Sanders, M. A. Hiltunen, and R. D. Schlichting, "Automatic recovery using bounded partially observable markov decision processes," in *International Conference on Dependable Systems and Networks (DSN)*, 2006, pp. 445–456.

[18] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, "Microreboot - a technique for cheap recovery," in *6th Symposium on Operating System Design and Implementation (OSDI)*, 2004, pp. 31–44.

[19] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, Jan.-Mar. 2004.

[20] N. Littlestone, "Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm." *Machine Learning*, vol. 2, no. 4, pp. 285–318, 1987.

[21] Y. Matias, J. Vitter, and W.-C. Ni, "Dynamic generation of discrete random variates," *Theory of Computing Systems (TCS)*, vol. 36, no. 4, pp. 329–358, 2003.

[22] R. D. Kleinberg, "Online decision problems with large strategy sets," Ph.D. dissertation, Massachusetts Institute of Technology, 2005.

[23] "Cadence SMV model checker," http://www.kenmcmil.com/smv.html.

[24] Cisco Systems White Paper, "Requirements for next-generation core routing systems," Available at `http://www.cisco.com/warp/public/cc/pd/rt/12000/clc/prodlit/reqng_wp.pdf`, URL circa March 2007.