Autonomic Computing: Learning to Repair Systems Effectively

Nicholas Lynn (ngl@andrew.cmu.edu)
Advisor: David Garlan (garlan@cs.cmu.edu)
School of Computer Science, Carnegie Mellon University
5000 Forbes Ave
Pittsburgh, PA, 15213
USA

Abstract. This research aims to integrate temporal difference learning methods into an autonomic learning system. The current Rainbow architecture models an adaptation engine that examines the system state to assess problems and determines a best course of action based on pre-programmed expertise. However, when expert knowledge is not readily available for a system, learning proper actions without *a priori* knowledge would often be more useful in practical situations. This report characterizes initial testing of a new learning engine designed to learn the proper corrective actions to take in order to repair problems in systems.

Specifically, reinforcement learning methods can enable a system to learn the proper actions by actually trying out actions and seeing how well they do. This project compares Q-learning and SARSA temporal difference learning algorithms with the existing expert algorithm on the Carnegie-Mellon designed Rainbow simulated system. Since they approach reinforcement learning in a different way, comparing the success rate and speed of the algorithms on various scenarios will provide evidence about their efficacy. My research aims to clarify the benefits and costs associated with these algorithms.

1. INTRODUCTION

The core problem that I address with this autonomic learner is how to increase the ability of systems to maintain themselves. Computer systems continue to become increasingly complex, with multitudes of new features. Furthermore, these systems are increasingly running in highly dynamic environments, where resources and requirements are often in flux. With this increase in complexity, systems also have an increasing variety and impact of possible failures and problems. Simple catch-all solutions such as rebooting or shutting down an offending program or client are often wholly unacceptable strategies for coping with such problems. For a system that requires 100% uptime to provide client services, any kind of failure can be a financially devastating problem. So the owners and maintainers of these systems often turn to human system administrators to apply their expertise to repair their systems after encountering the potential pitfalls.

This paper explores the viability of autonomous self-repairing computer systems. Of course, many of the problems associated with autonomous repair fall beyond the scope of this paper. Here, I specifically address the question of how best to apply temporal difference learning algorithms to choose repairs for dynamic systems. A temporal difference algorithm compares the states of the system as time progresses with updates to the preference for any particular repair after each decision. The learner using this algorithm requires the system to provide options for repairs to try, as well as a meaningful representation of the state of the system. From these, the learner may be able to learn to repair the system as well as a system that uses preordained expert strategies for repairing problems.

Expert humans can often provide degrees of precision and insight that are not available to computers, but computers have the advantage that they can always be there to address problems, even after hours or in a violent blizzard. Expecting the same level of performance from an autonomous machine as a human system expert is not appropriate; however, we can definitely produce autonomous systems that work to correct most of the worst problems most of the time. If the system is having particularly bad problems, an automatic repair mechanism could fail where an expert could identify some small but critical repair that solves the problem. However, for the more common case of something routine going wrong, a system that automatically detects the problem and fixes it pretty well, no matter the time or circumstances, is immensely useful. In addition, a computer-based repair scheme is not susceptible to human error from being overworked or tired.

Towards this end, IBM's research group created the Autonomic Computing Initiative, aiming to create entirely self-managing systems that can scale and learn about themselves effectively. The Initiative has divided the field of autonomic computing into four parts: self-configuration, self-healing, self-optimization, and self-protection [1]. The varied solutions pursued at IBM even include autonomous systems with processes that are the analogues of the human brain to model the learning process [2]. The work reported in this paper falls in the category of self-healing, which addresses the detection and reaction to errors in systems. Specifically, I focus on learning appropriate reactions to system errors. By doing so, an autonomous learner such as

this one could often replace the current human-based repair work and analysis.

The desire to allow for autonomous recovery of systems is a common one. In order for a complete system of that kind to exist, there are two requisite components: Automatic detection of faults and effective forms of recovery.[3]. Each of these components has myriad levels of complexity to reach a fully autonomic system capable of functioning entirely without assistance. The work I am performing here is focused on only the "effective" part of recovery. If the detection of faults and choice of repairs is given, I will show that the correct choice can be made.

A crucial part of the learner lies in its abstraction from any particular underlying system. My proposed learner abstracts the choice of repairs from any particulars about the underlying system. Such an abstraction can combine with other abstractions in this field, such as the already developed effector abstraction [4] that could execute the repairs on any number of subsystems. While developing an autonomous system for a specific task could provide insights, an autonomous framework with parts that are easy to implement or that "plug-in" to the framework allow for much more flexibility.

Rainbow is a reusable infrastructure developed to help facilitate autonomic computing [5] which enables the kind of self-repair that this project expands. Rainbow monitors an underlying system with multiple probes to observe the performance of numerous specific parts (such as bandwidth, latency, and load in a client/server setup). It then compiles these raw data into a model of the system, which is what Rainbow will then act upon for its decision-making and eventual repair choice. The repair choice and issuance of the command to repair causes changes in the underlying system. Currently, the decision-making process for a given system is predefined, based on previous experimentally determined "best decisions" for given circumstances.

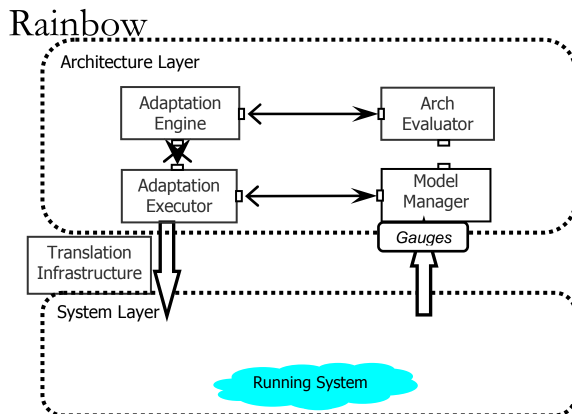

Fig 1: The Rainbow Architecture

In this paper, I will test the efficacy of two temporal difference learning algorithms constructed to learn what appropriate repairs to undertake for broken systems. The learner is highly abstracted, requiring very little from an underlying system to choose the best repairs. The underlying systems (here, the Rainbow architecture) must only implement a small number of core functions for the learner to function. I use the Rainbow framework because it is an especially useful tool with which to implement such functionality.

Specifically, given a system state and a set of options to attempt to repair the problem, the algorithm will learn which is the most appropriate. This project judges the merits of each algorithm by comparing their performance with hard-coded expert decision-makers, aiming to achieve at least that level of performance in correct decisions and in timeliness. It fits into the Rainbow model as an adjunct to the adaptation engine, performing the choice of which repair to execute on the system.

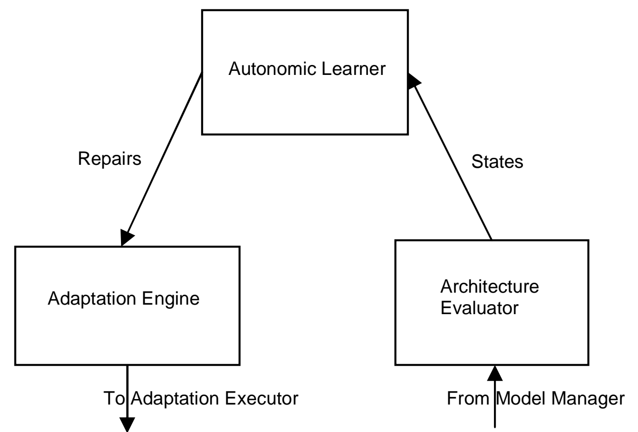

Fig 2: Where this autonomic learner fits into the Rainbow model.

2. METHODS

The purpose of the learner tested here is to replace these expert-determined "best decisions" with better learned decisions. The learner itself sits on top of the Rainbow system, which provides the learner with a set of states and repairs that it knows how to execute. The collapse of the multitude of data available to the underlying system into a finite set of states and the ability to execute repairs are the only responsibilities that lie with the underlying system, so any number of systems can substitute for Rainbow by providing the information the learner needs. Enabling this to happen requires reducing the level of detail the Rainbow model provides to a finite representation of states and repairs.

2.1 State

```
public abstract class State{

public abstract int getID();
public abstract double getReward();
public abstract boolean isAGoal();
public abstract boolean compareState(State s);
public abstract RepairScore
getRepairScore(Repair r);
public abstract Repair getBestRepair();
public abstract Repair chooseRepair();
public abstract void updateRepairScore(Repair
r,State s);

}
```

A state is the functional element of the learner. It is an abstract class which requires subclasses to implement the following functions. In general, all instances of a state will share a subclass that expresses whatever learning algorithm the learner is using. The learner uses LearnedStateQ and LearnedStateSARSA objects that each extend this base class.

The functions that are not simple accessors are the core of how the learner learns. GetBestRepair() chooses the highest-valued repair for this state. The values it uses to make this comparison arise from the historical success of using this repair from this state. The chooseRepair() function applies the policy of repair choice under whatever algorithm is running for this state. The updateRepairScore() function is called after a repair has finished and updates the state/repair pair with appropriate success values as indicated by whatever learning algorithm this learner uses. This function uses the Q-learning and SARSA algorithms described later.

Developing appropriate states for an existing system require some degree of knowledge of how the underlying system functions. A developer seeking to generate states for the learner to use would need to identify which properties of the system and what kinds of combinations are meaningful. In general, someone familiar with the system knows how to categorize its operating behavior into a few differing categories. The goal in state generation is for the properties that define a state to be interrelated in a sensible fashion and somehow different from the other defined states. Generating too many states slows learning down tremendously because each state needs to be visited a fair amount to obtain good learning. For example, defining a different state for every increment of 1 ms of latency over a connection is going to generate too many states that could have reasonably been combined. Generating too few states hinders repair effectiveness. Consider a system with only ‘good’ or ‘bad’ as states. A learner would learn the best average action to take to fix the system but would be ignoring the advantages it could gain from knowing why the system is in a bad state. Generation of states is a balancing act between making too many or too few.

2.2 Repair

```
public interface Repair {

public int id () ;
public int type ();
public void execute(String elemID) ;
}
```

The execute() function takes an ID for the broken component in the system. This allows for a repair like “Fix the client’s bandwidth” to specify which client is broken. The system, e.g., the Rainbow model, provides the details of how to actually execute the repair.

2.3 Q-Learning [6]

Q-learning is one learning algorithm used to update the state/repair pair values of the learner. This algorithm is an off-policy temporal difference learner. Off-policy means that its predictions about future actions are independent of the actual policy that makes those decisions. Specifically, each time a repair is executed, the current state/repair pair corresponding to that state will update based on the following update formula:

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

2.4 SARSA [7]

SARSA is an on-policy learning algorithm. An on-policy algorithm uses the same policy that makes choices of repairs to predict what choices will be made in the future. This operates very similarly to Q-Learning; but, the algorithm uses the defined policy of choosing a State (chooseRepair()) to predict future actions instead of assuming a greedy choice in the way that Q-learning does. Past research indicates that, over time, SARSA will produce the same results as Q-learning, but its patterns of learning are different. In general, SARSA is more measured, choosing states with better expected cost instead of best-case cost. Its update formula is:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t))$$

3. IMPLEMENTATION

State #	Client latency	Client bandwidth	Client’s server’s load	Client alternative viability
0	1	0 or 1	0 or 1	0 or 1
1	0	1	1	0 or 1
2	0	1	0	1
3	0	1	0	0
4	0	0	0 or 1	1
5	0	0	0 or 1	0

Figure 3: State definitions (1 means it is within the acceptable range for that property, 0 means it is in violation of the acceptable range)

The simulation system used here creates six states at its initialization, one goal state and five error states (Fig. Y). This is the only part of the learner that requires that a user provides some information to the learner. The information needed is in the state objects gathered from collapsing the system's internal state into one of these six. For the purposes of this experiment, the goal state has a reward value of 1.0, while each error state is equally weighted with -1.0 as a reward. This means that algorithms will optimize patterns that achieve the one non-erroneous goal state, with shorter predicted paths accruing less negative reward. This favors shorter and generally faster repairs. The learner was run on nine system failures for 500 iterations for each learning algorithm.

Repair #	Action taken
0	Do nothing
1	Add server
2	Move client

Figure 4: Repair definitions

The learner used three repair actions to attempt to repair the system. These repairs were executed by a simulated Rainbow system. Each repair's execution was allowed to finish (as defined on a per-repair basis) before re-polling the system state to determine the outcome of the repair.

4. EXPERIMENTAL DESIGN

Each algorithm as described above was implemented with gamma = 0.3 and alpha = 0.4*(100/99+n) where n is the number of times this state/action pair has been visited. Each also used an epsilon-greedy policy to choose repairs in each step, given by the following:

```
With probability (100/99+n) choose repair action
at random, otherwise choose the highest valued
action in this state.
```

The underlying system was a Rainbow-based simulation designed by S. Cheng at Carnegie Mellon University. It modeled the effects of repairs on a simulated underlying system. It also reported the system's state for the learner as specified above.

5. RESULTS

State #	Expert System	Q-learning	SARSA
1	0	2	2
2	1 or 2	2	2
3	1	1	1
4	2	0	0
5	0	0	0

Figure 5: Expert repair strategy and optimal repair number determined by each approach after 500 iterations

The algorithms each produced the same results as each other by the end of the run. The patterns of these scores over time were similar for each of the learner's algorithms as well. I will address each state's learning individually. I will show only one graph for each state because the other one that is not shown looks very similar in all cases. For all the charts, repair 0 is a dotted line, repair 1 is a solid line, and repair 2 is a dashed line.

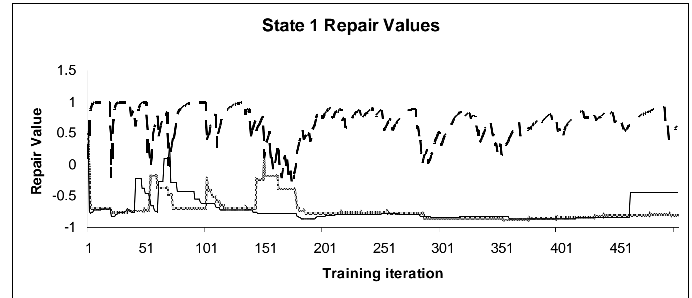

Figure 6: Repair values for state 1 (Q learning) over time. The upper line is repair 2.

In state 1, something has negatively affected the client's latency but its bandwidth and server load are well within expectations. The expert system suggests taking no action, while each learner shows a strong preference for the 'move client' action. While doing nothing could certainly allow an aberrant network hiccup that affects latency to clear out, moving the client to a new server sets it up on an entirely new connection which is unlikely to be in a failure state as well.

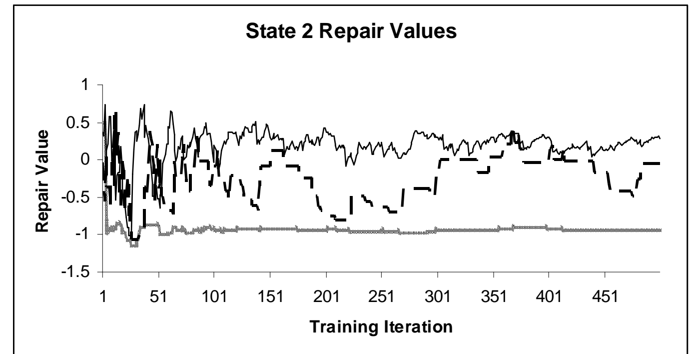

Figure 7: Repair values for state 2 (Q learning) over time. The upper line is repair 2 and the middle is repair 1

In state 2, the client is connected to a server group that is overloaded. However, alternate server groups are available to switch. The expert system will choose either repair, while the learners each settle on 'move client.' This is not a clear decision in either case, with the 'add server' repair being very close in preference throughout the learning period.

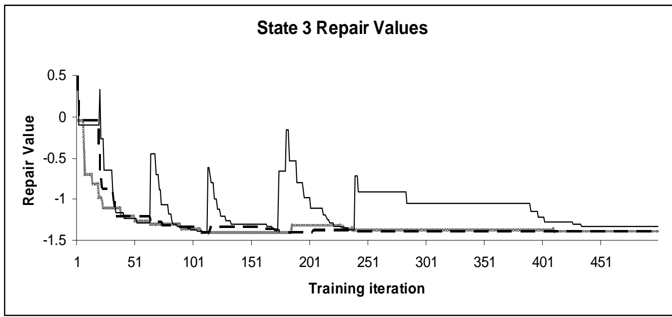

Figure 8: Repair values for state 3 (SARSA) over time. The upper line is repair 1.

In state 3, the state is similar to state 2, except there is no alternate group to move to. This eliminates the 'move client' option, making the expert prefer the 'add server' repair to fix the problem. The learners each come to the same conclusion as the expert system, although the difference between this repair and the other ones is not high. Often repair 1 here would fail, however it will fail less than the other options, giving it only a small advantage above the others in value.

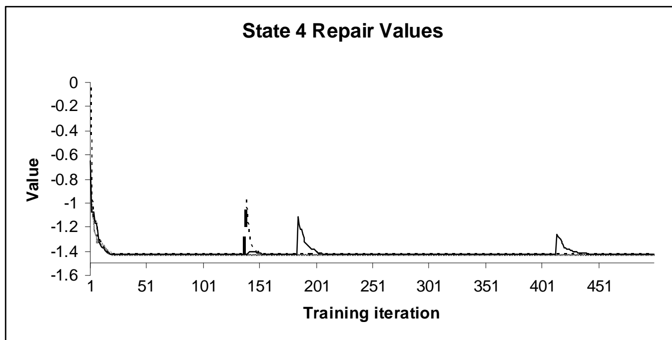

Figure 8: Repair values for state 3 (Q learning) over time. At the end all of the values are precisely equal.

In state 4, the bandwidth and latency available to this client are bad, but there is an alternative group. While it makes sense that moving the client would fix this problem, neither learner found any evidence that any repair strategy ever works. This could be a specific issue with the training set or the simulation, but this is the only state where the learners and the expert disagree. Interestingly enough, should the expert attempt to follow its strategy here, it will do no better than the learner's strategy because no repair actually fixes the system from this state.

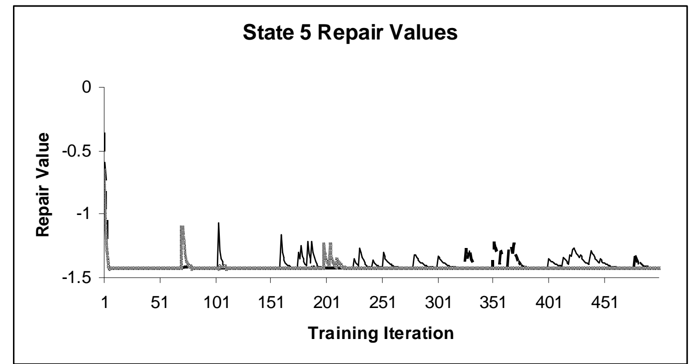

Figure 8: Repair values for state 3 (Q learning) over time. At the end all of the values are precisely equal.

In state 5, the client has the bad values of state 4 but there are no alternatives to move to. The expert suggests doing nothing because no repair really makes sense when there everything for this client is broken and there is nowhere to go. The learners arrive at the same conclusion, finding no evidence that any repair is useful.

5.1 Discussion

The results of the program are encouraging in that the learner very accurately identified appropriate repair methods on the given client-server system. The results were not perfect, but they showed a great deal of potential in learning the relevant solutions in a situation. In each case where any repair was useful, the system settled on a decisive solution which was sensible. In fact, in every case where it came up with an answer, the results matched those determined by expertise. SARSA and Q-Learning generated basically the same estimates of repair value—any differences were small and did not show substantial patterns of bias.

Future runs of the algorithm would do well to integrate more ways to perturb the state to present a wider variety of underlying problems with the system that map to the same state. Probably some of these changes would produce results analogous to those the expert system has hard-coded for state 4 in particular. While 'move client' never worked in this iteration, that strategy probably works for some different underlying system states, while 'add server' never does. Such an effect observed enough during training would correct the relative levels of preference for the 'move client' repair.

A number of limitations and additional opportunities for enhancement are apparent. The actual expert learner used in Rainbow has systems for which a failed repair will trigger an attempt at the second-best repair, which the learner currently does not. This strategy offers a substantial advantage in many cases, and we should modify this learner to allow testing of second-best repairs. The other advantage of Rainbow is its knowing when to quit. This learner will continually try to execute repairs until time runs out. This can be very costly for certain repairs (such as adding a server). The notion of cost is

not one that this learner addresses very well because the learner erroneously acts as if repeatedly trying to add a server and failing is no more costly than doing nothing for the same length of time. For example, in state 5 where no solution is possible, simply stopping would be more efficient in a real system.

Another potential limitation is the number of runs necessary to learn the best options. This run took around 100 iterations in each state before a best repair began emerging, and in some cases (state 2) the best repair was still not totally clear after 500 iterations of running. For a system that is costly to operate or cannot simulate a large number of runs, this learner will not be appropriate. However, since one can seed the repair values with whatever values one wants, one can pre-seed the repair values appropriately and save some time if needed. The exploratory nature of the learning process will make mistakes and poor choices to learn what not to do, so systems with high costs of failure would need to require a simulation to get an adequate time period in which to learn.

The learner system also has a narrow scope and requires some *a priori* information, which might eventually be generated by bootstrapping. For example, the scheme of state collapse from the underlying system to a state object is one that a learner could conceivably bootstrap, but the associated complexity would be enormous.

What the learner requires now is only for users of the learner to be able to answer the question, "What are the important properties of the underlying system?" This question is much simpler than what the expert system requires, which is an answer to "What are the important properties of the underlying system and how exactly do we react to them effectively?" Therefore, this learner requires a simpler set of information than one wholly reliant on expertise in order to function effectively. What this learner loses over the current expertise-oriented Rainbow model in raw hard-coded efficiency, it gains in adaptability to new situations.

6. CONCLUSION

The results are very promising: this work shows that using temporal difference learning as a replacement for strict expert methods is reasonable. In order to function at the level of an expert model, the learner requires only methods to implement repairs to its system and a mapping from the system to states. In addition, its abilities are independent of how the underlying system actually functions. As a proof of concept, these data definitely demonstrate feasibility. The independence from the underlying system allows the learner to work with multiple environments, from simple client-server systems to system security or databases. Conveniently, each algorithm performed basically the same, meaning that one could use either one in future work of this kind.

All further changes to the learner's capabilities build on the assumption that the underlying algorithms learn accurately. From this base, additions such as a GUI or the smaller improvements described below are straightforward. The proof of concept of the algorithm's successes demonstrates adequate baseline performance to justify further work in specific applications and enhancements.

6.1 Future work

Avenues to expand on this research split into two categories. Firstly, some methodological improvements would the functionality of this particular learning tool. Secondly, some work would facilitate self-healing repair for all systems.

The possible functionality improvements for this particular learner are numerous, depending upon the scope of development. The simplest improvements would improve the accuracy and real-world implications of the existing algorithms. Slightly more complex changes involve expanding the learner to perform the "state collapse" automatically, as well as learning the appropriate time to wait for a repair to execute before examining the result. Much larger in scope are the ideal changes of allowing for a history of repairs and states to be incorporated cumulatively to define otherwise inaccessible repair strategies. For example, if no single "repair X" can repair a problem, maybe combining "repair X" with "repair Y" and applying them successively can fix the system. Such an idea is well beyond the complexity of this Q-learning algorithm, although the framework provides a starting place.

A first and very important improvement would be learning the proper length of time for a repair to execute. If you reexamine the system too quickly, a repair may simply have had not enough time to execute, giving an erroneous impression that the repair was a failure. Even worse, future repairs may inappropriately register as having fixed the problem, when correct attribution would identify the latent effects of the prior repair. Such results break the algorithm, as the results are no longer attributable to the correct causes. Having extremely long wait times wastes time because the repair has finished executing long ago. In addition, intervening problems with the system would again mislead the algorithm in attributing success or failure to the repair. Accurately learning precisely when a repair has 'finished' would be very useful for future work.

In line with the above, other systems have pursued the idea of cost for repairs through a two-part system [8]. Firstly, the system can perform "test actions" that incur no cost. These actions are meant to give the learner accurate information about how likely certain repairs are to work. However, it can also perform the same repairs as "repair actions" which weigh the associated cost of the repair when determining which choice to make. This allows quick or simple repairs to have preference in some cases because, even if they fail, they don't take up a lot of time or resources to attempt.

Another simple improvement would be to bar trying a particular repair after it has failed once for a current instance. This

requires building in the assumption that whatever is preventing this state from being fixable by the failed repair is indigenous to the full underlying system state instead of the collapsed state reported by the system.

A logical extension of this property, albeit one much larger in scope, would be amending the collapsing process to include whatever is making this state unique. For example, if a certain state/repair combination either works very well or not at all, the state representation for it is likely to not capture everything important about the underlying system. Having the learner communicate with the system to pick out another important attribute in the underlying state in order to generate a new state for the learner to examine would be quite useful.

The addition of a memory for past failed state/repair pairs is a very useful opportunity for further research. This provides another approach towards the state collapsing problem described above. Basically, one would aim to teach the learner the correct actions to take by integrating past failed attempts with its existing knowledge about what states and repairs are appropriate in the current state. If a repair fails using the information the learner has for the current state, it could look backwards to see if past repair/state combinations are indicative of a repair being unable to succeed currently.

The most involved and complex avenue for future work is for the learner to be able to utilize the full decision trees developed for Rainbow [9]. These trees combine large numbers of smaller repairs into a comprehensive strategy of repair. These mixes of repair have considerable power to fix problems in a system, so bootstrapping them as well would be a remarkably efficient step forward. Such an approach provides an extremely powerful method for systems to be able to learn about themselves. As long as a learner is not entangled with the specific details of an underlying system, it will have a flexibility of application that makes it very useful. The learner that I present in this paper requires just a few abstractions from the system. Any system can produce these abstractions and thereby allow the learner to apply.

Looking at the larger picture, this kind of learner is a small part of an eventual goal of being able to trust autonomic systems to run entirely on their own. Ideally, developers will keep such systems abstract so that users can integrate any newly created system with the autonomic learner and have the learner discover how to run it effectively. The major limiting factor for many algorithms that attempt to tackle the generalizability problems associated with much of autonomic research is that of scalability. It is simple to imagine defining states for a single continuous variable (e.g., server load) by finding ranges where certain results seem more likely to occur. However, one has to add another variable like bandwidth. Not only is there an entirely different scale to discover for bandwidth, but also a scale to handle the interaction between bandwidth and load. Each new property greatly increases complexity. Most solutions simply do not scale.

The problems associated with complexity and scale are very important to solve for an ideal autonomous system. However, in the meantime, one can attack the problem by coming up with approximations that simplify things for a learner. Assuming approximations like the state collapse I used for this learner allows for a great deal of progress because it assumes that some other agent will handle complexity and scalability. In this case, that agent is a human who knows something about what makes a state interesting. Should we arrive at an algorithm that can make that decision on its own, our learners should be abstract enough to be able to just use that algorithm as the agent instead of a human. Maintaining the independence of a learner with respect to the underlying system while focusing improvements on the real-world modeling of the learner will preserve its generalizability. Making the real-world improvements will increase the power and applicability of this kind of learner to the problems we face today. Doing both lays the groundwork for integrating future advances while providing useful results in the present.

REFERENCES

[1] Bantz, D. F. et. al. Autonomic Personal Computing. *IBM Systems Journal*, **42**(1), 165-176, 2003.

[2] Norman, D. A, Ortony, A., & Russell, D. M. Affect and Machine Design: Lessons for the Development of Cognitive Machines. *IBM Systems Journal*, 42(1), 38-44, 2003.

[3] Candea, G., Kiciman, E., Kawamoto, S., & Fox, A. Autonomous Recovery in Compnentized Internet Applications. *Cluster Computing,* **9**(2), 175-190, 2006

[4] Valetto, G., Kaiser, G., & Phung, D. A Uniform Programming Abstraction for Effecting Autonomic Adaptations onto Software Systems. Proceedings of the International Conference on Autonomic Computing (ICAC '05), Seattle, WA, June 13-16, 2005

[5] Cheng, S, et. al. Rainbow: Architecture-based Self-adaptation with Reusable Infrastructure. Proceedings of the International Conference on Autonomic Computing (ICAC '04), New York, NY, May 17-18, 2004.

[6] Watkins, C. J. (1989). Models of Delayed Reinforcement Learning. Ph.D. thesis, Psychology Department, Cambridge University, Cambridge, United Kingdom.

[7] Sutton, R.S. (1996) Generalization in reinforcement learning: Successful examples using sparse coarse coding. In Advances in Neural Information Processing Systems 8, Cambridge, MA: MIT Press.

[8] Littman, M. L., et al. Reinforcement Learning for Autonomic Network Repair. Proceedings of the International Conference on Autonomic Computing (ICAC '04), New York, NY, May 17-18, 2004.

[9] Cheng, S., Garlan, D., & Schmerl, B. Architecture-Based Self-Adaptation in the Presence of Multiple Adapatations. To appear in

Proceedings of the ICSE 2006 Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), Shanghai, China, May 21-22, 2006.