Handling Uncertainty in Autonomic Systems

Shang-Wen Cheng
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
+1-412-567-0426
chengs@cmu.edu

David Garlan
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
+1-412-268-5056
garlan+@cs.cmu.edu

ABSTRACT

Autonomic, or self-adaptive, systems are increasingly important. One of the most prevalent techniques is to adopt a control systems view of the solution: adding a runtime, separate control unit that monitors and adapts the system under consideration. A problem with this paradigm for system engineering is that the control and the system are loosely coupled, introducing a variety of sources of uncertainty. In this paper we describe three specific sources of uncertainty, and briefly explain how we address those in the Rainbow Project.

Categories and Subject Descriptors

D.2.m [**Software Engineering**]: Miscellaneous – *architecture-based self-adaptive system, autonomic computing.*

General Terms

Management, Design, Languages

Keywords

Rainbow, self-adaptation, Stitch, strategy, tactic, uncertainty.

1. INTRODUCTION

Autonomic, or self-adaptive, systems are increasingly important. One of the most prevalent techniques is to adopt a control systems view of the solution: adding a runtime, separate control unit that monitors and adapts the system under consideration (the target).

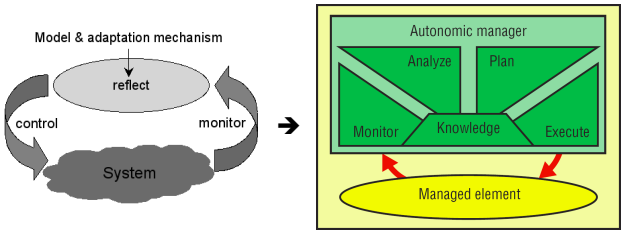

Figure 1. Control systems paradigm of self-adaptation

ASE'07, November 5, 2007, Atlanta, Georgia, USA.

The separate, external control typically maintains one or more explicit models of the running system and uses these as a basis for configuring, repairing, and optimizing the system. A recent branch of work suggests an architectural model of the software as a useful basis for dynamically changing the system [3][6][7]. An architectural model can provide a global system perspective, expose important properties and constraints, and support problem analysis. It therefore allows adaptation to be done in a principled and possibly automated fashion. Using the architectural model as a basis to monitor and adapt a running system is known as *architecture-based self-adaptation*.

A problem with this paradigm for system engineering is that the controller and the system are loosely coupled, introducing a variety of sources of uncertainty. In this paper we describe three specific sources of uncertainty and briefly explain how we address those in the Rainbow Project.

2. RAINBOW APPROACH

Architecture-based feedback control raises the challenges of developing and using a control model, getting information out of the target system, interpreting system states, reasoning about actions to take, making decisions, and effecting changes on the system, as well as the overall challenge of engineering such a system cost-effectively.

Rainbow provides an engineering approach and a framework of mechanisms to *monitor* system and environment states, manage and use an architectural model to *detect* problems, determine problem state and *decide* on a course of action, and *act* on the adaptation, corresponding to the *MAPE* loop of IBM autonomic computing shown in Figure 1.

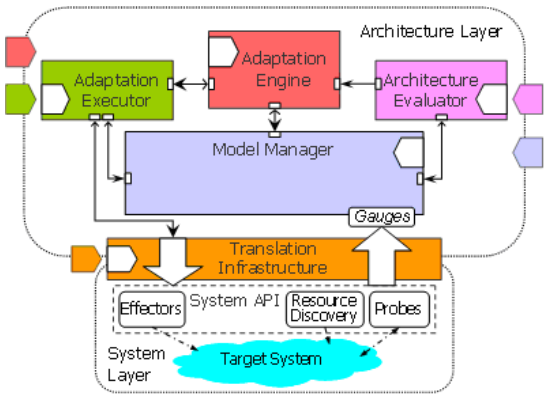

Figure 2. Rainbow self-adaptation framework

Illustrated in Figure 2, the Rainbow self-adaptation framework [2] functions as follows. Monitoring mechanisms, consisting of various kinds of *probes*, observe the running system. *Gauges* relate observations to properties in the architecture model managed by the Model Manager. When new updates occur, the Architecture Evaluator checks the model to ensure that the system is operating within an envelope of acceptable range, determined by constraints on the architecture. If the evaluation determines that the system is not operating within an acceptable range, the Adaptation Engine is triggered to determine the appropriate adaptation action to take. The Executor carries out the chosen action on the running system via system-level *effectors*.

A critical piece of the Rainbow framework, and more generally autonomic systems with a similar adaptation cycle, is the decision component. Although there are many ways to design this decision piece, our experience indicates that we can draw inspiration from system administrators (sys-admin), who are experts at handling adaptations of systems they manage. By considering the mental model, knowledge, and cognitive tasks of the sys-admin in keeping a system operational, we can emulate the decision-making process as well as formulate a set of first-class concepts important to self-adaptation.

We have thus developed a self-adaptation language called *Stitch*, for which we derive the ontology from system administration tasks and base the underlying formalism on utility theory [1]. Table 1 summarizes the self-adaptation concepts that comprise the Stitch self-adaptation language. Of note are three core concepts. The *operator* represents a single action taken by the sys-admin on the system, such as starting or killing a process. The *tactic* represents a script of such actions, and is more formally defined with conditions of applicability, as well as intended effects on the target system. The *strategy* represents a collection of possible actions that can be executed in response to the same system condition and that target the same quality objective. More formally, a strategy is a tree of tactics with root and branch conditions. No system observation occurs during the execution of a tactic, while intermediate observation of the system, through the architectural model, is possible at every branching, or decision, point within a strategy.

In such an adaptation framework, the management components, in particular the monitoring and effecting mechanisms, are loosely coupled from the target system by design. The major advantage provided by loose coupling is the ability to target the Rainbow framework to a variety of system types and adaptation needs. By the same token, it has a major disadvantage: the introduction of a variety of sources of uncertainty. We now examine these sources.

Table 1. Stitch self-adaptation concepts

Self-adaptation concept	Sys-admin inspiration
Operator	Single action on system, e.g., kill proc
Tactic	Script of actions, e.g., addWebServer
Strategy	Course of possible actions for problem
Strategy selection	Decision path taken
Objective + Preference	Business objective & op requirements
Tactic meta-information	Factors considered in action choice

3. ADAPTATION UNCERTAINTIES

In a Rainbow-like autonomic system, the steps to detect the existence of a problem, to decide on adaptation actions, and to carry out the actions, each contribute a source of uncertainty.

3.1 Problem-State Identification

Let us assume that system probes are well-designed and appropriately deployed to provide accurate readings, so that we avoid sensor measurement problems that one often encounters in control systems. The first challenging source of uncertainty lies in knowing when there is a problem in the system.

A common and effective way to identify a problematic system state is to define bounds for specific system properties, such as CPU load above 75% or end-to-end latency above X number of seconds. Even for quality attributes that do not have straightforward numeric measurements, such as intrusion or other security concerns, it is often possible to derive a numeric or discrete state representation (e.g., probability of intrusion) for which bounds can be defined.

Once bounds have been defined for relevant system properties, identifying problem state is still *not* straightforward simply due to the transient or stochastic nature of many system properties, such as CPU load or network latency. For instance, it is possible for the CPU load reading of a server to spike to 90% for a very brief moment. The fact that it is above the 75% threshold at that moment is not necessarily indicative of a problem that requires system adaptation. On the other hand, if the CPU load has been steadily rising over the previous minute, even if it has not yet triggered the 75% threshold, it may still indicate a problem worthy of system adaptation.

In Rainbow, we use two techniques to address the uncertainties with problem-state identification. First, to counter *transient* or stochastic properties, we use gauges that apply a moving-average filter to the probed values. A moving average computes the next value, $y(k+1)$, by summing the product of a constant, c ($0 < c < 1$), with the previous average, and the product of $1-c$ times the new value, i.e., $y(k+1) = cy(k) + (1-c)u(k)$. Thus, moving average requires knowing only one previous value. The moving-average filter is a well-known technique in control system to smooth measurement readings without affecting the "gain," that is, without shifting the system response from its intended target [4]. Averaging has the advantage of smoothing out the "sudden jumps," or outlier values, but also the disadvantage of increasing reaction time. The choice of c allows the engineer to decide whether to give more weight to the historical, or the new, value.

As another technique, ongoing work in our research group augments architectural description with probabilistic distributions beyond the basic Gaussian. This technique would enable engineers to explicitly characterize an expected distribution for a system property—e.g., Gaussian, exponential, or Weibull—and develop gauges that compare observations against the expected distribution, compute errors, and react to alarming error trends.[1]

[1] See paper on "Augmenting Architectural Modeling to Cope with Uncertainty" in this workshop by Celiku, Garlan, and Schmerl.

Second, to handle *trends* in readings that indicate problematic conditions, we rely on gauges with predictive capabilities, drawing on resource prediction work that enables anticipatory system adaptation, e.g., [8][2]. Poladian's prediction framework provides runtime, multi-step ahead predictions for (a) short-term trends based on recent history, (b) long-term seasonal trends, and (c) bounding trends.

3.2 Strategy Selection

Once a problem has been identified, the second source of uncertainty lies in determining which repair action to pick. When a sys-admin determines a course of action, he bases his decision on his knowledge and experiences given his observations of the system. He most likely considers various factors of cost and benefit when he makes his choice. After he chooses his adaptation strategy, he may change his mind based on changes in system condition. Also, he may perform more detailed inquiries into particular states of the system before committing to a strategy, such as deciding whether a sudden increase in network traffic is due to legitimate requests or a denial-of-service attack.

In the context of Rainbow, this step consists of selecting, from a potentially large adaptation repertoire, the adaptation strategy that best fits the current conditions of the system. To emulate the sys-admin's decision process, all of the considerations described above would manifest themselves as uncertainties. In particular, the cost and effect of actions are uncertain, conditions of applicability are uncertain, and outcomes of actions are uncertain. Nonetheless, Rainbow must select a strategy while accounting for all of these uncertainties up-front.

```
strategy SimpleReduceResponseTime() {
  boolean c0 = responseTime() > RespTimeLimit;

  t0: (c0) -> switchToTextualContent() {
    t1: (#[prob{t1}] success @[1000/*ms*/])
        -> done ;
    t2: (#[prob{t2}] c0 @[2000/*ms*/])
        -> enlistServer(1) {
      t2a: (success @[1000/*ms*/]) -> done ;
    }
  }
}
```

Figure 3. An illustrative strategy

To address the many facets of uncertainty in strategy selection, we provide for a rich notion of *strategy* in the Stitch language. Figure 3 illustrates a simple strategy to reduce the response time of a news provider system. In brief, this strategy defines a Boolean condition, *c0*, and uses the up-to-date value of that condition to determine how to traverse the tree. The strategy tree consists of four nodes: a root node at *t0* with enabling condition *c0*; two nodes, *t1* and *t2*, branching from *t0* with corresponding conditions, and a single node *t2a* branching from *t2*. The strategy uses two tactics, one to switch server content quality to "textual," and the other to enlist more servers. The keyword **success** is shorthand for a Boolean that conjoins two predicates: (a) the condition that enabled the parent node is *false*, and (b) the expected effect of the parent-node tactic is *true*. The keyword **done**, if reached, tells Rainbow to observe for and expect successful execution of the strategy.

To narrow the scope of uncertainty, Rainbow first evaluates the root-node condition of all available strategies and filters out the inapplicable strategies (i.e., root-node condition yields *false*). Note that although not explicitly illustrated here, the root-node condition could capture more elaborate predicates, for instance, to prescribe the type of system in which a strategy could apply.

Next, Rainbow computes a scalar score for each strategy based on its aggregate costs and benefits, and selects the strategy with the highest expected utility (in a probabilistic sense). Costs and benefits are defined at the tactic level as cost and effect elements of the tactic *attribute vector* (details in [1]). Each of the attribute elements must correspond to a utility dimension (e.g., response time, quality, cost, disruption) with a predefined value function that maps an attribute value to a utility value between 0 and 1. Essentially, the set of utility dimensions and their value functions correspond to the experiences that the sys-admin draws from to assess goodness and badness of respective system attributes.

One more aspect of uncertainty remains: the outcome of a tactic is uncertain at strategy-selection time, so how does one determine which branch to take, and consequently, which tactic attribute vector to use for scoring the strategy? In short, we estimate the likelihood that the known possible effects of a tactic would be observed once that tactic is executed. Then, we assign to each strategy tree branch a probability that the branch condition evaluates to true. For example, the strategy in Figure 3 suggests that tactic `switchToTextualContent()` has two outcomes, with a probability of *prob*{*t1*} being the *t1* condition, and a probability of *prob*{*t2*} being the *t2* condition. The branch probabilities are used to discount the expected contribution of each attribute vector element from tactics below that branch. Note that these branch probabilities need not be fixed, but can be updated dynamically by learners that track the outcomes of adaptations.

To score a strategy, we must first compute its *aggregate* attribute vector at the root node of that strategy. Note that this vector can be constructed from individual scalar attribute values. We define a simple recursive algorithm for computing this aggregate attribute vector; however, we must consider cost and effects separately since cost accumulates down the tree while effects are only applicable at the leaf node. Given a strategy with the root tactic X, children tactics A, B, etc., with corresponding probabilities pA, pB, etc., we can recursively compute:

$$E_{\mathbf{A}\_cost}(\mathrm{X}) = Agg\_AV_{cost}(\mathrm{X}) = tAV_{cost}(\mathrm{X}) + (\mathrm{pA} \times Agg\_AV_{cost}(\mathrm{A}) + \mathrm{pB} \times Agg\_AV_{cost}(\mathrm{B}) + \ldots)$$

$$E_{\mathbf{A}\_effect}(\mathrm{X}) = Agg\_AV_{effect}(\mathrm{X}) = \mathrm{pA} \times Agg\_AV_{effect}(\mathrm{A}) + \mathrm{pB} \times Agg\_AV_{effect}(\mathrm{B}) + \ldots$$

A strategy score is a fraction between 0 and 1, and it gives some confidence that, given the current set of system conditions, the strategy has that much expected likelihood of bringing the system back within normal operational bounds.

3.3 Strategy Success or Failure

The third challenging uncertainty lies in knowing if a given strategy has succeeded or failed in terms of achieving its intended effects. Naturally, as soon as we observe the set of conditions that

[2] See also the paper in this workshop by Vahe Poladian.

match the intended strategy effect, we would know the strategy succeeded. What is uncertain is the length of the time window within which to observe that effect. If the window is too short, Rainbow could misjudge the outcome of a strategy and potentially exhibit the problematic adaptation behavior of oscillating between two competing strategies.

To address this uncertainty, we import the notion of *settling time* from control theory. In essence, settling time gives us an indication of how long to wait before we could expect to observe the steady-state effects of an executed strategy. Assuming each strategy could be specified with a settling time, after executing a strategy, Rainbow would know approximately how long to wait to observe its effects. If the settling-time technique proves effective, then it might enable us to achieve the desirable hysteresis—lag between making changes—in triggering adaptations.

Recall that a strategy is composed of a tree of tactics. Uncertainty in the observation time window also applies after the execution of each tactic. Rainbow addresses this uncertainty by allowing the engineer to explicitly capture an estimated time window of observation, as illustrated by the millisecond value in each of the branches, *t1* and *t2*, in Figure 3. The time window for *t1* indicates that if the **success** condition holds true within 1000 milliseconds after the execution of `switchToTextualContent()` in *t0*, then the overall branching condition of *t1* holds true. The time window for *t2* indicates, on the other hand, that if condition *c0* holds true within 2000 milliseconds after the execution of `switchToTextualContent()` in *t0*, then the overall branching condition of *t2* holds true.

Although this technique does not eliminate the uncertainty, explicitly estimating and capturing observation time windows at the tactic-choice level increases confidence that the expected conditions are observed within sufficiently allotted time.

4. RELATED WORK

Similar to Rainbow, related researches on self-adaptive systems generally assume a control loop of some form to monitor and control a target system [6][7][9]. The Architecture Evolution Framework at UCI dynamically evolves systems using a monitoring and execution loop controlled by a planning loop [3]. IBM's Autonomic Computing initiative outlines an architecture where a *computing element* is managed by an *autonomic manager* that monitors the element, analyzes it and its environment for potential problems, plans actions, and executes changes in a control loop [5]. Rainbow's architecture corresponds closely with that presented in IBM's autonomic computing blueprint, particularly with respect to the MAPE loop. We contend that similar sources and issues of uncertainty exist in these autonomic or self-adaptive systems.

5. CONCLUSION

In this paper, we have briefly introduced Rainbow as a representative approach to engineer self-adaptive systems and identified three challenging sources of uncertainty in such systems—when identifying a system problem, when selecting the adaptation strategy, and when determining whether a strategy effected changes on the system successfully—and explained how we address them. We plan to continue present evaluation of these techniques as well as find more techniques, particularly for determining whether a strategy succeeded or failed, to enhance the self-adaptive capabilities of the Rainbow framework.

6. ACKNOWLEDGMENTS

This research was supported by DARPA under grants N66001-99-2-8918 and F30602-00-2-0616, by the US Army Research Office (ARO) under grant numbers DAAD19-02-1-0389 ("Perpetually Available and Secure Information Systems") to Carnegie Mellon University's CyLab and DAAD19-01-1-0485, and the National Science Foundation under Grant No. 0205266. The views and conclusions described here are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA, the ARO, NSF, the US government, or any other entity.

7. REFERENCES

[1] Cheng, S-W., Garlan, D., and Schmerl, B. Architecture-based Self-Adaptation in the Presence of Multiple Objectives. *Proc. of ICSE 2006 Workshop on Software Engineering for Adaptive and Self-Managing Systems* (SEAMS'06), Shanghai, China, May 21-22, 2006.

[2] Cheng, S-W., Huang, A-C., Garlan, D., Schmerl, B., and Steenkiste, P. Rainbow: architecture-based self-adaptation with reusable infrastructure. *IEEE Computer, 37, 10,* October 2004.

[3] Dashofy, E. M., van der Hoek, A., and Taylor, R. N. Towards architecture-based self-healing systems. Garlan, D., Kramer, J., and Wolf, A., eds., *Proceedings of the First ACM SIGSOFT Workshop on Self-Healing Systems* (WOSS'02), (New York, NY, USA, Nov 18–19, 2002). ACM Press, 2002, 21–26.

[4] Hellerstein, J. L., Diao, Y., Parekh, S., Tilbury, D. M. *Feedback Control of Computing Systems*, IEEE Press, John Wiley & Sons, Inc., NJ, 2004.

[5] Kephart, J. O. and Chess, D. M. The vision of autonomic computing. *IEEE Computer*, 36, 1, Jan 2003.

[6] Magee, J., Dulay, N., Eisenbach, S., and Kramer, J. Specifying Distributed Software Architectures. Schafer, W. and Botella, P., eds, *Proc. of 5th European Software Engineering Conference* (ESEC'95) (Sitges, Spain, Sep 26, 1995). Springer-Verlag, Berlin, 1995, 137–153.

[7] Oreizy, P., Gorlick, M. M., Taylor, R. N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D. S., and Wolf, A. L. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems, 14, 3*, May–Jun 1999, 54–62.

[8] Poladian, V., Garlan, D., Shaw, M., Schmerl, B., and Sousa, J. P. Leveraging Resource Prediction for Anticipatory Dynamic Configuration. In *Proc. of the First IEEE International Conference on Self-Adaptive and Self-Organizing Systems* (SASO-2007), Jul 2007.

[9] Wolf, A. L., Heimbigner, D., Carzaniga, A., Anderson, K. M., and Ryan, N.. Achieving survivability of complex and dynamic systems with the Willow framework. *Proceedings of the Working Conference on Complex and Dynamic Systems Architecture*, Brisbane, Australia, Dec 12-14, 2001.