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.

Architecture-based Self-Adaptation in the Presence of Multiple Objectives

Shang-Wen Cheng David Garlan Bradley Schmerl
School of Computer Science, Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, PA 15213, USA
zensoul@cs.cmu.edu david.garlan@cs.cmu.edu schmerl@cs.cmu.edu

ABSTRACT

In the world of autonomic computing, the ultimate aim is to automate human tasks in system management to achieve high-level stakeholder objectives. One common approach is to capture and represent human expertise in a form executable by a computer. Techniques to capture such expertise in programs, scripts, or rule sets are effective to an extent. However, they are often incapable of expressing the necessary adaptation expertise and emulating the subtleties of trade-offs in high-level decision making. In this paper, we propose a new language of adaptation that is sufficiently expressive to capture the subtleties of choice, deriving its ontology from system administration tasks and its underlying formalism from utility theory.

Categories and Subject Descriptors

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

General Terms

Management, Performance, Design, Security, Languages

Keywords

Self-adaptation, repair language, strategy, tactic, choice, utility, preference, trade-off

1. INTRODUCTION

In the world of autonomic computing, the ultimate aim is to automate human tasks in system management to achieve high-level stakeholder objectives. One common approach is to capture and represent human expertise in a computer-executable form, such as a shell script, a Java program, or even a set of rules in a rule engine like JESS. The script or program may be deployed as part of the managed system or with an external mechanism that monitors and adapts the system, and the prescribed adaptation is carried out in response to conditions on the managed system.

In previous work [2][3], we developed a software architecture-based self-adaptation framework, called Rainbow, which uses external mechanisms and a software architecture model to monitor a managed system, detect problems, determine a course of action, and carry out the adaptation actions. Rainbow relies on existing capabilities in the managed system to allow system states to be extracted and changes to be effected. Using a software architecture model allows the *adaptation engineer* to abstract away unnecessary details of the managed system. The software architecture of a system is the structure of its components, their interrelationships, and principles and guidelines governing their design and evolution over time [4]. It allows an engineer to obtain a global system perspective, explicitly capture system properties and constraints, and leverage existing, proven architectural analysis techniques to determine problems and remedies.

Furthermore, using architecture modeling at both the design- and run-time allows us to generalize the adaptation mechanisms and expertise of Rainbow across classes of systems that share similar design assumptions and constraints. That is, we can leverage the notion of software architectural *styles* [8] to reuse adaptation mechanisms across different system instances of the same style [2]. This approach potentially empowers software engineers to apply and tailor Rainbow to a wide variety of existing systems without significant redevelopment effort.

Our initial Rainbow prototypes managed the target system based on prescribed scripts. From that experience, we have discovered that capturing human expertise as programs or scripts can be quite effective to adapt a system for a single quality dimension, such as performance or security. Furthermore, combined with the architecture model, an adaptation script can be analyzed against the style of the managed system, i.e., the design constraints that establish the envelope of allowed structure and behavior, to ensure that the script correctly modifies the system.

However, when more than one dimension must be considered for adaptation, representing the choices and trade-offs in a program, or alternatively, as expert system rules, becomes unwieldy, if not impractical. Not only does the number of cases grow intractable, but updating and maintaining consistency between the trade-off preferences quickly becomes unmanageable. In short, a programmatic or rule-based approach is insufficient for expressing the necessary adaptation expertise and emulating the subtleties of trade-off decisions in the presence of multiple objectives.

In this paper, we propose a new language of adaptation for which we derive the ontology from system administration tasks and base the underlying formalism on utility theory. We hypothesize and illustrate with example that this language is sufficiently expressive for adaptation expertise and overcomes the subtleties of high-level human adaptation decision.

2. MOTIVATING NEWS WEBSITE

To help motivate the adaptation language, consider a fictitious news website, Z.com, which serves graphical news content to its customers. As illustrated in Figure 1 below, Z.com uses a load balancer to balance requests across a pool of replicated servers, the size of which is dynamically adjusted to balance server utilization with service response time. In other words, from a system management point of view, Z.com has the objective to serve news contents to its customers within a reasonable response time range while keeping the cost of the server pool within its operating budget. From time to time, due to highly popular events, Z.com experiences spikes in news requests that it cannot serve adequately, even at maximum pool size. To prevent losing customers, Z.com opts to serve minimal textual content during such peak times in lieu of providing zero service to its customers.

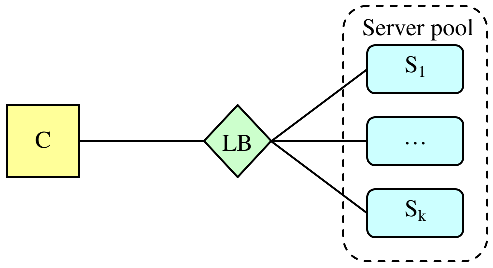

Figure 1. System architecture of Z.com

In this scenario, we learn from the expert system administrator of Z.com that two kinds of actions (a) and (b) are possible when the system comes under high load. The administrator may (a) increase the server pool size until a cost-determined maximum is reached, at which point the administrator would (b) switch the servers to serve textual content. If the system load drops, the administrator may (b) switch the servers back to graphical mode to make customers happy in combination with (a) reducing the pool size to reduce operating cost. The decision to adjust pool size and switch content mode is determined by observing overall high average response time and peak server load. We want to help Z.com automate its server management to adjust its server pool size as well as to switch content between graphical and textual mode.

Note that in reality, a news site like CNN.com may already support some level of automated adaptation. However, automating decisions that *trade off* between *multiple objectives* to adapt a system is still not supported in most systems today. For instance, while it is possible to automate adaptations on performance concerns (e.g., load balancers), it is much harder to automate adaptations that address potentially conflicting qualities, such as performance and security. This work is an important step in that direction: to allow automation of adaptations that must balance multiple objectives.

To keep the Z.com example manageable, we will make a few simplifying assumptions. First, to reduce the problem state space dramatically, we assume that every replicated server incurs the unit cost *c* and serves textual or graphical content uniformly. Since we care about the customers receiving timely news content, the most direct way to detect problems is to measure the average request-response time that customers experience. We assume that the system can be probed for the average millisecond response time of service to client requests and that the server infrastructure enables us to start, stop, and switch the content mode of the servers within a reasonably short time.

To reason about adaptation, we will distinguish between *observable* versus *actionable* states. Observable states are, namely, states of the system that we can observe using some system probing technology. The observable state space is usually infinite and intractable to adaptation analysis. We need to reduce this space to make reasoning tractable. State reduction functions are used to simplify the observable states into actionable states to facilitate decision-making for adaptation.

In the Z.com example with the state simplification assumptions, the observable state space has four components—response time in milliseconds (R: $\mathbb{N}$), server pool size (P: $\mathbb{N}$), server cost in dollars (C: $\mathbb{N}$), and server content mode as graphics or text (S: {g,t})—yielding an infinite state space of $\mathbb{N}\times\mathbb{N}\times\mathbb{N}\times\{g,t\}$. Assume that, from interacting with the system administration experts, we have developed the following reduction functions:

$f_R(r, LO\_T, HI\_T) : \mathbb{N}\times\{LO\_T\}\times\{HI\_T\}\rightarrow\{\text{low, med, high}\} =$
$\{\ low \text{ if } r<LO\_T,\ med \text{ if } LO\_T \le r \le HI\_T,\ high \text{ if } r>HI\_T\ \}$

$f_C(p, c, BUD\_T) : \mathbb{N}\times\mathbb{N}\times\{BUD\_T\}\rightarrow\{\text{ under\_bud, over\_bud }\} =$
$\{\ under\_bud \text{ if } pc \le BUD\_T,\ over\_bud \text{ if } pc > BUD\_T\ \}$

$f_S(s) : \{g, t\}\rightarrow\{g, t\}$ (*function defined for completeness*)

$f_{ALL}(r, p, c, s, LO\_T, HI\_T, BUD\_T) :$
$\mathbb{N}\times\mathbb{N}\times\mathbb{N}\times\{g, t\}\times\{LO\_T\}\times\{HI\_T\}\times\{BUD\_T\}\rightarrow$
$\{low, med, high\}\times\{under\_bud, over\_bud\}\times\{g, t\}$

The purpose of a reduction function is to reduce a continuous range in the state space to a small, discrete set using some qualitative judgment. In this case, the infinite range of response time is reduced to three states of *low*, *medium*, and *high*; the cost and server pool size states are reduced to two states of *under_bud* (within budget) and *over_bud* (over budget). Intuitively, a human observer evaluating a property of the system would apply such a judgment to reduce the amount of information to consider. Thus, f_{ALL} reduces the infinite observable state space of Z.com to an actionable state space of twelve states. In reality, we would expect many more actionable states, just not infinite.

This small set of actionable states allows us to better understand the conditions for adaptation. In the Z.com example, four adaptations are possible: (1) Switch the server content mode from graphical to textual, (2) switch the server content mode from textual to graphical, (3) increment the server pool size, and (4) decrement the server pool size. The application of an adaptation depends not only on the condition of the system, but also on high-level objectives from the system stakeholders.

We consider three qualities of concern, or objectives, to the stakeholders: (A) response time experienced by the customers, (B) news content quality, and (C) server provision cost to Z.com. Note that, at some level, these are competing goals, and an adaptation choice must make an appropriate trade-off between these three objectives, based on predefined stakeholder preferences. As a further simplifying assumption, we treat stakeholder preferences over the objectives to be static once defined in the adaptation framework.

Given the actionable state space and these three objectives, we can reason about adaptations for Z.com. When the response time is high, objective A suggests that Z.com will increment its server pool size (3) if it is within budget; otherwise, Z.com will switch the servers to textual mode (1). When the response time is low, objective C suggests that Z.com will decrement its server pool size (4) if it is near budget limit; objective B suggests that Z.com will switch the servers to graphical mode (2) if they are not already in that mode. When the response time is in the medium range, objective B suggests that Z.com will switch to graphical mode if it the mode is textual, while the server pool size may either be incremented to decrease response time or decremented to reduce cost.

We will now use the described Z.com example to motivate and explain the concepts in our adaptation language.

3. LANGUAGE FOR SELF-ADAPTATION

To realize automation of system management tasks, we require an adaptation language with the expressiveness to represent human expertise and the flexibility and robustness to capture complex and potentially dynamic preferences. In this section, we describe the ontology of concepts for such a language and how we derive it by observing system administration tasks.

3.1 What System Administrators Do

Imagine a system administrator, Carol, who is tasked to manage the infrastructure at Z.com. It is useful to consider the knowledge, model, and cognitive tasks involved for Carol to keep Z.com operational. Under normal conditions, faced with the large number of system conditions to check, Carol monitors the average request-response time as the first indicator of problem. At any time, Carol has various levels of control into the infrastructure, such as starting or killing a process, to cause changes and bring the system back to normal. Over time, Carol has packaged frequently performed sequences of actions into scripts for convenience and maintenance. Finally, Carol manages the system infrastructure with the objectives of the company in mind, namely the three objectives we described above for Z.com.

Let us assume at some point that Carol notices the average request-response time at Z.com rising above a high threshold. Note that although we have focused on the performance quality in our example so far, security compromise and other factors could also contribute to a rise in system response time. Looking at the system states, Carol determines from her experience that the rise in response time should be handled as a performance problem. She quickly checks the loads on the servers, sees that they are highly loaded, and concludes that there is a popular news event. She switches all the servers to textual mode so that the system can recover and fulfill pending news requests from customers. Meanwhile, she considers the size of the server pool and the company operating budget to determine that she can increase the pool size to serve more requests. With this increase, it turns out that she can switch the servers back to graphical mode. A successful adaptation resolves the initially observed problem.

Notice that Carol observes the system to determine the next step after taking each action. At each decision point, Carol considers several factors, including how much resource an action would require, how long the action would take, the extent to which the action would affect the system, and the relative ranking of the action against other alternatives.

Realistically, system administrators can encounter any number of challenging situations. For instance, not every action will succeed. One may fail in mid-execution, while another may complete without achieving the expected result. As another example, while resolving performance problem A, a new security compromise problem B may preempt it. Finally, multiple problems might occur simultaneously, in which case the system administrator would first need to classify and prioritize them on the fly.

Although we may still be unable to automate the more challenging situations, we can hope to automate the more routine tasks and reduce the heavy monitoring overhead of system administration.

3.2 Concepts of Self-Adaptation

Based on the description of Carol's administration of Z.com, we can derive and generalize a number of concepts for system self-adaptation.

The individual actions that Carol carries out on the system are the building blocks of change. In architecture-based self-adaptation, the action corresponds to the *architectural operator* provided by the architectural style of the target system. For example, an action or architectural operator might be to kill a running process.

```
tactic switchToTextualContent() {
  condition {
    responseTime() > HI_T
  }
  action {
    svrs = {select s : ServerT | s.active()};
    forall s : svrs {
      s.setTextualMode(true);
    }
  }
  effect {
    responseTime() < HI_T and
    forall s : ServerT | s.active() •
      s.isTextualMode()
  }
}
```

Figure 2. An example *tactic*.

A script of actions that Carol develops would correspond to a sequence of architectural operators, which form the adaptation *tactic*. Switching the server content mode from graphical to textual would be considered a tactic. More formally, a tactic as illustrated in Figure 2 is defined in three parts: the conditions of applicability, a sequence of actions, and a set of intended effects after execution. The execution semantics of a tactic is as follows: The adaptation engine chooses a tactic at each cycle, and the tactic is executed from beginning to end without new observations of the target system. A tactic completes successfully if no exception occurs during execution and all the intended effects are achieved; otherwise, the tactic may be considered to have failed.

The mental model of all the possible actions that Carol can take corresponds to a set of *strategies*. Intuitively, a strategy embodies a flow of actions over a sizable time frame and scope, with intermediate decision points, to fix a type of system problems, while a tactic embodies a small sequence of actions to fix a specific problem in a localized part of the system. More formally, a strategy is defined as a tree of tactics that tackles common quality issues, with conditions describing each branch. Intermediate system observation occurs after each tactic is executed in order to decide what successive branch to take in the strategy tree. This branch decision is called *tactic matching*. Though we will not discuss it in depth in this paper, we derive the syntax of a strategy from Dijkstra's Guarded Command Language [7], which provides an intuitive condition-action construct with repetition capability, supporting our defined notion of the strategy.

Carol's decision to treat an observed system condition as a performance versus a security problem corresponds to *strategy selection*. For humans, this process combines heuristics, trade-offs, and experience. To emulate and automate this process, the adaptation framework must be able to capture sufficient heuristics and trade-off information to allow the strategies to be scored and compared. The company objectives and operational requirements that Carol considers are important to inform this selection process, and these correspond to the *adaptation objectives* and *adaptation utility preferences* of the stakeholders. The cost and benefit factors that Carol considers while choosing actions correspond to the *tactic meta-information*. Combined with probability information associated with the branches in the strategy tree, tactic meta-information associated with the tactic nodes in the strategy tree allows the computation of aggregate strategy scores. This is explained further in Section 3.3.

Failure handling is necessary when something goes wrong during the execution of an action. *Preemption* occurs when a more important problem arises that takes priority over the problem currently under investigation.

Together, the architectural operator, tactic, strategy, strategy selection, tactic matching, adaptation objective, adaptation utility preference, tactic meta-information, failure handling, and preemption form the core concepts of self-adaptation. The *operator*, *tactic*, and *strategy* form the basic ontology of the adaptation language.

3.3 FORMALISM OF SELECTION

To overcome the complexity and subtleties of human decision when selecting an adaptation, we developed a formalism based on utility theory to evaluate strategies during an adaptation cycle and select the best strategy that balances multiple objectives. In the description below, formal expressions appear between square brackets following a descriptive prose.

The formalism is developed using set theory, and starts with the managed system. We model the managed system as a labeled Kripke structure, also known as a *doubly labeled transition system*. A Kripke structure is a type of nondeterministic finite state machine used in model checking to represent the behavior of a system. It defines a graph whose nodes represent the reachable states of the system, whose edges are labeled and represent state transitions, and where a labeling function maps each node to a set of atomic propositions that hold true in the corresponding state.

More formally, let *Act* be a countable set of action symbols and let *AP* be a set of atomic propositions, i.e., Boolean expressions over variables, constants, and predicate symbols. The system can be represented as a 4-tuple $\boldsymbol{M} = (S, I, R, L)$ with signature, i.e., available operations, (Act, AP), where S is a countable set of states, $I \subseteq S$ is a set of initial states, $R \subseteq S \times Act \times S$ is a transition relation that allows for reflexive transitions [$\forall s \in S \bullet \exists a \in Act \bullet (s,a,s) \in R$], and $L\colon S \rightarrow 2^{AP}$ is a labeling or interpretation function.

Using a utility function *reachables*: $S \times Act \rightarrow \mathbb{P}S$ that returns a set of states immediately reachable from a state s given an action a, we define a function *prob*: $S \times Act \times S \rightarrow [0,1]$ that evaluates the probability of an action $a \in Act$ transitioning the system from a state $s \in S$ to any of the possible resulting states $s' \in S$. First we require the elements in the domain of *prob*() to be in the transition relation R [$\forall (s,a,s'){:}S \times Act \times S \mid (s,a,s') \in \textbf{dom}\ prob \bullet (s,a,s') \in R$]. We then require the sum of all probabilities of the same non-deterministic transition a from the same state to add up to 1 [$\forall s{:}S \bullet \forall a{:}Act \bullet \sum_{s' \in reachables(s,a)} prob(s, a, s') == 1$].

A tactic [$t \in T$] is defined as a sequence of operations (basic type [OPERATION]) and corresponds to an action label in $\boldsymbol{M}$ [$T \subseteq Act$], but not all transitions in $\boldsymbol{M}$ necessarily correspond to a tactic. We define a unique *null tactic* as a null sequence. We assume that no spontaneous (τ) transition occurs in the system during an adaptation process, that is, all transitions in a Kripke model result from some adaptation actions.

Each tactic is associated with an attribute vector of n elements, which include both cost attributes such as resource consumed, and effect attributes such as reduction in response time. Thus, the attribute vector describes the expected cost and effect incurred and delivered by a tactic, respectively, when it completes. We define AV to be a set of such attribute vectors and define a function to retrieve the attribute vector of a tactic [$tAV\colon T \rightarrow AV$].

A preference vector is defined in correspondence to the tactic attribute vector, using utility curves to map the value of each attribute to a score in the range [0,1]. A utility function then computes the weighted sum of the utilities across the attribute vector, yielding a scalar utility value [$U_{pref}\colon AV \rightarrow \mathbb{N}$]. The utility curves capture the extent to which the users will be happy with particular values of each attribute. The overall utility function represents the relative priority of the attributes over one another.

A strategy [$g \in G$] is defined as a tree over a set of vertices V, corresponding to the set of tactics [$V \subseteq T$], and a set of paired vertices E, where $pred(e)$ gives the predecessor vertex of $e \in G$ and $succ(e)$ gives the successor vertex of e, with a condition function [$C\colon E \rightarrow \mathbb{P}AP$] and a probability function [$P\colon E \rightarrow [0,1]$] over E. In relation to the managed system $\boldsymbol{M}$, the condition on every branch can be mapped to corresponding state propositions [$\forall e \in E \bullet \exists a1, a2 \in Act \mid a1 == pred(e) \wedge a2 == succ(e) \bullet \exists s1, s2, s3 \in S \mid (s1, a1, s2) \in R \wedge (s2, a2, s3) \in R \bullet C(e) \subseteq L(s2)$]. To allow an aggregate expected value to be computed meaningfully over the entire tree, the probabilities at each branch level must sum to 1 [$\forall v \in V \bullet sum\{e \in E \mid pred(e) == v \bullet P(e)\} == 1$]. Finally, to model unexpected conditions under which no tactic at a particular branch level might apply, we define for every branch level a null tactic branch that is applicable any time no condition indicated by $C()$ matches [$\forall v \in V \bullet \exists e \in E \mid pred(e) == v \bullet C(e) == \{\} \wedge succ(e) == \langle\rangle \wedge \forall e' \in E \mid pred(e') == v \wedge e != e' \bullet C(e') != \{\} \wedge succ(e') != \langle\rangle$].

Using the probabilities and the attribute vector of each tactic in a strategy tree, we can compute the aggregate attribute vector, $E_{\boldsymbol{A}}()$ [$E_{\boldsymbol{A}}\colon G \rightarrow AV$], over the strategy at the root tactic. The algorithm to compute $E_{\boldsymbol{A}}(g)$ consists of two parts, one for the cost-based attribute elements, and the other for the effect-based attribute elements. We concatenate the vectors $E_{\boldsymbol{A}\_cost}$(X) and $E_{\boldsymbol{A}\_effect}$(X) to form $E_{\boldsymbol{A}}$(X).

Given a strategy with the root tactic X, its children A, B, etc., with corresponding probabilities pA, pB, etc., we recursively compute:

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

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

Finally, we can select from the set of available strategies one that yields the maximum utility value [$\max_{g\in G}\{U_{pref}(E_{\boldsymbol{A}}(g))\}$].

By formalizing the notions of strategy and tactic in terms of an underlying finite state model, we form a clean mathematical model to allow analysis against an abstract managed system. Assuming that the managed system is modeled with sufficient fidelity, we can check whether defined tactics correspond properly to transition *actions* in the Kripke model and ensure that branching conditions of defined strategies derive from corresponding state propositions.

More importantly, the utility theoretic basis of strategy selection allows tactics with different attributes and strategies that tackle different problems to be compared on even footing. Utility theory also gives us the assurance that trade-off is dynamically computable between possibly conflicting interests, provided we can elicit and capture preferences adequately from the stakeholders as well as estimate the probabilities on strategy branches with accuracy. Preference elicitation and probability estimation are two hypotheses that warrant proof through future case studies.

In the next section, we illustrate the strategy selection using this language formalism.

4. ADAPTATION SELECTION EXAMPLE

In this section, we will illustrate the adaptation selection using the Z.com example system. We start with the three high-level, potentially competing objectives and derive from those a set of utility preferences. We illustrate the definition of adaptation tactics with their attribute vectors and demonstrate strategy selection using the defined utility preferences.

The two primary groups of stakeholders in the Z.com example are the customers and the provider. The customers care about quick response time of their news requests and high content quality (i.e., graphical over textual). The provider, while aware of the customers' quality preferences, is constrained by the infrastructure cost to provide the service. To summarize the objectives:

1. Response time: low, medium, or high
2. Quality: graphical or textual
3. Budget: under or over

Note that these three stakeholder objectives suggest three corresponding attributes that are important to select an adaptation. In addition, since response time is one of the objective attributes, and it is affected by the amount of time required to complete a tactic, we also need to consider *disruption* as a cost attribute. We will use an ordinal scale of 1 to 5 to express degree of disruption.

Given our understanding of stakeholder objectives, we can derive a simple set of utility preferences over these four attributes:

1. Response time: 1 if low, 0.5 if medium, 0 if high
2. Quality: 1 if graphical, 0.5 if unchanged, 0 if textual
3. Budget: 1 if under or unchanged, 0 if over
4. Disruption: 1 if 1, 0.75 if 2, 0.5 if 3, 0.25 if 4, 0 if 5

Furthermore, we assign relative weights to these four attributes to enable the evaluation of an overall utility value. Let's assume that Z.com considers response time the most important, followed by budget, then content quality, and finally disruption. This yields a relative weight of 0.4 for response time, 0.3 for budget, 0.2 for content quality, and 0.1 for disruption.

As described in the scenario, four adaptations are possible and can be fulfilled with three tactics, one of which is shown in Figure 2. The *switchToTextualContent*() tactic uniformly switches the server content mode from graphical to textual. A corresponding tactic *switchToGraphicalContent*() achieves the opposite effect. An *adjustServerPoolSize*(int) tactic, not shown here, increments or decrements the server pool size by an integral count.

Associated with each of these three tactics is an attribute vector, each consisting of the four previously described attributes: [disruption, response, quality, and budget].

- *switchToTextualContent*()
  [disruption: 3; response: low;
  quality: textual; budget: unchanged]
- *switchToGraphicalContent*()
  [disruption: 3; response: medium;
  quality: graphical; budget: unchanged]
- *adjustServerPoolSize*(int kdelta)
  [disruption: 1; response: low if *kdelta>4*, medium if
  0≤kdelta≤4, high if *kdelta<0*; quality: unchanged;
  budget: under if (*k*+*kdelta*)**c* < *BUD_T*, else over]

For space reasons, we will simplify the illustration of strategy selection by defining two placeholder strategies, each consisting of the first and the third tactic. This eliminates an additional step to calculate the aggregate attribute vectors and focuses our discussion on the attribute- and utility-based strategy selection.

```
strategy SwitchToTextualContent() {
  t0: (responseTime() > Resp_Time_Threshold)
    -> switchToTextualContent(m) ;
  do {
    t1: (responseTime() <
          Resp_Time_Threshold) -> done ;
  }
}

strategy AdjustServerPoolSize(int kdelta) {
  t0: (responseTime() > Resp_Time_Threshold)
    -> adjustServerPoolSize(kdelta) ;
  do {
    t1: (responseTime() <
          Resp_Time_Threshold) -> done ;
  }
}
```

Figure 3. Two simple placeholder strategies.

Figure 3 shows the two placeholder strategies, each defined using the respective tactic. Each strategy starts out with the root tactic *t0*, which labels a condition-action pair, followed by a *do* block that defines the next level of tactics. In this simple example, the next level is simply the strategy completion case followed by the terminating keyword, **done**.

Let's assume that Z.com hits a peak load period, and the system state falls into an actionable state in which the response time is high, the infrastructure cost is under budget, and the content mode is graphical. In this case, both strategies are applicable, one to change the content mode to textual, and the other to increase the size of the server pool. So we need to score the strategies to determine which one is most appropriate given the stakeholder utility preferences.

Given the specified tactic attribute vectors, the two strategies have aggregate attribute vectors as follows:

- *SwitchToTextualContent* () [disruption: 3, response: low, quality: textual, budget: unchanged]
- *AdjustServerPoolSize* (5) [disruption: 1, response: medium, low: unchanged, budget: over]

Applying the weighted utility evaluation over the attributes of these two strategies results in the following:

- *SwitchToTextualContent* ():
  U = 0.1(0.5) + 0.4(1) + 0.2(0) + 0.3(1) = 0.75
- *AdjustServerPoolSize* (5)
  U = 0.1(1) + 0.4(1) + 0.2(0.5) + 0.3(0) = 0.60

The utility scores indicate *SwitchToTextualContent*() as the better adaptation strategy, given the current system conditions. Note that if Z.com attributed a lower weight to budget, or higher weight to disruption, or swapped the importance of disruption versus budget, then the other strategy would score higher.

Using utility evaluation, we can essentially choose a strategy by considering four dimensions and accounting for trade-offs across those using additional input of user preferences over outcomes. Although this example shows simple binary or ternary preference utility functions, one can define much more complicated utility curves and benefit from this computational technique of selection.

5. RELATED WORK

Our Rainbow approach consists of a framework that monitors and manages the target system in an adaptation cycle. It maintains the software architecture of the target system as the run-time analysis model. The adaptation language proposed in this paper allows adaptation actions and pertinent decision criteria to be represented in a form that the framework can carry out to manage the target system. This section discusses related work in these three areas.

The control loop paradigm of adaptation is not unique to Rainbow. Related researches on self-healing systems generally assume a control loop of some form to monitor and control a target system [11][12][14]. 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 [10]. The Architecture Evolution Framework at UCI dynamically evolves systems using a monitoring and execution loop controlled by a planning loop [6].

The use of external mechanisms and software architecture model to dynamically monitor and adapt a running system—i.e., architecture-based self-adaptation—is also not unique to Rainbow. A collection of recent work focuses on the use of specific architectural styles (together with their associated ADLs and toolsets) to support architecture-based self-adaptation. For example, Taylor and colleagues support architecture-based run-time software evolution using hierarchical publish-subscribe style via C2 [6][12]. Gorlick and colleagues support continuous observation and dynamic rearrangement using data-flow style via Weaves [9]. Magee and colleagues use Darwin's bi-directional communication links in a proposed distributed self-organizing system where components coordinate toward a common architectural structure [11]. In contrast, Rainbow provides reusable infrastructures generalized across multiple architectural styles, which can then be tailored to specific classes of systems.

A recent body of work, such as Plastik [1], complements our Rainbow approach by combining an architecture description language with a reflective infrastructure to support the specification of dynamic change. The capabilities of dynamic adaptation in such an approach are potentially as flexible as supportable by the integrated ADL and run-time framework, but such work will still need a way to represent preference and trade-off information to enable adaptation choices across multiple objectives.

A few related efforts have influenced or inspired the development of our adaptation language. Expert systems work gave rise to the important condition-action constructs found in our strategy specification. Poladian and colleagues argued a case for multi-dimensional utility analysis because converting all costs to a common currency was problematic [13]. We borrowed from this work in our language to support analysis of choice based on multi-dimensional adaptation attributes. Finally, policy languages, such as Ponder [5], have recently been developed to support the specification of management policies for distributed systems and networks management. Ponder can capture roles and relationships of entities in a system, specify security policies, and even support service related policies. However, policy specifications do not currently capture explicit preference and trade-off information to support high-level decision of choices.

6. DISCUSSIONS AND CONCLUSION

By observing commonly performed system administration tasks, we have extracted a minimal set of concepts—*operator*, *tactic*, *strategy*—and thus the basic ontology, for an adaptation language that holds the promise of automating human tasks in system management. Together with the concepts of strategy selection, tactic matching, adaptation objective, adaptation utility preference, tactic meta-information, failure handling, and preemption, the adaptation language we have developed has the potential expressiveness to represent human expertise and the flexibility to make use of dynamic preferences.

Using utility evaluation that incorporates additional user input of preferences over outcomes, we can effectively choose a strategy by making trade-offs across multiple dimensions. The explicit specification of objective attributes and enumeration of preferences and relative weights over those attributes not only allow fine-grained control over selection outcomes, but also provide traceability of selection decision via a quantitative framework.

One issue of note is the apparently large amount of information to elicit from the experts and involved parties of the managed system: the utility curves, weights, attributes, and probabilities. We observe that system administrators already have to process a large amount of information when making decisions. We argue that our efforts simplify the administrator's job by giving structure to the large quantity of information, providing placeholders for them in our framework, and allowing the information to be supplied incrementally to achieve management automation.

A few concepts in the adaptation language require further work to flesh out. In particular, it is unclear what the best way is to handle failure during adaptation execution, or whether it needs to be dealt with at all if we assumed a continuous adaptation cycle of monitor and control. Clearly, a proper treatment of failure must ensure that the adaptation framework can recognize what failure state it is in and recover from that failure.

Secondly, it is unclear how preemption should be handled. More generally, when one or more additional problems arise in the middle of a previous adaptation in progress, how does the adaptation framework determine whether it is a new problem, or more manifestations of the existing problem? Furthermore, preemption implies priority, which would require constructs in the language to specify problem priority. Thirdly, although the concepts of strategy and tactic seem intuitively separable, from the illustration, the astute reader might have raised the same doubt about whether a formal distinction between the two is necessary. We are still working to resolve this question.

Most important in our future work, we need to perform case studies to demonstrate the expressiveness of the adaptation language, the flexibility to capture preferences, and the effectiveness of the utility-based strategy selection to emulate human decisions and trade-offs.

7. ACKNOWLEDGMENTS

This work has been informed by many discussions with George Fairbanks, who helped us to crystallize the news site example, and Vahe Poladian, who was instrumental in clarifying the language formalism based on utility theory.

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 NASA High Dependability Computing Program under cooperative agreement NCC-2-1298. 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, NASA, the US government, or any other entity.

8. REFERENCES

[1] Batista, T., Joolia, A., and Coulson, G. Managing dynamic reconfiguration in component-based systems. Morrison, R. and Oquendo, F., eds., *Proceedings of the 2nd Workshop on Software Architecture* (EWSA2) (Pisa, Italy, June 13–14, 2005). Springer, Berlin, 2005, 1–17.

[2] Cheng, S-W., Garlan, D., Schmerl, B., Sousa, J. P., Spitznagel, B., and Steenkiste, P. Using architectural style as a basis for self-repair. Bosch, J., Gentleman, M., Hofmeister, C., and Kuusela, J., eds., *Proceedings of the 3rd Working IEEE/IFIP Conference on Software Architecture* (WICSA3) (Montréal, Québec, Canada, August 25-30, 2002). Kluwer Academic Publishers, 2002, 45–59.

[3] 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.

[4] Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., and Stafford, J., eds. *Documenting Software Architecture: Views and Beyond*. The SEI Series in Software Engineering. Pearson Education, Inc., 2003.

[5] Damianou, N., Dulay, N., Lupu, E., and Sloman, M. The Ponder Specification Language. *Proceedings of Workshop on Policies for Distributed Systems and Networks* (Policy2001), HP Labs, Bristol, January 2001, 29–31.

[6] 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, November 18–19, 2002). ACM Press, 2002, 21–26.

[7] Dijkstra, E. *A Discipline of Programming*, Prentice, 1976.

[8] Garlan, D., Allen, R. J., and Ockerbloom, J. Exploiting style in architectural design. *Proceedings of SIGSOFT'94 symposium on the Foundations of Software Engineering*, New Orleans, LA, USA, December 1994.

[9] Gorlick, M. M. and Razouk, R. R. Using Weaves for software construction and analysis. *Proceedings of the 13th International Conference of Software Engineering* (ICSE-13) (Los Alamitos, CA, USA, May 1991). IEEE Computer Society Press, 1991, 23–34.

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

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

[12] 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–June 1999, 54–62.

[13] Poladian, V., Butler, S., Shaw, M., and Garlan, D. Time is not money: the case for multi-dimensional accounting in value-based software engineering. *Proceedings of the 5th Int'l Workshop on Economics Driven Software Engineering Research* (EDSER-5), Portland, OR, USA, May 2003.

[14] 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, December 12–14, 2001.