1-2012

Stitch: A language for architecture-based self-adaptation

Shang-Wen Cheng
Carnegie Mellon University

David Garlan
Carnegie Mellon University

Published In
Journal of Systems and Software, 85, 12, 2860-2875.

Submitted for publication.

Stitch: A Language for Architecture-Based Self-Adaptation

S.-W. Cheng*, D. Garlan, B. Schmerl

School of Computer Science, Carnegie Mellon University,
5000 Forbes Ave., Pittsburgh, PA, 15213 USA

Abstract

Requirements for high availability in computing systems today demand that systems be self-adaptive to maintain expected qualities-of-service in the presence of system faults, variable environmental conditions, and changing user requirements. Autonomic computing tackles the challenge of automating tasks that humans would otherwise have to perform to achieve this goal. However, existing approaches to autonomic computing lack the ability to capture routine human repair tasks in a way that takes into account the business context that humans use in selecting an appropriate form of adaptation, while dealing with timing delays and uncertainties in outcome of repair actions. In this article, we present *Stitch*, a language for representing repair strategies within the context of an architecture-based self-adaptation framework. Stitch supports the explicit representation of repair decision trees together with the ability to express business objectives, allowing a self-adaptive system to select a strategy that has optimal utility in a given context, even in the presence of potential timing delays and outcome uncertainty.

Keywords: Rainbow, self-adaptation, strategy, tactic, uncertainty, utility

1. Introduction

High availability has become an increasingly important quality attribute in computing systems ranging from critical infrastructures to daily business operations. Traditionally, high availability has been achieved in one of two

*Corresponding author, now working at the Jet Propulsion Laboratory
Email addresses: chengs@cmu.edu (S.-W. Cheng), garlan@cs.cmu.edu (D. Garlan), schmerl@cs.cmu.edu (B. Schmerl)

ways: (a) through failure-handling code embedded within the system, and (b) through human oversight and intervention.

In the first case, self-adaptation capabilities typically use mechanisms such as program exceptions and network time-outs to trigger repairs. Such low-level mechanisms are good for detecting problems quickly and providing application-specific responses, but they lack a global system perspective, making it difficult for the repair code to identify the source of a systemic problem or detect softer anomalies such as trends in performance degradation. Furthermore, adaptation logic is dispersed throughout the implementation, making it costly to modify and maintain, challenging to analyze, and infeasible to reuse.

In the second case, human operators monitor a system and make adjustments when they detect problems, or, more generally, observe opportunities to improve the performance of the system. (This activity is often enabled by system infrastructure monitoring tools such as Microsoft Operations Manager [1]). While humans are better at understanding the overall problem context than computers, human operators are prone to long reaction time, fatigue, errors, and varying and potentially inconsistent expertise. Indeed, industry data indicate that the cost of ownership of IT systems attributable to managing a system ranges from 70–90 cents per dollar [2, 3, 4] and that 20%–50% of system outages are due to operator error [5].

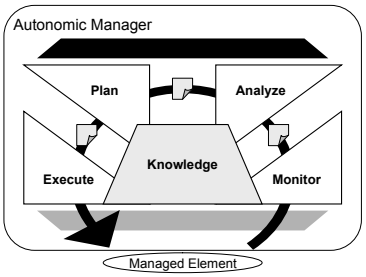

Figure 1: IBM MAPE Loop

In recent years, the emergence of Autonomic Computing [3] offers an alternative approach, where self-adaptation is designed independently of, and external to, the target system in order to automate tasks that humans would otherwise perform. Shown in Fig. 1, the IBM Autonomic Computing MAPE control loop embodies this approach, to monitor, analyze, plan, and execute changes for self-management [6].

A key issue in this approach is how to *program* the policies for doing adaptation: Can one define adaptation strategies to capture what humans do? What would that language look like? What kinds of special language features are required to support such self-adaptation?

In principle, a variety of ways exists to program adaptation, ranging from general-purpose programming languages (e.g., Java) to languages and techniques from the AI community to express complex dynamic reasoning (e.g., Markov Decision Processes and Q-learning) [7]. In this article, we focus on

an important class of policies and strategies, namely those carried out by system administrators (sys-admins) that are routine but complex. Our goal is to provide a language that can capture the key ingredients for decision making in this domain, including complex decision paths that take into consideration how effective previously executed steps have been in order to determine the next step, ways to evaluate the utility of a strategy in a given context so that the best strategy can be selected, and mechanisms to account for asynchrony, uncertainty, and timing delays inherent in externalized control.

Specifically, we propose a language for expressing adaptation strategies, called *Stitch*, which builds on an architecture-based approach to self-adaptation. Key features of this language are:

- definition of adaptation strategies using a control-theoretic point of view in which system feedback and dynamically-updated models (here, architectural models) are used to determine the next course of action;
- explicit representation of quality-of-service (QoS) objectives and priorities for capturing business context;
- the ability to calculate the best strategy in a given problem scenario, where best is determined in a utility-theoretic way based on business and system context, as well as prior history of strategy outcomes; and
- explicit representation of uncertainties in adaptation outcome and timing delays.

In Section 2 we set the context for this work with a brief overview of software architecture-based self-adaptation . We then consider the requirements for an appropriate adaptation language in Section 3. Next we describe Stitch in Section 4, illustrate the expressiveness of this language for representing realistic adaptation expertise and supporting multi-objective trade-offs with a full example in Section 5, and present evidence of Stitch's expressiveness for adaptation concerns in Section 6. In Section 7 we describe related work. Finally, we conclude with a discussion of the strengths and weaknesses of Stitch and consider future work in Section 8.

2. Context of Self-Adaptation

As noted earlier, autonomic computing is typically based on a MAPE-like externalized control regime in which systems are dynamically monitored to update control models. These models are analyzed to decide if conditions warrant self-adaptation. If so, an adaptation strategy is determined. The

strategy is then executed using adaptation *hooks* into the running system. There are many variations on this theme, depending on the nature of the monitoring mechanisms, the kinds of models that are maintained in the control layer, whether strategies are selected from a fixed set or created on-the-fly, and the kinds of adaptation interface that a system provides.

One particularly important class of self-adaptation approach relies primarily on architectural models of the system in the control layer [8, 9, 10]. An architectural model provides a high-level view of a system as a collection of components and connectors, annotated with properties that indicate component and system attributes such as reliability, performance, security, etc. [11]. Such models are particularly attractive for self-adaptation since they allow one to detect when a system is failing to meet its objectives, either through hard component failures or through aberrations in softer dynamic QoS behavior [12].

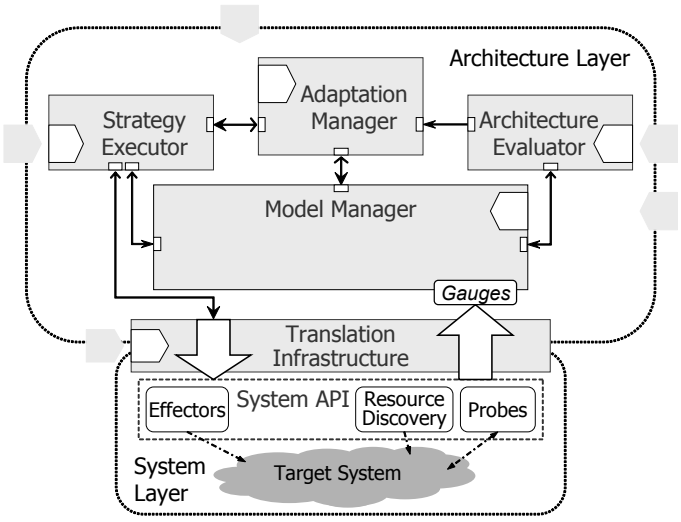

Figure 2: The Rainbow Framework

Rainbow is a particular instance of an architecture-based self-adaptation framework. As pictured in Fig. 2, and described in detail elsewhere [13, 14, 15], Rainbow uses *Probes* to monitor the target system and *Gauges* to aggregate and abstract monitored information to update its control models (maintained with the *Model Manager*). These models consist of system architecture and environment models that are used to reason about the target system and its execution context. It uses an *Architecture Evaluator* to detect when a target-system is in a state suitable for repair, an *Adaptation*

Manager to select an appropriate repair strategy, and a *Strategy Executor* to carry out the actions in the strategy, along with appropriate infrastructure for effecting changes in the target system.

Repair strategies, housed within the Adaptation Manager, are written in Stitch, the subject of this article. As we detail later, Stitch takes advantage of the overall framework in several ways. Most importantly, it uses the state of the architecture and environment models to determine both (a) conditions of applicability for selecting an appropriate repair strategy from a fixed repertoire, and also (b) the outcome of an intermediate step in the execution of a strategy in order to choose the next step.

While Stitch relies on the Rainbow infrastructure for its execution, the design of the language goes far beyond this particular embodiment of autonomic computing. The ideas behind Stitch could be applied directly to virtually any architecture-based adaptive approach, and, with minor modifications, more generally to most existing autonomic approaches.

3. Requirements for a Self-Adaptation Language

System adaptations can be viewed along a spectrum of complexity. At one end of the spectrum are highly routine operations, such as rebooting a crashed server, which can be easily automated and require minimal decision-making skills. At the other end of the spectrum are operations that require complex, domain-specific, context-sensitive and dynamically changing cognitive processes, such as directing the trajectory of a Mars rover in order to avoid some newly-encountered obstacle. In between these extremes is a large and important space of human-assisted adaptive control activities that are routine, but context-dependent, and involve dynamic decisions to select an appropriate repair and to carry it out. These represent a sweet spot for autonomic computing, as they are typically not automated today, but in principle could be. This is the domain of adaptation that Stitch addresses.

3.1. A Motivating Example: Znn.com

To make this class of adaptation concrete, and to illustrate the kinds of requirements that are thereby implied for representing repair strategies, consider an example of a web-based news provider called Znn.com. Modeled after typical infrastructure for news websites like CNN.com, Znn.com has a tiered architecture consisting of a set of application servers that serve content from backend databases to clients via front-end presentation logic. As illustrated in Fig. 3, Znn.com uses a load balancer (lbproxy) to balance

requests across a pool of replicated servers, the size of which can be manually adjusted to balance server utilization against service response time. In the figure, the dashed connectors and components represent those that are available in the pool, but not yet utilized by the load balancer. A total of four servers can be used to balance server usage. A set of client processes makes stateless content requests from the servers. In response a server delivers static files (e.g., images and videos), as well as dynamic content (e.g., news populated from periodically-updated sources).[1]

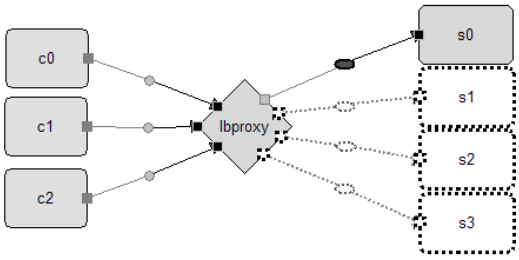

Figure 3: The Znn.com system architecture (dashed lines represent available but inactive elements)

For a system like Znn.com there are a number of business concerns that determine the system's QoS objectives. First is *response time*: response latency for client requests should be kept low. Second is *quality*: ideally content is served in full fidelity, including video, images, etc., although sometimes it may be acceptable to deliver degraded content (e.g., text only). Third is *cost*: to keep operating costs low, it is important to minimize the number of active servers. As is typical with complex system design, these qualities are not independent. For instance, performance can be improved, but only at the cost of a larger server pool and lower fidelity.

In the normal course of operations, there are a variety of situations that may impact the achievement of quality objectives. Servers may crash. Load balancers may crash. High demand for news items may overload the system. Low demands on a slow day may leave servers under-utilized. To handle such situations, systems like Znn.com typically provide several mechanisms for human operators to adapt them. These include the ability to dynamically reboot servers and load balancers, the ability to bring servers on- or off-line, and the ability to reduce the fidelity of content provided by the servers. To assist the operator in knowing when to perform such operations, such systems also include various monitoring mechanisms, such as the ability to display server status or response latency for client requests.

[1] To keep the example simple here we have not included the databases in the model.

3.2. The Nature of System Administration Tasks

Imagine a sys-admin, Sam, who manages the Znn.com infrastructure. Suppose that a system monitor indicates that average request-response latency is exceeding some threshold (say, 2 seconds). What does Sam have to do? First, Sam needs to decide on a course of action. Since such problems are typical, it is likely that Sam has a repertoire of strategies that he uses. For example, one strategy might be to bring new servers on-line until enough have been activated to improve performance. Another strategy might be to reduce the fidelity for some or all of the clients.

Importantly, choice of strategy is highly context-dependent. For instance, it makes no sense to adopt a strategy of adding servers if the pool of free servers has been used up. Moreover, business priorities play a role. Even if spare servers do exist, Sam may choose not to add more of them in order to keep cost of operations low. Furthermore, Sam may have to balance short-term and long-term effects: for example, rebooting a set of poorly-performing servers may actually degrade performance in the short term, but lead to overall higher performance in the end.

Having chosen a strategy, Sam needs to carry it out. In most cases the process of doing this is not a simple atomic operation, but rather a decision process in which certain actions (e.g., adding a server) are attempted and the results of that action are used to determine what to do next (e.g., adding another server). Note that in doing this Sam will need to exercise judgment to figure out how long to wait to observe the effects of one action, since most system adaptation actions take time to accomplish and for their effects to propagate.

Finally, as Sam gains experience with various strategies, he may learn which of them tend to work better and in which situations. That will allow him to adapt his strategy selection process over time.

3.3. Implications for a Strategy Definition Language

The nature of system administration tasks, sketched above (and substantiated in more detail in Section 6), frames a set of requirements for a strategy definition language.

First, it must support the definition of adaptation decision processes in which the choice of the next action may depend on the outcome of previous actions. Each such action may involve a set of lower-level activities (often implemented today as execution scripts that use configuration operators provided by the system). As we will see in the next section, Stitch addresses this requirement by defining *strategies* as decision trees in which each node

is the application of a change-producing action (termed a *tactic*), and future actions are guarded by conditions that can depend on the state of the system, as reflected through a dynamically-updated architectural model.

Second, the evaluation of the outcome of a tactic executed as part of a strategy must take into account potential delays in observing the result of that tactic. This is because of the asynchronous nature of an externalized control model, and the fact that system changes take time to achieve their effects. Stitch addresses this requirement by allowing a strategy writer to indicate a *delay window* within which the effect is expected to be observed.

Third, it must be possible to define the conditions under which it makes sense to consider a particular strategy. In Stitch this is accomplished by including a dynamically-evaluated *applicability condition* with each strategy.

Fourth, given a set of applicable strategies it must be possible to determine which is the *best* one to execute. The notion of best will depend on the current context, the business priorities and objectives of the organization hosting the system, and the relative costs and benefits that each strategy has to offer. Such priorities are typically multi-dimensional, requiring tradeoffs between qualities such as performance, cost, and fidelity. Stitch addresses this requirement through the specification of business priorities in terms of a *multi-dimensional utility space,* together with functions that assign values to quality levels within those dimensions. Further, each tactic is associated with an *impact vector,* which determines its expected contribution to the utility dimensions. Such vectors can be used to calculate the expected aggregate utility of a strategy in the current run-time context, and thereby provide a quantitative basis for strategy selection.

Fifth, past successes or failures of strategies should contribute to the evaluation process. Stitch addresses this requirement by incorporating a quality dimension that reflects past behavior, allowing overall utility calculation to take this into account when choosing a strategy.

4. Stitch

We now describe Stitch. In Section 4.1 we explain how the language is used to define adaptation *strategies* as decision trees built up from adaptation *tactics*, which are in turn defined in terms of more primitive *operators*. Next, in Section 4.2 we describe how Stitch supports the selection of the highest utility strategy from a repertoire of available strategies, taking into consideration the current system state, applicability conditions of the individual strategies, their expected costs and benefits, and prior history of

Table 1: Feature summary of the Stitch Strategy Definition Language

Stitch feature	**Support for adaptation decision processes**
operator	System-provided configuration command
tactic	Action primitive with conditions, operators, & effects
strategy	Decision tree of condition-tactic-delay nodes
quality dimensions	Business quality-of-service concerns
utility preferences	Business priorities over quality dimensions
impact vectors	Tactic cost-benefit attributes w.r.t quality dimensions
branch probabilities	Likelihood of strategy branch conditions evaluating to true
adaptation conditions	System indicators of opportunities for improvement
strategy selection	Choice of best strategy w.r.t. quality dimensions

strategy invocation. A summary of these concepts appear in Table 1. Finally, in Section 4.3 we sketch the formal execution semantics of the language. Throughout we will use the Znn.com example to illustrate the key features of the language. For reference Stitch's grammar is shown (in slightly redacted form) in Appendix A.

4.1. Specifying adaptation strategies

4.1.1. Operators

The most primitive unit of execution for an adaptation process is the *operator*. An operator represents a basic configuration command provided by the target system, and thus corresponds to a system-level effector (see Fig. 2).

In the context of Rainbow, we assume that the architectural style [11, 16] of the target system, which defines the class of models to which the target system architecture belongs, determines the operators representing available configuration changes on systems in that style. For example, in the Znn.com example the architectural style would define a vocabulary of clients, servers, databases, etc. Architectural operators on systems of this style would include stopService to terminate a running service process. Other operators include startServer and setFidelity.

In Stitch, an operator is syntactically a function. A set of such operators can be imported and then called by a tactic, as explained below.

4.1.2. Tactics

A *tactic* provides an abstraction that (a) packages operators into larger units of change to form steps of adaptation used by strategies, and (b) serves

as a logical unit for specifying the cost and benefit impact of an adaptation step with respect to the quality dimensions important to the system.

Specifically, a tactic is characterized as follows

- It encapsulates a sequence of calls to operators and serves as an action primitive for strategy definition.[2]
- It is guarded by a dynamically evaluated pre-condition that determines its applicability.
- It defines the effects that it is attempting to achieve.
- It defines an impact vector that specifies how it will affect the quality dimensions of the system (cf., Section 4.2)

During execution a tactic is given read-only access to the system model, captured at the time the tactic is invoked. This allows its internal logic to depend on the state of the system (as viewed through its architectural model), and hence respond to dynamic changes that occurred within the system up to the point at which it is invoked. Tactics are not allowed to invoke other tactics, but only operators. (We return to this restriction in Section 8.)

As an example, Fig. 4 shows a tactic to switch servers to textual mode. The module containing this tactic first imports the architectural style Znn-Fam, referred to as (T), the model instance (M), and the style operators (lines 1–3). The condition block specifies its condition of applicability: at least one client component's response time (stored in the expRspTime property) exceeds a maximum threshold (lines 6–8). The action block finds all servers that are not already in textual mode and sets them to textual mode via an imported operator with signature, setTextualMode(ServerT, boolean) (lines 9–14). The effect block specifies the expected outcome of the tactic: no Client exhibits an above-threshold response time, and all Servers should be in textual mode (lines 15–18). Constraints in condition and effect blocks can be written using a form of first-order predicate logic over properties in the architectural model (similar to the Object Constraint Language in UML).

A tactic can terminate in one of four ways. In the successful case, the condition block evaluates to true, signifying that the tactic is applicable; the

[2]In principle each operator could be wrapped as a tactic, but in practice we have found that tactics typically bundle up patterns of operations into higher-level primitives for use in strategies.

```
1  module znn.tactics;
2  import model "ZnnSys.acme" { ZnnSys as M, ZnnFam as T };
3  import op "znn.operators.ArchOps" { ArchOps as Sys };
4
5  tactic switchToTextualMode () {
6    condition {
7      exists c:T.ClientT in M.components | c.expRspTime > M.MAX_RSPTIME;
8    }
9    action {
10     svrs = { select s : T.ServerT | !s.isTextualMode };
11     for (T.ServerT s : svrs) {
12       Sys.setTextualMode(s, true);
13     }
14   }
15   effect {
16     forall c:T.ClientT in M.components | c.expRspTime ≤ M.MAX_RSPTIME;
17     forall s:T.ServerT in M.components | s.isTextualMode;
18   }
19 }
```

Figure 4: An example *tactic* switchToTextualMode

action block completes, signifying that no operators have failed; and the effect block evaluates to true, signifying that the tactic has achieved its aim. There are correspondingly three unsuccessful situations: (1) the condition block evaluates to false, signifying inapplicability; (2) the condition block evaluates to true, but the action block fails to complete, signifying some operator has failed; (3) the condition block evaluates to true, the action block completes, but the effect block evaluates to false, signifying that the tactic has not achieved its aim. We will see how tactic failure is handled in the strategy section below.

4.1.3. Strategies

A *strategy* encapsulates a dynamic adaptation process in which each step is the conditional execution of some tactic. Conditions along the path allow us to make the path sensitive to how effective the tactics have been. In Stitch, a strategy is characterized as a tree of condition-action-delay decision nodes, with explicitly defined probability for conditions and a delay time-window for observing tactic effects. A strategy also specifies an applicability condition as a predicate that is evaluated on the mode during strategy selection. Finally, it supports the specification of utility impact for its constituent tactics and thereby enables the computation of aggregate utility for strategy selection (as described in Section 4.2).

In Stitch a collection of strategies is packaged as a module. Fig. 5 shows an example of a simple module for Znn.com containing a single strategy that attempts to reduce system response time. First, the module imports (lines 2–

5) the namespace of the architecture style (T), architecture and environment (M & E) models, the tactics module defined in Fig. 4, and a library of utilities (Model) for querying the model.

```
1  module znn.strategies;
2  import model "ZnnSys.acme" { ZnnSys as M, ZnnFam as T };
3  import model "ZnnEnv.acme" { ZnnEnv as E };
4  import lib "znn.tactics";
5  import op "org.sa.rainbow.stitch.lib.*";  // Model, Set, & Util
6
7  define boolean styleApplies = Model.hasType(M,"ClientT")//.."ServerT";
8  define boolean cViolation =
9      exists c:T.ClientT in M.components | c.expRspTime > M.MAX_RSPTIME;
10
11 strategy SimpleReduceResponseTime [ styleApplies && cViolation ] {
12   define boolean hiLatency =
13     exists k:T.HttpConnT in M.connectors | k.latency > M.MAX_LATENCY;
14   define boolean hiLoad =
15     exists s:T.ServerT in M.components | s.load > M.MAX_UTIL;
16
17   t1: (#[Pr{t1}] hiLatency) -> switchToTextualMode() @[1000/*ms*/] {
18     t1a: (success) -> done ;
19   }
20   t2: (#[Pr{t2}] hiLoad) -> enlistServer(1) @[2000/*ms*/] {
21     t2a: (!hiLoad) -> done ;
22     t2b: (!success) -> do [1] t1 ;
23   }
24   t3: (default) -> fail;
25 }
```

Figure 5: An example *strategy* SimpleReduceResponseTime

Next the strategy specifies its applicability condition, which is used in strategy selection to determine whether the strategy should be considered. In this example SimpleReduceResponseTime (line 11) is defined using a set of convenience predicates styleApplies and cViolation. The predicate styleApplies (line 7) checks whether the model defines the architectural types ClientT and ServerT used in the strategy. The predicate cViolation (lines 8–9) checks for the condition that some client is experiencing above-normal response time.

The body of a strategy is modeled after Dijkstra's Guarded Command Language [17], which provides condition-action and while-loop constructs, but is augmented in several important ways, described below. In the example, the top-level condition-action blocks in the strategy body are labeled t1, t2, and t3.

The example strategy first defines two Boolean auxiliary functions, hiLatency (lines 12–13) and hiLoad (lines 14–15). In node t1 (lines 17–19), if condition hiLatency evaluates to true, then tactic switchToTextualMode is exe-

cuted (line 17 after arrow).

To account for the delay in observing the outcome of tactic execution in the system, t1 specifies a delay window of 1000 milliseconds (end of line 17). During execution the child node t1a is evaluated as soon as the tactic effect is observed or the delay window expires, whichever comes first.

Several keywords can be used within the body of a strategy to support control flow and termination: **success** is true if the tactic completes successfully; **done** terminates the strategy, signifying that the strategy has achieved its adaptation aims, whereas **fail** terminates the strategy, signifying that the adaption aims have not been achieved; the **default** keyword specifies what should happen if none of the other nodes is applicable.

In the strategy in Fig. 5, if the tactic switchToTextualMode successfully completes, then the strategy terminates with **success** (node t1a, line 18). In the unsuccessful case, since no other peer nodes match, the default branch is chosen (and the strategy fails, on line 24). If condition hiLatency evaluates to false, but hiLoad is true, then node t2 (lines 20–23) is evaluated and tactic enlistServer is executed to enlist an available server.

Again, due to asynchrony, t2 specifies up to 2000 ms for observing the tactic effect. After completion of the tactic, in the case that hiLoad is true and tactic enlistServer does not succeed, child node t2b is chosen and the **do**-repetition is evaluated, which is semantically equivalent to repeating the subtree rooted at node t1 as a child node, *t1'*, of t2b. Evaluation proceeds with the hiLatency condition of node *t1'*: if true, then the rest of the node is evaluated as described before; if false, then the implied **default** branch aborts the strategy. The [1] indicates the number of times the **do** repetition can occur within a single evaluation of this strategy if the **do** node is revisited (not illustrated by this example).

In the example strategy, if both top-level node conditions hiLatency and hiLoad are true, then one is chosen non-deterministically. If neither hiLatency nor hiLoad is true, then node t3 is chosen, and the strategy aborts.

Note that this strategy has five termination points: nodes t1a and t2a define two success points, node t3 defines an explicit failure point, and nodes t1 and t2 have two implicit child-level failure points. Thus, for nodes t1 and t2, under unsuccessful cases, or if none of the conditions apply, the default branch is chosen.

When a strategy fails, the Adaptation Manager handles any unresolved adaptation conditions in the next adaptation cycle by triggering another round of strategy selection and execution.

Although we have used a simplified strategy to illustrate Stitch syntax, Stitch is sufficiently expressive for defining more elaborate strategies. For

the Stitch grammar, see Appendix A. More complex examples from the Znn.com system can be found in Cheng's Ph.D. dissertation [18].

4.2. Strategy Selection

As we argued earlier, a key requirement for a strategy language is the ability to select a strategy that is consistent with the business objectives of the organization. In the context of Rainbow, where we work with a fixed repertoire of strategies, this amounts to identifying the strategy with the highest utility among those that are applicable in a given situation. To do this Stitch requires the specification of four kinds of information: quality dimensions, utility preferences, impact vectors, and branch probabilities. Additionally, we define a set of adaptation conditions, which serve as triggers for initiating the process of strategy selection and execution.

Quality dimensions specify the dimensions of the multi-dimensional utility space that captures business QoS concerns for the target system (e.g., system reliability, service availability, or database performance). A quality dimension provides a notion of utility for particular values of a quality attribute. Each dimension defines a utility function and maps to an architectural property monitored in the model.

Table 2: Data schema for a Utility Profile

Field Definition	**Description**	**Example**
identifier : string	A unique mnemonic	"uR"
label : string	Human-readable name	"Average Response Time"
description : string	Descriptive comment	"R, client experienced response time (ms), float arch property"
mapping : string	Monitored arch property	"ClientT.expRspTime"
utility function : linear | sigmoid | custom: $\langle(x_i, y_i)\rangle$	Type & domain-range	custom: $\langle(0,1),(500,.9),(1500,.5),(4000,0)\rangle$

Each quality dimension is captured as a *Utility Profile* using a data schema summarized in Table 2. The profile consists of an identifier, label, description, mapping to a monitored architectural property, and utility function definition. The mapping refers to a property defined in the architectural style. A utility function defines how the values of a given quality dimension correspond to *happiness*, with values in the range $[0, 1]$, where 0 is *undesirable* and 1 is *desirable*. Such utility functions can take many forms (depending on

the way in which that quality affects happiness. Stitch supports three kinds of functions: linear, sigmoid function with two defining points, and as a set of value pairs (with intermediate points linearly interpolated).

Utility preferences are used to define business priorities, or relative importance, between the quality dimensions. Following standard utility theory [19], we specify a set of weights, w_i. in the range $[0, 1]$, one for each dimension, and summing to 1. The overall utility is then given by $U = \sum_d w_d u_d$, where d is the number of quality dimensions.

Impact vectors define the expected cost and benefit of a tactic on each of the quality dimensions, and hence capture the causal relationship between an adaptation step and the quality dimensions. They are represented as a vector of dimension-value pairs, where the value component captures delta value of cost or benefit. Thus, if a target system owner defines three quality dimensions, then the attribute vector would look like $[a_1 : \Delta v_1, a_2 : \Delta v_2, a_3 : \Delta v_3]$.

For instance, tactic enlistServers for Znn.com is defined with the impact vector, $[a_R : -1000, a_F : 0, a_C : +1.00]$, which specifies that the tactic is expected to reduce the average response time by 1000, to have no impact on content fidelity, and to increase cost by 1. (The delta values are understood to have the units of the quality dimension as defined in the utility profile.)

Branch probabilities. Because there is uncertainty in whether (a) a tactic achieves its intended effect and (b) a condition is observed, we estimate the likelihood of observing the branch conditions. As we will see later, stochastic branch conditions allow an aggregate impact vector to be computed for each strategy in utility-based selection. The probabilities can be captured explicitly for each strategy node condition. The Pr{*} expression preceding each of the conditions of t1 and t2 (Fig. 5, lines 17 and 20) denotes the estimated probability that the condition will evaluate to true among the peer-node conditions. Actual probability values are defined elsewhere and not significant during strategy execution. Peer-node probabilities must sum to 1; thus, in the example, t3's probability is an implied complement.

Adaptation conditions identify system states for which adaptation should be considered, including the handling of faults, system degradation, and, more generally, opportunities for improving the target system. To automate self-adaptation, we define quantitative expressions with respect to specific quality dimensions, such as the maximum latency for the response-time dimension. Further, we specify in the architecture model a measurable threshold quantity as a system-instance architectural property and an adaptation condition as an architectural constraint, a violation of which identifies an opportunity for adaptation:

```
1  // threshold quantity (property defined in style; value, sys instance)
2      Property MAX_RSPTIME : float = 1000.0;
3  // adaptation condition (defined in the style)
4      invariant self.avg_latency < MAX_RSPTIME;
```

Once we have defined an impact vector for each tactic and specified utility preferences over the quality dimensions, the next step is to compute the expected aggregate attribute vector for each strategy. Recall that a strategy is composed of a tree of tactics, and the condition of each node is annotated with the likelihood of matching. Computing the aggregate vector consists of descending the strategy tree, unfolding **do**-repetitions, and collecting the impact vector values of each tactic.

The probabilities defined for the conditions allow the attribute values to be propagated up the strategy tree and eventually collected into an expected aggregate attribute vector. The vector values designate aggregate delta costs and benefits against the quality dimensions, so they are combined with architectural property values representing current system conditions, as identified by the mappings of corresponding quality dimensions (e.g., the exponential average of the client-experienced response time for the u_R dimension). Using the utility profiles, these aggregate values are converted to utility values per dimension, then a weighted sum is computed using the preference weights to yield an expected utility score for each strategy. Finally, the strategy with the highest score is selected.

Once a selected strategy is executed, its success or failure, i.e., *prior history*, provides additional clue to its fitness for future consideration. One can think of success history as a predefined quality attribute that the strategy writer can use, as with any other quality dimension, for strategy selection. In Stitch, one can define a *strategy-failure* utility profile to enable a basic learning feature in the Adaptation Manager to track the historical failure rate of each strategy, computed as its failure count divided by how many times it was selected, and to incorporate it as a utility component in the utility score. Prior history thus provides control over how likely a strategy, which has been prone to failure, is selected in the future.

4.3. Execution Semantics

We now sketch the operational semantics of Stitch constructs.[3] In the context of the Rainbow framework, an *adaptation cycle* repeatedly monitors the target system for adaptation conditions, and triggers the Adaptation

[3]See [18] for details of the language semantics, including its denotational semantics as a Markov Decision Process.

Manager to take appropriate actions when such conditions are detected. Specifically, the adaptation cycle proceeds as follows:

1. Detecting an *adaptation condition* triggers a round of adaptation.
2. For each strategy in the strategy repertoire, first check the strategy *applicability conditions* to filter a subset of applicable *strategies* based on current system conditions (reflected in the model), then select the best strategy from that subset by computing the expected utility of each strategy as follows:
   (a) Compute the expected aggregate impact of each strategy on each *quality dimension* using the *impact vector* specified for the tactics;
   (b) Score strategies using *utility preferences* over the dimensions; and
   (c) Select the highest-scoring strategy, G_{best}.
3. Execute G_{best} as follows:
   (a) Check its applicability condition to confirm that it still applies.
   (b) If so, find all its top-level nodes.
   (c) For each node, evaluate the condition on the incoming edge.
   (d) If none matches and no *default* branch is defined, **abort** strategy.
   (e) If more than one branch matches, pick a branch randomly.
   (f) Taking the matching branch, evaluate the node by action type:
      i. If *done*, terminate strategy with **success**.
      ii. If *fail*, **abort** strategy.
      iii. If a *do* loop, set the corresponding do counter, find the node referenced by the do label, and continue strategy execution from step 3c.
      iv. If a tactic, execute the tactic as follows:
         A. Check tactic *condition*; if applicable, continue the next step; otherwise, **abort** tactic.
         B. Execute tactic *action*; if any operator fails, **abort** tactic.
      v. Evaluate the tactic *effect* until it is observed in the target system's model, which signifies tactic **success**, or until the *delay* expires.
   (g) Find the next level nodes, and continue execution from step 3c, noting that the keyword *success* in any branch condition is synonymous with querying whether the parent-node tactic was effected successfully.

5. Znn.com Illustration

To bring together the concepts, we now illustrate the features of Stitch using the Znn.com example system. We start with the three high-level, potentially competing, objectives and specify a set of utility functions and preferences for those. We illustrate the definition of adaptation tactics with their impact vectors and demonstrate strategy selection using the defined utility preferences.

Typical of news-provider concerns, the quality objective of Znn.com is to serve news content to its customers within a reasonable response time, while keeping the cost of the server pool within a certain operating budget. From time to time, due to highly popular events, Znn.com experiences spikes in news requests that it cannot serve adequately, even at maximum pool size. To prevent losing customers, we opt to serve minimal textual contents during such peak times in lieu of providing zero service to the customers. In short, we identify three quality objectives for the system to self-adapt—(A) performance, (B) cost, and (C) content fidelity—from which we derive three quality dimensions, captured as discrete value sets:

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

We elicit from the service providers the utility values and preferences for these dimensions. In addition, since response time is affected by the amount of time required to complete a tactic, we also need to consider a fourth dimension, disruption, which should be minimized. We use an ordinal scale of 1 to 5 to express the degree of disruption. Given our understanding of the quality dimensions, we can specify discrete utility functions for these four dimensions and complete the utility profiles (*description* elided). To determine the utility preferences, assume that Znn.com considers response time the most important, followed by budget, then content quality, and finally disruption. This might yield a set of linear, relative weights as shown in Table 3.

The quality dimensions correspond to measurable properties in the target system, which has an N-tier-client-server architectural style defining the types ClientT, ServerT, ProxyT, HttpConnT. Performance analysis of Znn.com suggests that we monitor the request-response time (ClientT.experRespTime), server load (ServerT.load), and connection bandwidth of the system (HttpConnT.bandwidth). Cost analysis identifies the number of active servers as the primary contributor to cost (ServerT.cost); hence we monitor the server count.

Table 3: Znn.com utility profiles and preferences

ID	**Label** (**Mapping**)	**Utility Function**	w_i
u_R	Avg Response Time (ClientT.experRespTime)	$\langle(\text{low}, 1), (\text{medium}, 0.5), (\text{high}, 0)\rangle$	0.4
u_F	Avg Content Quality (ServerT.fidelity)	$\langle(\text{textual}, 0), (\text{multimedia}, 1)\rangle$	0.2
u_C	Avg Budget (ServerT.cost)	$\langle(\text{within}, 1), (\text{over}, 0)\rangle$	0.3
u_D	Disruption (ServerT.rejectedRequests)	$\langle(1, 1), (2, 0.75), (3, 0.5), (4, 0.25), (5, 0)\rangle$	0.1

Finally, we characterize three different levels of content fidelity (ServerT.fidelity) ranging from full multimedia to static text (high, medium, and low).

The operators ServerT.activate() and deactivate() activate and deactivate a ServerT instance, respectively. The operator ServerT.setFidelity(level:int) sets the server content fidelity to the level identified by the input parameter. Using these operators, we specified two pairs of tactics with opposing effects. One pair *enlist*s or *discharge*s *server*s while the other pair *raise*s or *lower*s the server content *fidelity*. In effect, these tactics stratify the service level of the Znn.com system into gradients to trade off the various objectives. Associated with each tactic is an impact vector, each consisting of four elements corresponding to the four previously described quality dimensions, shown in Table 4. The following example illustrates how these tactics might interact:

When response time is high, objective A (above) suggests that Znn.com should increment its server pool size (using the enlistServer tactic) if it is within budget; otherwise, Znn.com should switch the servers to textual mode (using lowerFidelity). When the response time is low, objective C suggests that Znn.com should decrement its server pool size (using dischargeServer) if it is near budget limit; objective B suggests that Znn.com should switch the servers to multimedia mode (using raiseFidelity) if they are not already in that mode. When the response time is in the normal range, objective B suggests that Znn.com should switch the servers to multimedia mode if they are currently textual, while the server pool size may either be incremented to decrease response time or decremented to reduce cost.

In our case study, we have defined several adaptation strategies from these tactics, with juxtapositions that allow system adaptation to balance the overall objectives (cf. p.109 in [18]). Here we focus our illustration on

Table 4: Znn.com tactic cost-benefit attribute vectors

Tactic	u_R	u_F	u_C	u_D
enlistServers(int Δk)	-2 steps *if* $\Delta k > 4$, -1 step *otherwise*	$+0$	$+1$ step *if* $c(k+\Delta k) \geq Th_{bud}$, $+0$ step *otherwise*	$+1$
dischargeServers(int Δk)	$+2$ steps *if* $\Delta k > 4$, $+1$ step *otherwise*	$+0$	-1 step *if* $c(k+\Delta k) < Th_{bud}$, $+0$ step *otherwise*	$+1$
lowerFidelity(int Δf)	$-\Delta f$ steps	$-\Delta f$ step	$+0$ (no change)	$+3$
raiseFidelity(int Δf)	$+\Delta f$ step	$+\Delta f$ step	$+0$	$+3$

Table 5: Znn.com utility evaluation for two applicable strategies

Strategy	u_R	u_F	u_C	u_D
	Weighted Utility Evaluation			
DropFidelityStrategy	-2⇒low	-1⇒textual	+0⇒within	3
	$U = 0.4\,(1) + 0.2\,(0) + 0.3\,(1) + 0.1\,(0.5) = 0.75$			
EnlargeServerPoolStrategy	-2⇒low	+0⇒multimedia	+1⇒over	1
	$U = 0.4\,(1) + 0.2\,(1) + 0.3\,(0) + 0.1\,(1) = 0.70$			

utility-based strategy selection—skipping the step to calculate the aggregate impact vectors—by defining two placeholder strategies: DropFidelityStrategy, which invokes the tactic lowerFidelity(-2); and EnlargeServerPoolStrategy, which invokes the tactic enlistServers(5) ($\Delta k = 5$ yields a -2-step effect on u_R).

Let us assume that Znn.com hits a peak load period, and the system state falls into a problem state in which the response time is high, the infrastructure cost is within budget, and the content mode is multimedia. 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 utility preferences. The specified tactic impact vectors would yield aggregate attribute vectors for the two strategies, which are then used to compute the weighted utility score, as shown in Table 5.

The utility scores indicate DropFidelityStrategy as the better adaptation strategy, given the current system conditions. Note that if Znn.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 have scored higher.

Using such utility-based analysis, we can choose a strategy by considering

four dimensions and accounting for trade-offs across those using the additional input of business utility preferences. Although this example shows simple utility functions with few discrete values, one can define more complex utility functions and benefit from this utility-based technique.

6. Evaluation

The design of the Stitch language was informed by our understanding of how sys-admins perform adaptive administrative tasks when they encounter system problems. We developed this insight over time from personal experiences and interactions with five system administrators. To substantiate the expressiveness Stitch and the suitability of its design, we evaluate Stitch on three fronts by showing that:

1. realistic sys-admin concerns and practices can be expressed in the language;
2. others besides the authors can capture adaptation policies in Stitch; and
3. runtime overhead of strategy selection is reasonably low.

In this section, we summarize results from a case study to express sys-admin tasks in Stitch, a preliminary set of interviews with sys-admins, and a performance evaluation of the runtime overhead of the strategy selection algorithm in Stitch. Details are found in Cheng's dissertation [18].

6.1. Real-World Adaptive Scripts in Stitch

Over the past decade, Carnegie Mellon University has invested extensive engineering efforts to improve its networking infrastructure and automate system administration, making it a prime candidate to find evidence of system adaptation processes. Of the network administrative subsystems, one is the network bandwidth enforcement (*netbwe*) subsystem, which enforces bandwidth quota per machine. The *netbwe* subsystem is a set of Perl scripts that are executed daily and gather information from sensors installed on campus routers. Using a database, *netbwe* tracks usage history, records quota violation, and tracks violation states. It interacts with another subsystem, *epidemic*, to alert offending machine owners by email, and interacts with a *netblock* component to block network access for repeat offenders. In short, *netbwe* has the monitoring, detection, decision, and action elements of a self-adaptive system. To evaluate the expressiveness of Stitch, we examined whether it could faithfully represent the adaptations contained within the Perl scripts.

Table 6: Strategies and Tactics for netbwe.

Strategies	
EscalateViolationStates	escalates violation states of machines exceeding daily quota.
RestoreViolationStates	reverses violation states of machines that pass probation.
BlockMachines	blocks network access of machines with a *ban* violation state.
ExemptMachines	adjusts violation states of machines with exemption override.
Tactics	
markViolation	marks machines with the next state of violation.
emailNotify	notifies machine owners of quota violation.
modifyMachines	modifies the violation or exemption state of machines.
netblockMachines	issues a block request on machines via *netblock*.
setExempt]	sets exempt privileges machines.

From talking with the network admins, *netbwe* has these *adaptation objectives*: (i) fair usage; (ii) system fit-for-purpose (i.e., campus researchers can perform research); and (iii) reasonable campus connectivity cost, as determined by capacity and usage. These objectives correspond roughly to *adaptation conditions* predicated on (a) bandwidth usage threshold and (b) exemptions and dated restoration of violation states.

The *netbwe* subsystem consists of 19 Perl source files with about 10k source lines of code. In particular, three frontend Perl programs and two backend Perl module subroutines comprise the core adaptation functionality, while some of the subroutines act as effectors with *operator* counterparts for the adaptation script, and the remainder serve as library utilities. The required operators are translated to effectors realized by the corresponding Perl subroutines. By analyzing the adaptation objectives in combination with the Perl subroutines, we identified four *strategies* and five *tactics*, which we were able to capture in Stitch, as summarized in Table 6. (Again for details see [18].)

This case study of CMU's network bandwidth enforcement subsystem, by identifying and extracting parallel adaptation elements from the scenario, provides evidence that Stitch is capable of representing the adaptation-oriented concerns embodied in real-world, system-administrative Perl scripts.

Furthermore, this exercise demonstrated additional benefits in representing the *netbwe* subsystem using Stitch. Stitch helps separate the concerns of adaptation, so that monitored properties, operators to change the system, and adaptation choices are not distributed throughout the management logic. In particular, adaptation choices are made prominent in strategies, rather than being buried deep inside a Perl subroutine. Finally, the distinction between strategy and tactic enables the adaptation engineer to reason about and describe the specifics of an adaptation action as an intellectually separate process from deciding when to take each action.

6.2. Interviews with System Administrators

To evaluate whether the concepts embodied in Stitch match system administrative processes, we arranged to interview sys-admins. Here we summarize results from two sys-admins, one from interview, another from self-guided decision analysis.[4]

We conducted an interview with a former Carnegie Mellon University sys-admin who described a self-adaptive scenario for managing students who abuse network bandwidth. In a network bandwidth abuse case, a student had backed up ~80 GB of his hard disk onto his server space. The sys-admin observed a spike in bandwidth usage the next day and a technician noticed that the backup tape was exhausted at around the same time. To prevent future repeat of similar situations, the sys-admin would ideally want to track disk usage by file type, such as MP3s. The sys-admin also contemplated enforcing per-user disk quota, but that policy would have unjustly prevented legitimate users from transferring large quantities of data; one solution was to enforce disk quota on only the users with excessive usage. Finally, a monitoring capability with email notification might have alerted the sys-admin to a bandwidth abuse problem much sooner than the *next day*.

Based on the description of the abuse problem and potential solutions, we were able to capture three adaptive concerns, three tactics, and an overall strategy, summarized in Table 7. Only one of the tactics was directly drawn from the interview, the others being inferred from context. Whether inferred or directly elicited, since these tactics fit the problem context, they serve our purpose to assess the expressiveness of Stitch.

The three adaptation objectives are shown below, where *cost* captures a potential for dollar value lost when executing a particular tactic.

[4] Derived from the work of Ali Almossawi, an undergraduate student with prior experience as a sys-admin, who performed a summer independent study on Rainbow under our supervision.

Table 7: Summary of White's solution strategy and tactics

[S] DealWithAbusiveUser *trigger: notable spike in bandwidth usage* *trigger: backup tape unexpectedly runs out*	
	[T] increaseServerPoolSize (User user, Host h, int n) *guard: bandwidth is high* [overhead: 0.5; ill-feeling: 1; cost: 0]
	[T] banAbusiveUser (User user) *guard: bandwidth is high* *guard: the offense has been committed n times (where n>1)* *guard: the disk space usage is high* [overhead: 1; ill-feeling: 0; cost: 1]
	[T] warnAbusiveUser (User user) *guard: bandwidth is high* *guard: the offense has been committed once before* *guard: the disk space usage is at least medium** [overhead: 1; ill-feeling: 0.5; cost: 1]

1. Overhead: 1 if low, 0.5 if medium, 0 if high
2. Ill-feeling: 1 if low, 0.5 if medium, 0 if high
3. Cost (dollars): 1 if low, 0.5 if medium, 0 if high

From our analysis of the interview, we were able to develop an architecture for this system and compose a Stitch module that corresponds closely to what is described in Table 7. Interested readers are referred to the Appendix in [18] for details. The module uses both an architectural and environment model, and is able to express predicates that reference elements simultaneously from both models (users belong to the environment). This expressiveness empowers the adaptation engineer to reason about adaptations by combining information from both the architecture and the environment.

As additional supporting evidence that others can capture their adaptation process in Stitch, we had a sys-admin who was familiar with Stitch, but not one of the authors, structure his decision-making process when managing a webserver against intrusion. He noted two primary concerns in choosing an adaptation: (a) which course of action requires the least effort and (b) how severe is the situation. To combat suspicious activity on a server, this sys-admin followed the decision process outlined below:

```
|_ Logwatch email indicates suspicious IP address or brute force
     attempt
```

```
|_ Add IP address to firewall's "deny" list
|_ Flush firewall's rules then restart it
|_ Check server logs and look up IP address to see if it has a
    history
  |_ If it does
    |_ Change superuser's password
|_ Check server logs to ensure IP address didn't gain access to the
    system
  |_ If I suspect it did
    |_ Immediately run rootkit checker
    |_ Immediately run anti-virus scanner
    |_ If for some reason I'm still suspicious
      |_ Backup all data
      |_ Reformat webserver
      |_ Restore data
```

Based on these interviews, the concerns, problems, and solutions expressed corroborate the concepts embodied in Stitch. In particular, evident in the documented responses were objectives, observable system conditions, specific actions in response to specific conditions, and, to a lesser extent, preferences. Assessed abstractly, Stitch provides the appropriate constructs and has the expressiveness for capturing an adaptive administrative strategy concisely and intuitively.

6.3. Performance of the Strategy-Selection Algorithm

A long decision-making process hinders the usefulness of a self-adaptive approach. The main decision-making process in Stitch is embodied in its utility-based strategy-selection algorithm. As described in Section 4.2, during the selection process, the Adaptation Manager traverses the tree of each applicable strategy to compute an aggregate vector of cost and benefits, then calculates its expected utility as a weighted sum across the utility dimensions. If $q =$ the number of dimensions, $n =$ the number of nodes per strategy tree, and $s =$ the number of strategies, the algorithm has a complexity of $O(nqs)$, in other words, linear with the number of strategies.

In practical usage, the size of the strategy repertoire in any particular target system is expected to range from a handful to several dozens (< 100), while the number of nodes per strategy and the size of quality dimensions would likely be no more than a dozen. For good measures, we vary the number of strategies by orders of magnitude from 10 to 10000, and the number of dimensions at 5, 10, 50, and 100. Strategy node counts range from 4 to 15 nodes. All strategies are made applicable in each run.

Using a 3.00 GHz Intel Pentium 4 machine with 2 GB of RAM, we

Table 8: Summary of performance (milliseconds) for Stitch's strategy-selection algorithm.

Strategy count s	**Quality Dimensions** q			
	5	**10**	**50**	**100**
10	23	19	17	17
100	171	173	169	167
1000	1495	1468	1465	1454
10000	13761	13742	13639	13730

performed 50 trial runs for each s-q pair[5] and measured the duration of time beginning when adaptation is triggered and ending when the Executor receives the best-scoring strategy to execute. Table 8 summarizes the data. Note that s is the primary determinant of algorithm run time, and the data confirms the analytic result that a 10x increase in the number of strategies roughly yields 10x increase in run time. The data indicate that utility-based strategy-selection algorithm performs very well in nominal usage cases ($\leq$ 100 strategies), and even handles 1000 strategies decently well. If necessary, algorithmic optimization can further improve performance.

In summary, the *netbwe* case study provided solid anecdotal evidence that system-administrative concerns can be expressed as Stitch policies in an intuitive, arguably more transparent manner. Interviews with sys-admins strengthened confidence that Stitch naturally captures the way in which sys-admins think about and resolve system problems, and that others could express adaptive concerns in Stitch. Finally, performance evaluation demonstrated favorably low runtime overhead in the utility-based strategy selection algorithm, the decision-making part of Stitch.

7. Related Work

In the Rainbow approach, a run-time software architecture model of the target system is maintained to manage the system. The adaptation language proposed in this article allows adaptation policies and decision criteria to be expressed in a form that the adaptation framework can analyze and automate. This section discusses two areas of related work to the Stitch language: adaptation frameworks and languages for specifying adaptation policies.

[5] 31 runs for $s = 10000$, but with only 5 runs at $q = 100$.

7.1. Adaptation frameworks

To date, many dynamic software architectures and architecture-based adaptation frameworks have been proposed and developed [8, 9, 10], often targeting specific architectural styles (e.g., Weaves for data-flow systems [20], ArchStudio for C2 hierarchical publish-subscribe systems [21], and [22] for managing robotics systems), and quality dimensions (e.g., Willow for survivability [23], Plastik for performance properties [24], and CASA for resource availability concerns in mobile network environments [25]). These related approaches on self-adaptive systems generally share a structure similar to the MAPE control cycle from IBM's Autonomic Computing initiative [6]. Viewed as an instance of the MAPE loop, Rainbow uses a shared knowledge base consisting of an explicit architecture model, a fixed repertoire of adaptation strategies, and utility preferences to monitor system states, detect problems, decide on a suitable strategy, and act on the system to effect adaptation. Furthermore, whereas most approaches assume certain structures in the target system and adapt for a single, or handful of, quality attributes, Rainbow is generic to architectural styles and handles multiple objectives.

Existing approaches also vary in the kind of systems that they can manage. For example, CoBRA manages service-oriented systems by weaving in changes via dynamic aspect-oriented programming (d-AOP) [26], limiting them to Java-based systems. IRIDIUM is based on a light-weight CORBA middleware for managing high-performance and real-time distributed systems [27]. StarMX is a framework exploiting standard Java MX management infrastructure to effect management and uses standard policy engines (e.g., JESS or QEngine) for specifying adaptation [28]. Weyns et al. use a multi-agent approach to provide flexible, context-driven dynamic organizations [29]. In terms of Rainbow, these complementary approaches typically provide versatile *translation layer* infrastructures and mechanisms for adaptation engineers to tailor monitoring and effecting to specific target systems, but are agnostic to using run-time architecture models.

7.2. Languages for specifying adaptation policies

Stitch comprises a number of key features for codifying how to adapt a system: a domain-specific language for self-adaptation, architectural abstractions as first-class entities, QoS-based choice, utility-based conflict resolution of policies, constraints, event-condition-actions, timing, and probabilities. Poladian and colleagues have argued a case for multi-dimensional utility analysis because converting all costs to a common currency was problematic [30]. We applied their ideas in our language to support analysis

of choice based on multi-dimensional adaptation attributes. Many adaptive approaches, especially Kephart and colleagues from IBM, leverage utility theory to select the best adaptation policy.

While various languages and tools have been developed with subsets of similar features as Stitch, none that we know of integrates these adaptation-enabling features into a single language. In most cases, a language may offer the operational constructs, but lack inherent support for one or more of Stitch's declarative constructs. CoBRA adaptation policies are manifest as Java aspects [26]; Edwards et. al. specify policies as Java classes in the Prism-MX platform [22]; IRIDIUM adaptation policies map fault-tree-like rules to an action, at a configurable evaluation rate [27]; Starfish provides a policy language that supports event-condition-actions as well as authorization (which Stitch does not have) [31]; Vogel and Giese's model-driven approach specifies adaptation policies in terms of model transformation rules [32]; although versatile and powerful, these policy mechanisms lack support for timing and uncertainty, and make it difficult to specify impact on quality-of-service and, thus, to resolve conflicts as well as to make QoS-based choice of the best adaptation. PBAAM [33] and StarMX [28] specify policies as expert system rules, which supports notions of uncertainty, but does not inherently support QoS evaluation, conflict resolution, and settling time. Petrucci et al. use a Python-based language with explicit notions of control and settling time [34], but lacks support for QoS-based evaluation and conflict resolution.

Ponder [35] is a full-featured policy languages that supports 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.

Some work has been done on formally modeling languages used to specify adaptations. For example, Darwin [36] and ArchWare [37] are modeled using π-calculus semantics to specify reconfigurations of distributed systems. While temporal-based and probabilistic model checkers (e.g., Symbolic Model Verifier [38], Spin [39], and PRISM [40]) can verify system properties at design time, Stitch harnesses temporal and probabilistic specification at runtime to control system properties for quality-of-service.

Finally, a number of researchers are looking into approaches to dynamically generate adaptations from high-level specifications of goals (cf. Kramer and Magee's three-layered approach [41]), using a variety of techniques such as planners [42, 43, 44] and workflow generators [45]. These approaches can

account for new components not conceived at design time, and so are quite flexible. However, it is difficult to analyze these plans to ensure that system qualities are maintained, and it is difficult to convey to sys-admins how these plans manage the system. In contrast, it is possible to statically analyze adaptation strategies in Stitch, and for sys-admins to specify adaptations using Stitch. On a spectrum of self-adaptive systems, from a completely static and manually managed system at one extreme, to a fully dynamic and autonomic system at the other, Stitch hits an important sweet spot: It offers an analyzable and declarative language that builds in more things statically and and takes advantage of the routine nature of adaptation. It may be possible to use a planning approach where statically specified strategies like those in Stitch could be used to prune the search space. Such an approach would require future research.

8. Discussion and Conclusion

In this article, we motivated and identified a set of language requirements to codify dynamic but routine system-administrative adaptation processes and presented the strategy definition language, Stitch. We presented Stitch's operational constructs (operator, tactic, strategy), declarative constructs (adaptation conditions, quality dimensions, utility preferences, tactic impact vectors, and strategy applicability conditions), its strategy selection algorithm, and its execution semantics in brief. We then illustrated Stitch using the example system, Znn.com. Finally, we provided evidence demonstrating the expressiveness and usefulness of Stitch. The Stitch language has been integrated into the Rainbow framework, which compiles and executes repairs, and uses the strategy meta-data to choose appropriate actions to take to adapt a system being managed by Rainbow.

In the remainder of this article, we discuss some limitations and open issues with the design of Stitch, and share some insights and future work.

Why three operational constructs?. By observing commonly performed system administration tasks, we have extracted a set of three constructs — operators, tactics, and strategies — and thus a basic ontology of adaptation language for automating mundane tasks in system management. Together with utilities, control of timing, and probabilities, these concepts form an adaptation language with the expressiveness to represent human adaptation expertise and the flexibility to incorporate dynamic preferences.

It might be argued that only two operational constructs suffices: one for the primitive steps, the other for the adaptation plan. In Stitch, the choice

Table 9: Invocation relationship between Stitch operational constructs

Invokes⇒	Operator	Tactic	Strategy
Operator	Unspecified	X	X
Tactic	√	X	X
Strategy	X	√	?

of three constructs was motivated by the needs of abstraction, packaging, and separation of concerns. *Operators* are provided by an architectural style and are mapped to effectors in the target system. A system represented by an architectural style may have many different ways to adapt, but will likely have a fixed set of operators that change the model correctly. Therefore, operators are a crucial reuse link between the adaptation mechanism and the architecture model, as well as the translation link between the adaptation mechanism and the target system.

However, *operators* do not suffice as an adaptation primitive for two reasons. First, the style writer who provides the architectural operators generally cannot know the adaptation context in which the operators will be used. Knowledge of the adaptation context and, particularly, the impact of operators on the utility dimensions for the target system are separate concerns of the adaptation engineer. Second, the adaptation engineer may need to define larger steps of adaptation than is provided by architectural operators. Hence, we need a second, distinct concept of *tactics*.

The third concept, *strategies*, embody explicit decision choices of the sys-admin and provide a packaging construct to constrain adaptation to individual domains of expertise for tractable reasoning.

Invocation relationship between constructs. The invocation relationship currently implemented in Stitch is summarized in Table 9. It should be obvious that no construct may invoke constructs above it, as that would defeat our expressed intent for abstraction, packaging, and separating concerns. Since operators are not implemented in Stitch (only used), Stitch does not designate whether operators can call other operators. If realized as programs, operators might conceivably invoke other, reusable modules of operators.

In contrast, nesting tactics makes cycles possible, complicating the single-step semantics of the tactic, as well as the evaluation of the condition and effect blocks. In particular, it is not obvious what role the condition block of a called tactic should play: Should it be allowed to abort the calling tactic? Should it be ignored altogether? Whether a strategy should be allowed to invoke another strategy is not immediately obvious. If such invocation were

allowed, the natural semantics would be to graft the tree of the invoked strategy at the node of the invoking strategy. However, we have chosen not to allow such invocation for simplicity, as it is unclear (a) how to reconcile the applicability conditions in the called strategy, (b) what it would mean for a strategy written for one quality to invoke a strategy intended for a different quality, and (c) how to handle recursive invocations. Furthermore, utility-based scoring of nested strategies would be greatly complicated.

Eliciting a large number of inputs. Stitch requires a number of different *inputs* from the stakeholders of a self-adaptive system, inputs which may be hard to elicit. Among these, utilities and probabilities are notoriously difficult for humans to capture precisely and correctly [46, 47]. Fortunately, our approach allows utility and probability specifications (and all aspects of Stitch specifications, for that matter) to be defined and refined incrementally. As Sousa has shown [47], a small set of simple utility function curves (linear, step, sigmoid, etc.) suffices to capture a significant variation of utility spaces while imposing low cognitive burden. For probability values, our preliminary analysis simulating adaptation scenarios using Markov Decision Processes suggests that adaptation decision outcomes are not overly sensitive to minor perturbations in probability values. While the complexity of utility and probability specifications will depend on domain and scope of adaptation, we could apply learning and simulation techniques to obtain these values.

Strategies versus plans. Given the current state of the target system as observed through the model (conditions) and a set of available tactics (actions), it is possible to use a planning algorithm to search for the best sequence (path) of tactics, and thus generate a strategy, as others have done (see Section 7). In our approach, we elicit strategies rather than use a planning algorithm for a number of reasons. Planning is ideal for exploring a large space of potential paths, but in the domains we are targeting, adaptation decisions are often known or, at least, constrained in scope. Because plans are generated on-the-fly, from a utility perspective, the uncertainty and potentially large number of generated plans make it difficult to perform an end-to-end, closed analysis of adaptation outcomes. In contrast, during adaptation, the set of strategies to explore is much smaller and, thus, a more efficient set on which to perform utility-based analysis of adaptation outcomes. A strategy yields consistent, verifiable, and reusable adaptation outcomes. Furthermore, we believe the notion of strategy is intuitive, giving sys-admins greater control over Stitch-type representations and its decision outcome.

Historical information. To avoid repeated invocation of strategies that have not historically resulted in system improvements, we incorporate simple history in subsequent strategy selections as a strategy failure profile that is factored into the utility calculation. Having access to history, we can integrate learning as part of the selection process to avoid oscillation and to improve selection quality. The next leap would be to integrate predictions into the monitoring infrastructure, possibly via predictor gauges [48] to anticipate certain quality-of-service problems, such as a rise in CPU load, drop in available bandwidth, or even change in the state of user tasks.

Dynamic update of strategies and strategy selection information. Currently, Rainbow does not support dynamic updates to account for changing user concerns or new strategies. Strategy meta-data, such as user preference profiles, could be easily changed at runtime. Furthermore, information about the strategies themselves, including branch probabilities, timing windows, impact vectors, and utility profiles could also be updated dynamically based on historical information or user preferences. We anticipate another layer above Rainbow that explicitly models user goals and preferences, and provides this information to Rainbow. The final form of this layer is a topic of future work, but a good starting point is our prior work on modeling user tasks and preferences in Aura [49, 47].

In terms of adding new strategies dynamically, because the Adaptation Manager of Rainbow chooses among a set of strategies each time an adapation cycle is performed, it would be feasible for this set to be changed, allowing incorporation and removal of strategies at run time. This forms another area of future work.

Opportunistic Improvement. The Adaptation Manager of Rainbow is reactive to problems in the system, invoking Stitch when constraints fail. While this favors rapid recognition of problems, it means that Rainbow is currently only reactive to problems in the system. Replacing the simple constraint evaluation mechanism with a more sophisticated QoS Analyzer would enable more proactive adaptation by looking for opportunities for system improvement. However, this would incur a greater computation overhead and potentially be more disruptive to the system. We do not envisage that this would result in changes to Stitch, but experimenting with more proactive approaches to adaptation is a rich area of future research.

Fault Localization. The first step of a tactic is typically identifying the elements of the system that caused the failure, to target the tactic to adapt those elements. This means that fault localization is done in Stitch. We

are investigating an alternative approach that factors fault localization out of the Stitch, and uses Spectrum-Based Multiple Fault Localization [50] to elicit sets of candidates for the most likely cause of a problem. These would then be passed to the Stitch strategies. Such an approach will allow us to also detect problems that are caused by multiple components [51].

Strategy Assurance. There are two aspects to assurance that we can investigate in future work. Does a strategy leave the system in a legal state? Does a strategy have the effect that it was designed to achieve? In the former, we can use Kim's work [52] to check the combination of operators to assure that a strategy produces legal systems according to the architectural style. In the latter, Stitch is amenable to utility-based analysis and perhaps model-checking with PRISM [40] or SPIN [39].

Stitch strikes a balance of allowing imperative specification of procedural algorithms, as well as declarative specification of decisions. It naturally captures human cognitive models when performing routine, but dynamic tasks, and provides simple constructs to control timing and handle uncertainties in the outcome of computation. We believe we may have struck a sweet spot of expressiveness and analysis capability for a class of systems with inherent uncertainties and timing concerns. While there may be many ways to specify the scripts themselves, an effective adaptation language must address certain critical issues, such as how long an adaptation takes, its effect on quality-of-service dimensions, and so on. We believe that these aspects of Stitch will generalize to other approaches to adaptation.

Acknowledgements

We would like to thank Ali Almossawi, who performed a summer independent study on Rainbow, during which he interviewed sys-admins and wrote a Stitch Editor Eclipse plug-in.

This research was sponsored by DARPA under grants N66001-99-2-8918 and F30602-00-2-0616, the US Army Research Office (ARO) under grants DAAD19-01-1-0485 and DAAD19-02-1-0389 ("Perpetually Available and Secure Information Systems") to Carnegie Mellon University's CyLab, the NASA High Dependability Computing Program under cooperative agreement NCC-2-1298, and a 2004 IBM Eclipse Innovation Grant. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of DARPA, the ARO, the U.S. government, NASA, IBM, or any other entity.

References

[1] Microsoft Corporation, System center operations manager 2007, `http://www.microsoft.com/systemcenter/opsmgr/` (2008).

[2] C. Frye, Self-healing systems, Application Development Trends (2003) 29–34.

[3] A. G. Ganak, T. A. Corbi, The dawning of the autonomic computing era, IBM Systems Journal 42 (1) (2003) 5–18.

[4] K. Scott, Computer, heal thyself, Information Week.

[5] A. B. Brown, D. A. Patterson, Undo for operators: Building an undoable e-mail store, in: In Proceedings of the 2003 USENIX Annual Technical Conference, 2003, pp. 1–14.

[6] J. O. Kephart, D. M. Chess, The vision of autonomic computing, Computer 36 (1) (2003) 41–50.

[7] T. M. Mitchell, Machine Learning, McGraw-Hill series in computer science, McGraw-Hill, New York, 1997.

[8] J. S. Bradbury, J. R. Cordy, J. Dingel, M. Wermelinger, A survey of self-management in dynamic software architecture specifications, in: WOSS '04: Proc. of the 1st ACM SIGSOFT Workshop on Self-managed Systems, ACM, New York, NY, 2004, pp. 28–33.

[9] D. Ghosh, R. Sharman, H. R. Rao, S. Upadhyaya, Self-healing systems - survey and synthesis, Decis. Support Syst. 42 (4) (2007) 2164–2185.

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

[11] M. Shaw, D. Garlan, Software Architecture: Perspectives on an Emerging Discipline, Prentice-Hall, 1996.

[12] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, P. Steenkiste, Rainbow: Architecture-based self adaptation with reusable infrastructure, IEEE Computer 37 (10).

[13] S.-W. Cheng, D. Garlan, B. Schmerl, J. a. P. Sousa, B. Spitznagel, P. Steenkiste, Using architectural style as a basis for self-repair, in: J. Bosch, M. Gentleman, C. Hofmeister, J. Kuusela (Eds.), Software Architecture: System Design, Development, and Maintenance, Kluwer Academic Publishers, Massachusetts, USA, 2002, pp. 45–59.

[14] D. Garlan, S.-W. Cheng, B. Schmerl, Increasing system dependability through architecture-based self-repair, in: R. de Lemos, C. Gacek, A. Romanovsky (Eds.), Architecting Dependable Systems, Lecture Notes in Computer Science, Springer-Verlag, Inc., New York, NY, USA, 2003, pp. 61–89.

[15] S.-W. Cheng, D. Garlan, B. Schmerl, Making self-adaptation an engineering reality, in: O. Babaoghu, M. Jelasity, A. Montroser, C. Fetzer, S. Leonardi, A. Van Moorsel (Eds.), Proceedings of the Conference on Self-Star Properties in Complex Information Systems, Vol. 3460 of LNCS, Springer-Verlag, 2005, also available from Springer-Verlag here.

[16] G. D. Abowd, R. Allen, D. Garlan, Formalizing style to understand descriptions of software architecture, ACM Trans. Softw. Eng. Methodol. 4 (4) (1995) 319–364.

[17] E. W. Dijkstra, Guarded commands, non-determinacy and formal derivation of programs, Communications of the ACM 18 (8) (1975) 453–457.

[18] S.-W. Cheng, Rainbow: Cost-effective software architecture-based self-adaptation, Technical Report CMU-ISR-08-113, Carnegie Mellon University School of Computer Science, 5000 Forbes Avenue, Pittsburgh, PA 15213 (May 17, 2008).

[19] Wikipedia, Utility — wikipedia, the free encyclopedia, `http://en.wikipedia.org/w/index.php?title=Utility&oldid=200699805`, [Online; accessed 25-March-2008] See "Expected utility" section. (Mar. 17, 2008).

[20] M. M. Gorlick, R. R. Razouk, Using Weaves for software construction and analysis, in: Proc. of the 13th International Conf. of Software Engineering, IEEE Computer Society Press, Los Alamitos, CA, USA, 1991, pp. 23–34.

[21] E. M. Dashofy, A. Hoek, R. N. Taylor, Towards architecture-based self-healing systems, ACM Press, New York, NY, USA, 2002, pp. 21–26.

[22] G. Edwards, J. Garcia, H. Tajalli, D. Popescu, N. Medvidovic, G. Sukhatme, B. Petrus, Architecture-driven self-adaptation and self-management in robotics systems, [53], pp. 142–151.

[23] A. L. Wolf, D. Heimbigner, A. Carzaniga, K. M. Anderson, N. Ryan, Achieving survivability of complex and dynamic systems with the Willow framework, in: Proceedings of the Working Conference on Complex and Dynamic Systems Architecture, 2001.

[24] T. V. Batista, A. Joolia, G. Coulson, Managing dynamic reconfiguration in component-based systems., in: EWSA, Vol. 3527 of LNCS, Springer, 2005, pp. 1–17.

[25] A. Mukhija, M. Glinz, A framework for dynamically adaptive applications in a self-organized mobile network environment, in: ICDCSW '04: Proceedings of the 24th International Conference on Distributed Computing Systems Workshops - W7: EC (ICDCSW'04), IEEE Computer Society, Washington, DC, 2004, pp. 368–374.

[26] F. Irmert, T. Fischer, K. Meyer-Wegener, Runtime adaptation in a service-oriented component model, in: Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems [54], pp. 97–104, 529080.

[27] S. S. Andrade, R. J. de Araujo Macedo, A non-intrusive component-based approach for deploying unanticipated self-management behaviour, [53], pp. 152–161.

[28] R. Asadollahi, M. Salehie, L. Tahvildari, Starmx: A framework for developing self-managing java-based systems, [53], pp. 58–67.

[29] D. Weyns, R. Haesevoets, B. Van Eylen, A. Helleboogh, T. Holvoet, W. Joosen, Endogenous versus exogenous self-management, in: Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems [54], pp. 41–48, 529080.

[30] V. Poladian, S. Butler, M. Shaw, D. Garlan, Time is not money: The case for multi-dimensional accounting in value-based software engineering, in: Fifth Workshop on Economics-Driven Software Engineering Research (EDSER-5), 2003.

[31] T. Bourdenas, M. Sloman, Starfish: policy driven self-management in wireless sensor networks, in: Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems [55], pp. 75–83.

[32] T. Vogel, H. Giese, Adaptation and abstract runtime models, in: Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems [55], pp. 39–48.

[33] J. C. Georgas, R. N. Taylor, Policy-based self-adaptive architectures: a feasibility study in the robotics domain, in: Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems [54], pp. 105–112, 529080.

[34] V. Petrucci, O. Loques, D. Mossé, A framework for dynamic adaptation of power-aware server clusters, in: Proceedings of the 2009 ACM symposium on Applied Computing, SAC '09, ACM, New York, NY, USA, 2009, pp. 1034–1039.

[35] N. Damianou, N. Dulay, E. Lupu, M. Sloman, The ponder policy specification language, in: POLICY '01: Proc. of the International Workshop on Policies for Distributed Systems and Networks, Springer-Verlag, London, UK, 2001, pp. 18–38.

[36] J. Magee, J. Kramer, Dynamic structure in software architectures, in: SIGSOFT '96: Proc. of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering, ACM, New York, NY, USA, 1996, pp. 3–14.

[37] R. Morrison, D. Balasubramaniam, F. Oquendo, B. Warboys, R. M. Greenwood, An active architecture approach to dynamic systems co-evolution, in: ECSA, Vol. 4758 of LNCS, Springer, 2007, pp. 2–10.

[38] J. R. Burch, E. M. Clarke, D. E. Long, Symbolic model checking with partitioned transition relations, in: VLSI 91, 1990.

[39] G. J. Holzmann, The spin model-checker, in: Proc. FORTE 1999, Vol. 28, 1997, pp. 481–497.

[40] M. Kwiatkowska, G. Norman, D. Parker, Prism: Probabilistic symbolic model checker, in: T. Field, P. Harrison, J. Bradley, U. Harder (Eds.), Proc. TOOLS 2002, Vol. 2324 of Lecture Notes in Computer Science, Springer, 2002, pp. 200–204.

[41] J. Kramer, J. Magee, Self-managed systems: an architectural challenge, in: 2007 Future of Software Engineering, FOSE '07, IEEE Computer Society, Washington, DC, USA, 2007, pp. 259–268.

[42] D. Kim, S. Park, Reinforcement learning-based dynamic adaptation planning method for architecture-based self-managed software, [53], pp. 76–85.

[43] D. Sykes, W. Heaven, J. Magee, J. Kramer, Exploiting non-functional preferences in architectural adaptation for self-managed systems, in: Proceedings of the 2010 ACM Symposium on Applied Computing, SAC '10, ACM, New York, NY, USA, 2010, pp. 431–438.

[44] C. Ghezzi, M. Pradella, G. Salvaneschi, Programming language support to context-aware adaptation: a case-study with erlang, in: Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems [55], pp. 59–68.

[45] C. E. da Silva, R. de Lemos, Using dynamic workflows for coordinating self-adaptation of software systems, [53], pp. 86–95.

[46] S. A. Butler, Security attribute evaluation method: a cost-benefit approach, in: Proc. of the 24th International Conf. on Software engineering, ACM Press, 2002, pp. 232–240.

[47] J. P. Sousa, Scaling task management in space and time: Reducing user overhead in ubiquitous-computing environments, Technical Report CMU-CS-05-123, Carnegie Mellon University School of Computer Science, 5000 Forbes Avenue, Pittsburgh, PA 15213 (Mar. 28, 2005).

[48] V. Poladian, D. Garlan, M. Shaw, B. Schmerl, J. P. Sousa, M. Satyanarayanan, Leveraging resource prediction for anticipatory dynamic configuration, in: Proc. of the 1st IEEE International Conf. on Self-Adaptive and Self-Organizing Systems (SASO '07), 2007, pp. 214–223.

[49] J. P. Sousa, D. Garlan, Aura: an architectural framework for user mobility in ubiquitous computing environments, in: J. Bosch, M. Gentleman, C. Hofmeister, J. Kuusela (Eds.), Software Architecture: System Design, Development, and Maintenance (Proc. of the 3rd Working IEEE/IFIP Conf. on Software Architecture), Kluwer Academic Publishers, 2002, pp. 29–43.

[50] R. Abreu, A. J. C. van Gemund, Diagnosing multiple intermittent failures using maximum likelihood estimation, Artif. Intell. 174 (18) (2010) 1481–1497.

[51] P. Casanova, B. Schmerl, D. Garlan, R. Abreu, Architecture-based run-time fault diagnosis, submitted for Publication (2011).

[52] J. S. Kim, D. Garlan, Analyzing architectural styles, Journal of Software and Systems 83 (7) (2010) 1216–1235.

[53] 2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, IEEE Computer Society, Los Alamitos, CA, USA, 2009.

[54] SEAMS '08: Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems, ACM, New York, NY, USA, 2008, 529080.

[55] SEAMS '10: Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, ACM, New York, NY, USA, 2010.

Appendix A. Stitch Grammar

The Stitch grammar is shown below. Details of statements and expressions have been elided for space considerations.

```
1  # ANTLR EBNF grammar: lowercase for non-terminal; uppercase, terminal
2
3          module ::= MODULE IDEN SEMI  # SEMI := semicolon
4                     (import)* (function)* (tactic)* (strategy)* EOF ;
5
6          import ::= IMPORT (LIB|MODEL|OP) STR_LIT impRenameList? SEMI ;
7   impRenameList ::= LBRACE impRenamePair (COMMA impRenamePair)* RBRACE;
8   impRenamePair ::= IDEN AS IDEN ;     # IDEN := identifier
9
10       function ::= DEFINE type IDEN ASSIGN expression SEMI ;
11       operator ::= IDEN (LPAREN argExprList RPAREN)? ;
12
13         tactic ::= TACTIC signature LBRACE
14                      ( declaration SEMI )*
15                      CONDITION LBRACE (booleanExpr SEMI)* RBRACE
16                      ACTION LBRACE (statement)* RBRACE
17                      EFFECT LBRACE (booleanExpr SEMI)* RBRACE
18                    RBRACE ;
19
20       strategy ::= STRATEGY IDEN
21                    LBRACKET booleanExpr RBRACKET
22                    LBRACE (function)* (strategyExpr)* RBRACE ;
23    strategyExpr ::= IDEN COLON strategyCond IMPLIES strategyAction ;
24    strategyCond ::= LPAREN (HASH LBRACKET condProbVal RBRACKET)?
25                           (booleanExpr | SUCCESS | DEFAULT) RPAREN ;
26  strategyAction ::= strategyClosed SEMI
27                   | strategyOpen (AT LBRACKET expression RBRACKET)?
28                     LBRACE (strategyExpr)+ RBRACE ;
29  strategyClosed ::= DONE | FAIL
30                   | DO LBRACKET (IDEN | INT_LIT)? RBRACKET IDEN ;
31    strategyOpen ::= IDEN LPAREN argExprList RPAREN
32                   | NULLTACTIC ;
33     condProbVal ::= FLOAT_LIT | IDEN (LBRACE IDEN RBRACE)?
34
35       signature ::= IDEN LPAREN (type IDEN (COMMA type IDEN)*)? RPAREN;
36     declaration ::= type IDEN (ASSIGN expression)?
37                          (COMMA IDEN (ASSIGN expression)?)* ;
```