FUSION: A Framework for Engineering Self-Tuning Self-Adaptive Software Systems

Ahmed Elkhodary
Department of Computer Science
George Mason University
aelkhoda@gmu.edu

Naeem Esfahani
Department of Computer Science
George Mason University
nesfaha2@gmu.edu

Sam Malek
Department of Computer Science
George Mason University
smalek@gmu.edu

ABSTRACT
Self-adaptive software systems are capable of adjusting their behavior at run-time to achieve certain objectives. Such systems typically employ analytical models specified at design-time to assess their characteristics at run-time and make the appropriate adaptation decisions. However, prior to system's deployment, engineers often cannot foresee the changes in the environment, requirements, and system's operational profile. Therefore, any analytical model used in this setting relies on underlying assumptions that if not held at run-time make the analysis and hence the adaptation decisions inaccurate. We present and evaluate FeatUre-oriented Self-adaptatION (FUSION) framework, which aims to solve this problem by learning the impact of adaptation decisions on the system's goals. The framework (1) allows for automatic online fine-tuning of the adaptation logic to unanticipated conditions, (2) reduces the upfront effort required for building such systems, and (3) makes the run-time analysis of such systems very efficient.

Categories and Subject Descriptors
D.2.10 [**Software Engineering**]: Design – *Methodologies.*

General Terms
Algorithms, Performance, Design.

1. INTRODUCTION
The ever-growing complexity of software systems coupled with the need to maintain its quality of service (QoS) characteristics, even under adverse conditions and highly uncertain environments, have instigated the emergence of *self-adaptive* software systems [13]. A self-adaptive software system is capable of modifying itself at run-time to achieve certain functional or QoS goals. The development of such systems has shown to be significantly more challenging than static and predictable software systems [2].

In particular, engineering the adaptation logic poses the greatest difficult. Since software engineers often cannot foresee all of the changes in the environment, requirements, and system's operational profile at design-time, they rely on analytical models that given the monitoring data obtained at run-time assess the system's ability to satisfy its goals. The results produced by the analytical models thus serve as indicators for making the adaptation decisions.

FSE'10, November 7–11, 2010, Santa Fe, New Mexico, USA.

Generally, this approach suffers from three shortcomings:

- **Unwieldy for use.** Existing state of the art self-adaptive frameworks require the engineer to construct and utilize complex analytical models. Unfortunately, the majority of widely used analytical models (e.g., Queueing Network models [8] for performance analysis) have to painstakingly be customized to the unique characteristics of an application domain. Moreover, for any application-specific goal, an appropriate analytical model would have to be developed from scratch; a task that is often very difficult, when one considers the complexity of today's software systems. Further exacerbating the problem is that software engineering practitioners are typically not savvy mathematicians and find it difficult to build systems that make use of such models.
- **Wrong assumptions.** Analytical models make simplifying assumptions or presume certain properties of the running system that may not bear out in practice. These models are specified at design-time and cannot cope with the run-time changes that were not accounted for in their formulation. These assumptions could make the analysis and hence the adaptation decisions inaccurate.
- **Efficiency.** Efficiency of analysis and planning is of utmost importance in most self-adaptive software systems that need to react quickly to situations that arise at run-time. At the same time, searching for an optimal architectural configuration (i.e., solution) is often computationally very expensive [2].

In this paper, we present an alternative and relatively unexplored method of constructing self-adaptive software systems aimed at alleviating the three problems mentioned above. Instead of manually developing an analytical model that relates the impact of adaptation decisions on the system's goals, we present a learning-based approach in which such a model is automatically induced from the monitored data. The approach not only allows for automatic online fine-tuning of the adaptation logic to unanticipated conditions, but also reduces the upfront effort required for building such systems.

We describe this research in the context of a framework, entitled *FeatUre-oriented Self-adaptatION (FUSION)*, which by using a feature-oriented system model learns the impact of feature selection and feature interactions on the system's competing (conflicting) goals. It then uses this knowledge to efficiently adapt the system to satisfy as many user-defined goals as possible.

In this paper, we elaborate on three key contributions of FUSION:

- FUSION adapts and learns in terms of *features*. A feature is a domain and platform independent method of representing a particular system capability [7,12]. This along with the fact that FUSION does not prescribe a particular analytical model

makes the approach applicable to any software system with minimal effort.

- FUSION copes with the changing dynamics of the system, even those that were not anticipated, through continuous observation and induction. In turn, FUSION is capable of learning run-time behaviors unforeseen at design-time.
- FUSION incorporates the engineer's knowledge of the system and its capabilities in the form of feature relationships. It then uses these relationships to reduce the valid configuration space significantly, which makes not only the learning feasible but also the adaptation planning efficient for use at run-time.

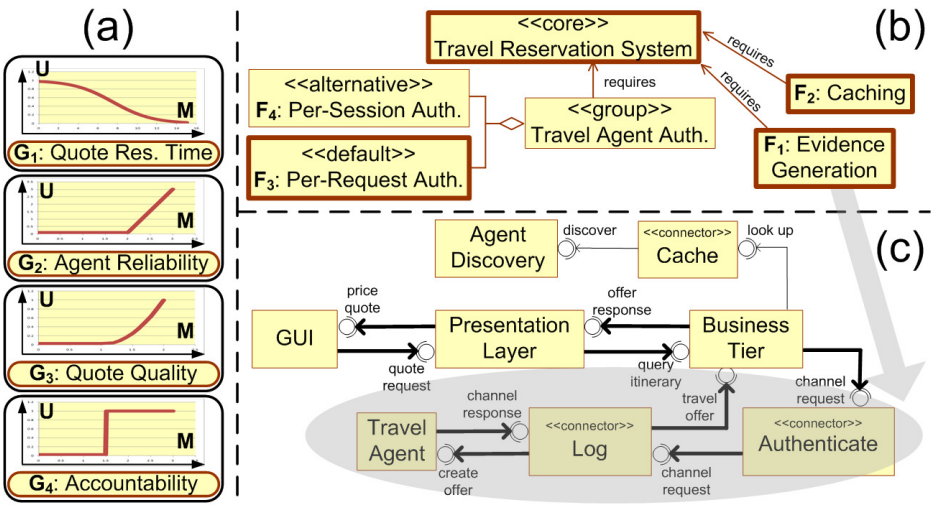

Figure 1. Travel Reservation System: (a) goals are quantified in terms of utility obtained for a given level of metric; (b) subset of available features, where features with thick borders are selected; (c) software architecture corresponding to the selected features, where the thick lines represent an execution scenario associated with goal G1.

The rest of this paper is organized as follows. Section 2 motivates the problem using a system that also serves as a running example in this paper. Section 3 provides an overview of FUSION. Sections 4, 5, and 6 respectively detail FUSION's feature-oriented model of adaptation, learning method, and adaptation planning. Sections 7 and 8 present the implementation and evaluation details of FUSION. The paper concludes with an overview of the related work and future avenues of research.

2. MOTIVATION

We illustrate and evaluate the concepts using an online Travel Reservation System (TRS), which is representative of web applications used by large organizations for making travel reservations. Figure 1c shows a subset of its software architecture using the traditional component-and-connector view. TRS aims to provide the best airline ticket prices in the market. To make a price quote for the user, TRS takes the trip information from the user, and consequently discovers and queries various travel agent services. The travel agents reply with their itinerary offers, which are then sorted and presented in ascending order of quoted price.

In addition to the functional goals, the system is required to attain a number of QoS goals, such as performance, security, and accountability. To that end, solutions for each QoS concern were developed, e.g., caching for performance, authentication for security, and logging of activities for accountability purposes.

In addition, TRS needs to be self-adaptive to deal with unexpected situations, such as traffic spikes or security attacks. For instance, enable caching to improve performance during a traffic spike, increase authentication to thwart a security attack, and enable logging to ensure non-repudiation of transactions (i.e., accountability). The adaptation logic of TRS also needs to balance tradeoffs (conflicts) when it selects from the available adaptation choices, e.g., improving security may degrade response time.

As mentioned earlier, there are three problems associated with the construction of adaptation logic. Consider the issues that may arise in the context of TRS:

- **Unwieldy for use.** Consider the difficulty of accurately estimating the impact of enabling a particular type of authentication on the price quotes in TRS. Using a heavy authentication protocol increases the system's response time, which forces more timeouts on the client-side. This reduces the total number of received offers, and hence the quality of price quotes. Quantitatively modeling this trade-off is difficult, as it depends on many dynamic parameters: available service providers, network characteristics, and so on.
- **Wrong assumptions.** Consider an analytical model that quantifies the impact of an adaptation decision on the response time of receiving price quotes from travel agents (thick lines in Figure 1c). Such a model would inevitably make simplifying assumptions based on what the engineers believe to be the main sources of delay in the system. For instance, if fast communication links are assumed, the analytical model may ignore the network delay. Since accurately predicting the characteristics of a dynamic system is extremely difficult, the assumptions may not hold, making the analysis and hence the adaptation decisions inaccurate.
- **Efficiency.** To satisfy multiple goals, self-adaptation logic needs to search in a configuration space that is equivalent to the combined complexity of all the analytical models involved. As an example, consider how TRS would make use of P authentication components for authenticating the network traffic between its N software components, which may be deployed on M different hardware platforms. Analyzing the impact of authentication alone on the system's goals would require exploring a space of $(M^N \text{ possible deployments})^{P \text{ possible ways of authentication}} = M^{NP}$ possible configurations. Such a problem is computationally expensive to solve at run-time for any sizable system. This is while authentication may be only one concern out of many.

The aforementioned difficulties have shaped our motivation in the development the FUSION framework, as described next.

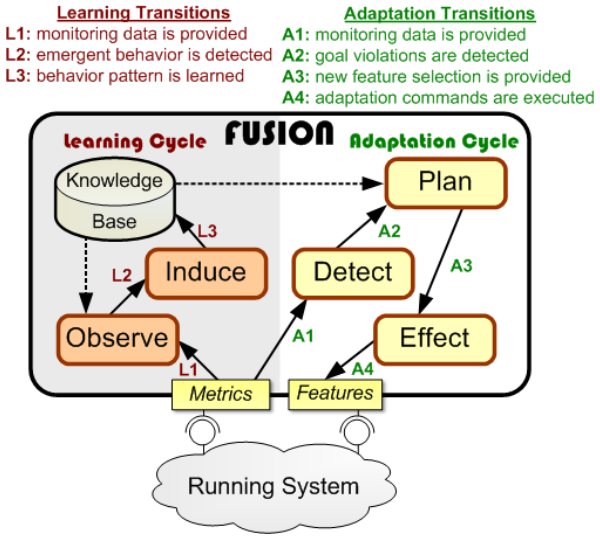

Figure 2. Overview of the FUSION framework.

3. FUSION OVERVIEW

Figure 2 depicts the FUSION framework as it adapts a running software system. The running system is variable in the sense that its features can be selected (i.e., enabled/disabled) on demand. FUSION modifies the feature selection to resolve QoS tradeoffs and satisfy as many goals as possible. For example, if the TRS system violates *Quote Response Time* goal, it is adapted to a new feature selection that brings down the response time and keeps other goals satisfied.

As depicted in Figure 2, FUSION makes such adaptation decisions using a continuous loop, called *adaptation cycle*. The adaptation cycle collects metrics (measurements) and optimizes the system by executing three activities in the following sequence:

- Based on the metrics collected from the running system, *Detect* calculates the achieved utility (i.e., measure of user's satisfaction) to determine if a goal violation has occurred.
- When a goal is violated, *Plan* searches for an optimal configuration (feature selection) that maximizes overall utility.
- Given a new feature selection, *Effect* determines a set of adaptation steps (i.e., enable/disable features) to ensure consistency during adaptation.

FUSION uses *learning cycle* (depicted in Figure 2) to learn the impact of adaptation decisions in terms of feature selection on the system's goals. The first execution of learning cycle occurs before the system's initial deployment. The system is either simulated or executed in offline mode and metrics corresponding to each feature selection is collected. This data is used to train FUSION to induce a preliminary model of the system's behavior.

At run-time, the learning cycle continuously executes, and as the dynamics of the system and its environment change, the framework tunes itself. For example, when FUSION adapts TRS to resolve a "*quote response time"* violation, it keeps track of the gap between the expected and the actual outcome of the adaptation. This gap is an indicator of the new behavioral patterns in the system. Learning cycle collects such indicators and tunes itself by executing two activities in the following sequence:

- Based on the measurements collected from the system, *Observe* detects any emerging patterns of behavior. An emergent pattern is detected when the system sets the wrong expectation (i.e., inaccurate impact of adaptation on utility).
- *Induce* learns the new behavior through induction and stores the refined model in the *knowledge base,* which is used to make (more) informed adaptation decisions in future cycles.

In the following three sections, we describe FUSION's underlying model, learning cycle, and adaptation cycle in more detail.

4. FUSION MODEL

We describe FUSION's approach to modeling adaptation choices and goals. As detailed in Sections 5 and 6, FUSION's model is the key enabler of effective learning and efficient analysis.

4.1 Feature-Based Adaptation

In FUSION, the unit of adaptation is a *feature*. A feature is an abstraction of a capability provided by the system. A feature may affect either the system's functional (e.g., ability to print receipts) or non-functional (e.g., ability to authenticate) properties.

The use of features as an abstraction makes the FUSION framework independent of a particular implementation platform or application domain. For example, in a rule-based system a feature may correspond to a set of rules, in a service-oriented system it may correspond to a set of services in a workflow, and so on. For clarity, in this paper we assume a particular realization of a feature: a feature represents an extension of the architecture at well-defined *variation points*. A feature maps to a subset of the system's software architecture. For example, Figure 1b shows the mapping of *Evidence Generation* feature to a subset of the TRS.

Figure 1b shows a simple feature model for TRS. There are four features in the system and one common *core*. The features in the example use two kinds of relationships: *dependency*, and *mutual exclusion*. The dependency relationship indicates that a feature requires the presence of another feature. For example, enabling the *Evidence Generation* feature requires having the *core* feature enabled as well. Mutual exclusion is another relationship, which implies that if one of the features in a mutual *group* is enabled, the others must be disabled. For example, *Per-Request Authentication* and *Per-Session Authentication* cannot be enabled at the same time. Feature modeling supports several other types of inter-feature relationships (see [7]) that for brevity are not detailed here.

The feature model is used to identify the current system configuration in terms of a feature selection string. In a feature selection string, enabled features are set to "1"; disabled features are set to "0". For example, one possible configuration of TRS would be "1101", which means that all features from Figure 1b are enabled except *Per-Request Authentication*. The adaptation of a system is modeled as a transition from one feature selection string to another, which we detail in Section 6.3.

4.2 Goals

A *goal* represents the user's functional or QoS objectives for a particular execution scenario. A goal consists of a *metric* and a *utility*. A metric is a measurable quantity (e.g., response time) that can be obtained from a running system. A utility function is used to express the user's preferences (satisfaction) for achieving a particular metric. For instance, goal G_1 in Figure 1a specifies the user's degree of satisfaction (U) with achieving a specific value of *Quote Response Time* (M).

Elicitation of user's preferences, while an important prerequisite for using the framework, is a topic that has been investigated extensively in the existing literature (e.g., [17]), and considered to be outside the focus of this paper. FUSION is independent of the type of utility functions and the approach employed in extrapolating them. Arguably any user can specify hard constraints, which can be trivially modeled as step-functions (e.g., G_4 depicted in Figure 1a). Alternatively a utility function may take on more advanced forms (e.g., sigmoid curve), and express more complex preferences, such as G_1, G_2, and G_3.

FUSION places one constraint on the range of utility functions: they need to return zero for the metric values that are not acceptable to the user. As will be discussed in Section 6.1, when a utility associated with a goal reaches zero, FUSION considers that goal violated and initiates adaptation.

5. FUSION LEARNING CYCLE

FUSION copes with the changing dynamics of the system through learning. Learning discovers relationships between features and metrics. Each relationship is represented as a function that quantifies the impact of features, including their interactions, on a metric. In TRS, for example, the result of learning would be four functions, one function for each of the four metrics M_{G1} through M_{G4}. Each function takes a feature selection as input and produces an estimated gain/loss value for the metric as output.

Learning is typically a very computationally intensive process. In particular, learning simply at the architectural-level is infeasible for any sizable system, which is the reason why its application in existing architecture-based adaptation approaches has been limited. FUSION's feature-oriented model offers two opportunities for tackling the complexity of learning:

1. Learning operates on *feature selection space*, which is significantly smaller than the traditional architectural-level configuration space. The features in FUSION encode the engineer's domain knowledge of the practical variation points in a given application. For instance, the engineer may only consider a small reasonable subset of M^{NP} authentication driven architectural choices (recall Section 2). Figure 1b shows two authentication strategies modeled as features in TRS: F_3 and F_4. These two features represent what the TRS security engineer envisioned to be the reasonable applications of authentication in the system.
2. By using the inter-feature relationships (e.g., mutual exclusions, dependencies), one can significantly reduce the feature selection space. For instance, Figure 1b shows a mutual exclusion relationship between F_3 and F_4. This relationship is manifestation of the domain knowledge that applying two authentication protocols to the same execution scenario is not appropriate. Such relationships reduce the space of valid feature selections significantly, further aiding FUSION to learn their trade-offs with respect to goals.

```
SelectionCounter(Feature F) :int
int Count = 1;
switch (F.Type)
    case "MutualGroup":
        for each ( C in F.Children )
                Count += SelectionCounter(C) - 1;
    case "LeafFeature":
        Count = 2;
    default:
        for each ( C in F.Children )
            Count *= SelectionCounter(C);
            Count ++;
return Count;
```

Figure 3. Algorithm for sizing the feature selection space.

Figure 3 is an algorithm that determines the size of the valid feature selection space in a feature model recursively. Applying this algorithm to the feature model in Figure 1b yields a space of 8 valid feature selections, calculated as follows: $2 \text{ from } F_1 \times 2 \text{ from } F_2 \times (2 \text{ from } F_3 + 2 \text{ from } F_4 - 2)$. Without considering the inter-feature relationships to prune the invalid selections, the space of feature selection would have been $2^{number\ of\ features} = 2^4 = 16$.

Learning starts with a training process that populates FUSION's knowledge base with an initial set of functions. Consequently, at run-time, the learning cycle fine-tunes the functions to accommodate emergent behaviors. The rest of this section describes the two activities that take place to populate and fine-tune the knowledge base.

5.1 Observe

Observe is a continuous execution of two activities: (1) normalize raw metric values to make them suitable for learning, and (2) test the accuracy of learned functions. We describe each of these activities below.

Learning in terms of raw data hampers the accuracy. For instance, consider the fact that the actual impact of a feature on a metric may depend on the system's workload. Therefore, the actual metric data obtained from executing the same software system (i.e., same feature selection) under different workloads may result in starkly different metric readings, thus making it difficult to generalize in the form of a learned function.

To address this issue, *Observe* takes raw metric data through an automated normalization process prior to storing them as observation records. Many normalization techniques can be applied to transform the learning inputs into a representation that is less sensitive to the execution context. In Table 1, observation records were normalized using *studentized residual* [1] as follows: $(raw\ value - \overline{X})/s$, where $\overline{X}$ and s are the mean and the standard deviation of the collected data, respectively. Normalization using studentized residual does not require knowledge of population parameter, such as absolute min-max values and population mean. It only requires knowledge of mean and standard deviation for sample data.

Once a preliminary set of functions are learned (details provided in the next section), *Observe* continuously tests the accuracy of functions against the latest collected observations. Accuracy is defined as the difference between predicted value of a reward using the learned functions and actually observed value. For that purpose, we use the *learning accuracy threshold* provided by the learning algorithm itself. Note that the majority of learning algorithms provide an error threshold that indicates the noise in learned functions. On top of this, one may specify an additional margin of inaccuracy that can be tolerated, in cases where it is not desirable to run the learning algorithm frequently. If the accuracy test fails, *Observe* takes this as an indicator that either learning is incomplete or new patterns of behavior are emerging in the system and, thus, notifies the *Induce* activity to fine-tune the learned functions using the latest set of observations.

5.2 Induce

Based on the collected observations, the *Induce* activity constructs several functions that estimate the impact of making a feature

Table 1. Normalized observation records

Indep. Vars				Dependent Variables				
F_1	F_2	F_3	F_4	M_{G1}	M_{G2}	M_{G3}	M_{G4}	..
..	..	..	..	..	..	..	..	..
0	0	0	1	- 0.842	- 0.308	1.432	- 0.521	..
1	0	0	1	0.650	0.513	1.371	1.501	..
0	1	0	1	- 1.470	- 0.719	1.378	- 0.522	..
0	0	1	0	- 0.132	-0.103	0.740	- 0.712	..
0	0	0	1	- 0.736	- 1.335	1.103	- 0.117	..
1	0	1	0	1.574	1.951	0.550	1.566	..
1	1	0	1	0.153	0.513	1.090	1.501	..
1	1	1	0	0.804	-0.513	0.562	1.566	..
..	..	..	..	..	..	..	..	..

selection on the corresponding metrics. *Induce* executes two steps. The first step is a *significance test* that determines the features with the most significant impact on each metric. This allows us to reduce the number of independent variables (recall Table 1) that learning needs to consider for each metric. After the significance test, we apply the learning, which for each goal, given the normalized observations and the features with significance, derives the corresponding relationships.

While FUSION is not tied to a particular learning algorithm, in our implementation we have used the M5 model tree (MT) algorithm [9], which is a machine learning technique with three important properties: (1) ability to eliminate insignificant features automatically, (2) fast training and convergence, and (3) efficient interaction detection. Table 2 shows the induced relationships among features and metrics for TRS.

The information in this table can also be represented simply as a set of functions. For instance, a function estimating M_{G1} corresponds to the second column of the table as follows:

$$M_{G1}=\mathbf{1.553}\ F_1 - \mathbf{0.673}\ F_2 + \mathbf{0.709}\ F_3 + \mathbf{0.163}\ F_1F_3 - \mathbf{0.843} \quad (1)$$

Each feature is assigned a coefficient that is effective only when the feature is enabled (i.e., it is set to "1"). For example, the expected value of M_{G1} for a feature selection where only F_1 and F_3 are enabled ("1010") can be calculated as follows:

$$M_{G1}=\mathbf{1.553}\times 1 + 0 + \mathbf{0.709}\times 1 + \mathbf{0.163}\times 1\times 1 - \mathbf{0.843} = \mathbf{1.482} \quad (2)$$

When making adaptation decisions, values obtained from the induced functions (e.g., 1.482 from Eq. 2 above) are denormalized by using the inverse of normalization equation presented in the previous section. The denormalized value for a metric is then plugged into the corresponding utility function to determine the impact of feature selection on the goal.

Note that the induction also captures the impact of feature interactions on metrics. For example, Eq. 1 specifies that enabling both F_1 (*Evidence Generation*) and F_3 (*Per-Request Authentication*) increases M_{G1}. This is because according to Table 2, F_1F_3 increases the response time by 0.163, which decreases the utility of G_1 (utility of G_1 is shown in Figure 1a). Using Figure 1c we can explain this feature interaction as follows. F_1 introduces a delay by adding a mediator connector, called *Log*, that records the transactions with remote travel agents. At the same time, F_3 changes the behavior of the *Log,* as it causes an additional delay in mediating the exchange of per session authentication credentials. Enabling the two features at the same time has a negative ramification that is beyond the individual impact of each.

In some cases, learning may need to incorporate some contextual factors as independent variables, due to their impact on metrics. Consider a system with drastically different workloads at different times of day that cannot be dealt with effectively through

Table 2. Learned metric functions. An empty cell means that the corresponding feature has no significant impact.

Significant Variables	Induced Functions				
	M_{G1}	M_{G2}	M_{G3}	M_{G4}	..
Core	- 0.843	- 0.161	1.332	- 0.488	..
F_1	1.553	1.137		1.548	..
F_2	- 0.673	- 0.938			..
F_3	0.709		- 0.672		..
F_4		- 0.174			..
F_1F_3	0.163				..
..	..	..	..	..	..

normalization. In that case, the result of learning would be a set of equations that estimate the impact of feature selection in different contexts. For example, the following equations estimate the impact of feature selection on M_{G1} under different workloads (w):

$$M_{G1}=\begin{cases} \ldots + \mathbf{5.54}F1 - \mathbf{2.14}F2 + \mathbf{2.4}F3 \ldots, & w \le 1.21 \\ \ldots + \mathbf{1.98}F1 - \mathbf{1.46}F2 + \mathbf{1.4}F3 \ldots, & 1.21 < w \le 1.29 \\ \ldots + \mathbf{0.95}F1 + \mathbf{0.66}F3 + \mathbf{0.24}F1F3 \ldots, & w > 1.29 \end{cases} \quad (3)$$

Where w is the average inter-arrival time between requests in milliseconds; lower inter-arrival time implies higher workload. Here, the generated functions indicate that TRS reaches saturation when w is in the range of 1.21–1.29 milliseconds. Since the impact of features on M_{G1} changes dramatically in that range, the learning algorithm produces a separate equation targeted at that. Although these equations may be of any type (e.g., linear, sigmoid, or exponential as in [4]), for clarity we have limited the discussion to multi-linear equations only.

6. FUSION ADAPTATION CYCLE

In this section, we describe how *Detect*, *Plan* and *Effect* use the learned knowledge to adapt a software system in FUSION. The underlying principle guiding the adaptation strategy in FUSION is simple: *if the system works (i.e., satisfies the user), do not change it; when it breaks, find the best fix for only the broken part*. While intuitive, this approach sets FUSION apart from many of the existing works that either attempt to continuously optimize the entire system, or solely solve the constraints (i.e., violated goals) in the system. FUSION adopts a middle ground, which we believe to be the most sensible, and achieves the following objectives:

1. *Reduce Interruption*: Adaptation typically interrupts the system's operation (e.g., transient unavailability of certain functionality). In turn, even if at run-time a solution with a higher utility is found, one may opt not to adapt the system to avoid such interruptions. FUSION reduces interruptions by adapting the system only when a goal is violated.
2. *Efficient Analysis*: Often in run-time adaptation, the performance of analysis is crucial. FUSION uses the learned knowledge to scope the analysis to only the parts of the system that are affected by the adaptation, hence making it significantly more efficient than assessing the entire system.
3. *Stable Fix*: Given the overhead and interruption associated with the adaptation, effecting solutions that provide a temporary fix are not desirable. We would like FUSION to minimize frequent adaptation of the system for the same problem. To that end, instead of simply satisfying the violated goals, FUSION finds a near optimal solution that is less likely to be broken due to fluctuations in the system.

6.1 Detect

The adaptation cycle is initiated as soon as *Detect* determines a goal violation. This is achieved by monitoring the utility functions

(recall Section 4.2). A utility function serves two purposes in the adaptation cycle: (1) when the metric values are unacceptable, returns zero to indicate a violated goal, and (2) when the metrics satisfy the minimum, returns a positive value less than one to indicate the user's preference for improvement. Therefore, utility is not only used to initiate adaptation, but also to perform trade-off analysis between competing feature selections, such that an optimization of the system can be achieved.

6.2 Plan

To achieve the adaptation objectives, FUSION relies on the knowledge base to generate a tailored problem:

- Given a violated goal, we use the knowledge base to eliminate all of the features with no significant impact on the goal. We call the list of features that may affect a given goal *Shared Features*. Consider a situation in the TRS where G_2 is violated. By referring to column M_{G2} in Table 2, we can eliminate feature F_3, since it has no impact on G_2's metric. In this example *Shared Features* = $\{F_1,F_2,F_4\}$.
- *Shared Features* represent our adaptation parameters. These features may also affect other goals, the set of which we call the *Conflicting Goals*. To detect the conflicts, again we use the knowledge base, except this time we backtrack the learned relationships. For each feature in the *Shared Features* we find the corresponding row in Table 2, and find the other metrics that the feature affects. In the above example, we can see that features F_1, F_2, and F_4 also affect metrics M_{G1} and M_{G4}, and hence the corresponding goals, G_1 and G_4.

By using the knowledge base, FUSION generates an optimization problem customized to the running software. The objective is to find a selection of *Shared Features*, F^*, that maximizes the system's overall utility for the *Conflicting Goals* as follows:

$$F^* = argmax_{(F \in SharedFeatures)} \sum_{\forall g \in Conflicting\ Goals} U_g\,(M_g(F))$$

where U_g represents the utility function associated with the metric M_g for goal g (recall Figure 1a). Since we do not want the solution to violate any of the conflicting goals, the problem is subject to:

$$\prod_{\forall g \in Conflicting\ Goals} U_g(M_g(F)) > 0$$

Note that we do not need to include the goals that are unaffected by *Shared Features*. To prevent feature selections that violate the mutual exclusion, we specify the following constraint:

$$\forall group \in feature\ model, \sum_{\forall f_c \in group} f_c \leq 1$$

Here when more than one feature from the same mutually exclusive group is selected, the left hand side of the inequality brings the total to greater than 1 and violates the constraint. Finally, we ensure the dependency relationship as follows:

$$\forall f_{child} \in Shared\ Features,\ f_{parent} - f_{child} \geq 0$$

This inequality does not hold if a child (dependent) feature is enabled without its parent being enabled. Applying this formulation to the TRS scenario in which G_2 is violated generates the following optimization problem:

Shared features = $\{F_1, F_2, F_4\}$

$argmax_{(F)}\ U_{G1}(M_{G1}(F)) + U_{G2}(M_{G2}(F)) + U_{G4}(M_{G4}(F))$

Subject to: $U_{G1}(M_{G1}(F)) \times U_{G2}(M_{G2}(F)) \times U_{G4}(M_{G4}(F)) > 0$

$F_3 + F_4 \leq 1$

Where: $M_{G1}=1.553\ F_1 - 0.673\ F_2 + 0.709\ F_3 + 0.163\ F_1F_3 - 0.843$

$M_{G2}=1.137\ F_1 - 0.938\ F_2 - 0.174\ F_4 - 0.161$

$M_{G4}=1.548\ F_1 - 0.488$

Note that by eliminating U_{G3} and F_3 from the optimization problem, we obtain an optimization problem tailored to the violated goals. The customized problem has less number of features and goals than the original problem. In our small example, the gain may not seem significant. However, as shown in Section 8, in large software systems pruning the optimization problem achieves significant performance gains.

6.3 Effect

Once an optimal feature selection is determined, the *Effect* activity is initiated to make the system transition from the current feature selection to the new one. *Effect* chooses a path containing several *adaptation steps* (transitions) towards the new feature selection. The steps take one of the three forms: *enable* and *disable* an optional feature, or *swap* two mutually exclusive features. Figure 4 shows an adaptation path that takes the TRS system from feature selection "1010" to "0101" in three steps.

Since there are many possible paths to reach a target feature selection, the *Effect* component is responsible for picking a path that satisfies feature model constrains in addition to system goals. In the above example, enabling F_3 and F_4 at the same time produces a feature selection that violates the mutual exclusion relationship in the feature model. If two features are mutually exclusive, the system should never be in a state where both features are enabled. Similarly, a dependent feature is never enabled without its prerequisite (parent) being enabled first.

7. IMPLEMENTATION

Figure 5 shows snapshots of a prototype implementation of FUSION. This figure closely matches the structure of Figure 1, and illustrates the realization of the modeling concepts in FUSION. To streamline the development of tool support for FUSION, we have adopted, extended, and integrated existing tools to the extent possible.

We have provided support for FUSION's modeling methodology by extending XTEAM [3]. XTEAM supports modeling of software architectures using well-known Architectural Description Languages (ADLs). It supports Finite State Processes (FSP) and eXtensible Architecture Description Language (xADL) for modeling the behavioral and structural properties, respectively. A snapshot of xADL model for a subset of TRS is shown in Figure 5c. The metrics are specified in terms of the properties associated with the architectural constructs. For example, the response time of a given execution scenario is modeled as a summation of the computational delay of its components.

We have enhanced XTEAM with support for modeling goals (Figure 5a) and features (Figure 5b). As the arrow in Figure 5b indicates, an engineer specifies a mapping for each feature to the underlying architectural model snippet that realizes it. The model snippet uses references to the constructs in the core architectural model to specify the variation introduced by the corresponding feature. For example, Figure 5c shows the impact of selecting the *Caching* feature on the core architectural model, i.e., it results in the addition of a new *Cache* connector in between *AgentDiscovery* and *BusinessTier*.

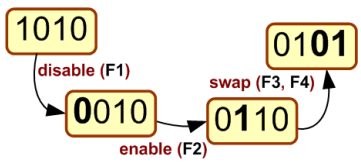

Figure 4. Stepwise adaptation.

When FUSION selects a set of features, the

architectural snippets are weaved with the base (core) architectural model to form the complete architecture of the system. The generated architectural models are used at run-time (i.e., kept synchronized) with an implementation of the system running on top of Prism-MW [14]. Prism-MW is a middleware platform with extensive support for monitoring and dynamic adaptation. FUSION and the running system are integrated as follows: (1) *Monitoring*: Prism-MW's monitor services provide the information for FUSION in terms of raw readings of the metrics. (2) *Adaptation*: Whenever FUSION changes the feature selection, a new architectural model is generated, an *architectural diff* is performed, and the differences are effected through the dynamic adaptation services of Prism-MW. FUSION sends the change requests in small steps (recall Figure 4) to avoid feature violations during the adaptation.

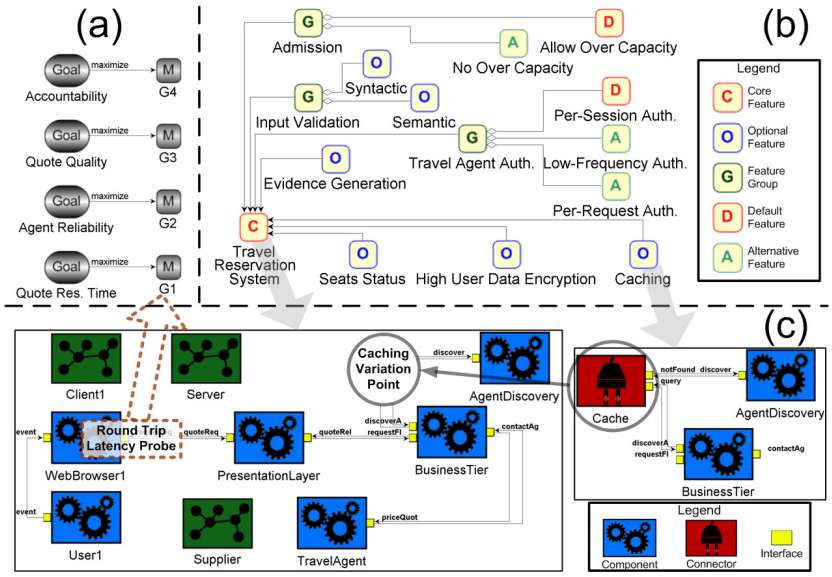

Figure 5. Subset of TRS in our prototype implementation of FUSION: (a) goals and metrics, (b) feature model, (c) implementations of Core and *Caching* features.

Finally, we have integrated the FUSION's modeling environment with WEKA [20], which provides an open source implementation of a number of learning algorithms [9] leveraged in our work.

8. EVALUATION

We have evaluated a prototype implementation of FUSION described in the previous section using an extended version of TRS, which consisted of 78 features and 8 goals. To evaluate FUSION's ability to learn and adapt under a variety of conditions, we set up a controlled environment. We used XTEAM to simulate the execution context of the software (e.g., workload) as well as the occurrence of unexpected events (e.g., database indexing failure). However, note that neither the TRS software nor FUSION was controlled, which allowed them to behave as they would in practice. FUSION was executed on a dedicated Intel Quad-Core processor machine with 5GB of RAM. We conducted the evaluation under four different execution scenarios, which correspond to the four possible situations FUSION may face:

(NT) Similar context—the system is placed under a workload setting that is comparable to that used during FUSION's training. We use a scenario, called *Normal Traffic (NT),* in which the system is invoked with the typical expected number of requests.

(VT) Varying context—the system is placed under a workload setting that is changing at run-time and different from that used during FUSION's training. We use a scenario, called *Varying Traffic (VT),* in which the system is invoked with a continuously changing inter-arrival rate of price quote requests.

(IF) Unexpected event with emerging pattern—the system faces an unexpected change, which results in a new behavioral pattern (i.e., change in the impact of features on metrics) that can be learned. We use a scenario, called database *Index Failure (IF)*, in which the index of a database table used by the *Agent Discovery* component during the execution of the make quote workflow (see Figure 1c) fails, and forces a full table scan.

(DoS) Unexpected event with no pattern— the system faces an unexpected change, which results in new random behaviors that cannot be accurately learned. We use a scenario, called *Randomized DoS Traffic (DoS),* in which the system is flooded with an online denial of service attack that does not follow a pattern (e.g., does not correspond to an exponential distribution).

In our evaluation, an *observation* corresponds to an adaptation decision and its effect. It consists of (1) a new feature selection, and (2) the predicted and actual impact of the feature selection on metrics. An *observation error* with respect to a metric is the difference between predicted and actual value. In the experiments reported here, learning is initiated if the average error in 10 most recent observations is more than 5%. Other learning initiation policies are also possible and would present a tradeoff between learning overhead and accuracy.

8.1 Accuracy of Learning

Figure 6 shows the observation error for the *Quote Response Time* metric in the four scenarios described earlier. The models selected for comparison are: (1) offline learning, which corresponds to a static learning model that is based on the same observations used to train FUSION at design-time; and (2) Queueing Network (QN) model, which assumes that workload and service demand parameters follow an exponetial distribution.

Note that since each feature selection may result in a different architectural model, and hence a different QN model, incorporating QN in our experiments was challenging. In particular, a large number of QN models is neededwould have to be developed (we estimated a total of 26×10^{12} valid feature selections from the total search space of $2^{78} \approx 30\times10^{22}$), which corroborates our earlier assertion about the unwieldiness of using analytical models. This is while performance may be only one goal of interest out of many in the system. In our accuracy comparisons reported below we constructed a subset of QN models that correspond to the feature selections made by FUSION.

Figure 6a shows the TRS system under the NT scenario, where both FUSION and offline learning come to less than 5% error on average (i.e., average of the last 10 observations, which as

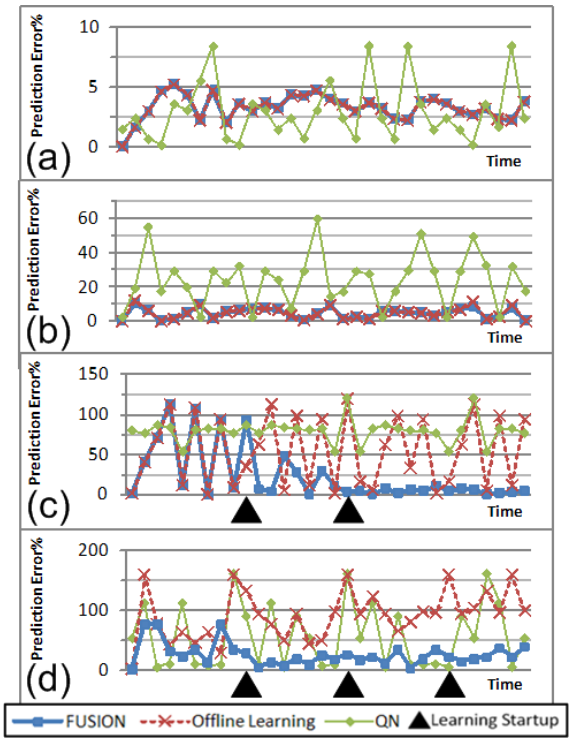

Figure 6. Accuracy of learned functions for "Quote Response Time" metric: (a) Normal Traffic, (b) Varying Traffic, (c) Database Indexing Failure, (d) Randomized DoS Traffic.

mentioned earlier is the criteria for initiating run-time learning). As exptected, this indicates that both FUSION and offline learning achieve good level of accuracy under expected execution conditions. QN also shows relatively good level of accuracy with average error rate of 2.9% and some spikes of 5-8% errors.

Figure 6b shows the TRS system under the VT scenario. This shows that even when the workload changes significantly, FUSION's observation error remains within 5% error rate on average. As a result, a new behavioral pattern sufficient for run-time learning never emerges. On the contrary, in the case of QN, the wrong assumptions about service demands exacerbate the prediction errors.

Figure 6c shows the TRS system under the IF scenario. It shows that when there are unexpected events in the system, FUSION is capable of learning the new behavior and adjusting its model. FUSION's error rate increases up to 54% at the beginning of the execution scenario. This error could be attributed to the fact that the model did not anticipate the impact of *Caching* feature when the table scans were taking place in the *AgentDiscovery* component. As you may recall from Figure 1, *Caching* reduces the need for agent discovery, hence it is more effective in reducing the response time due to a full table scan for each discovery. *Caching* was estimated to be responsible for 35% of FUSION's prediction error. Gradually, FUSION fine-tunes the coefficient of *Caching* and other features in the learned functions. As a result, the observation error rate goes down to less than 5% on average and the system reaches a steady state . In constrast, the prediction error of QN reaches 80%, since the QN model presumes the existence of a table index (i.e., the service demand of the queue representing the database in the model is different).

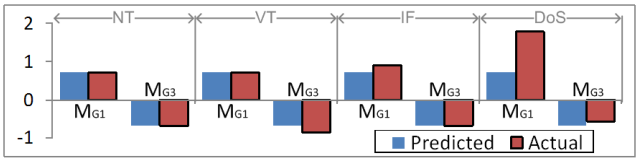

Figure 7. Impact of Feature "Per-Request Authentication" on Metrics "Quote Response Time" and "Quote Quality".

Figure 6d shows the the DoS scenario. The random nature of traffic, makes it impossible for FUSION to converge to an induced model that can predict the behavior of the system within the on average 5% error rate goal. As soon as a new model is induced, the execution conditions change, making the prediction models inaccurate. As a result, FUSION's learning cycle is periodically invoked to induce. Even though FUSION does not reach the same level of accuracy as in the other execution scenarios, it is still significanlty better than QN and offline learning. This can be attributed to the fact that FUSION is benefiting from the continuous tuning, although it loses accuracy in the absense of a stable pattern.

8.2 Adaptation in Presence of Inaccuracy

Clearly the quality of adaptation decisions depends on the accuracy of induced models. However, when the execution context changes, the model is forced to make some adaptation decisions under uncertainty, which are in turn used to fine-tune the induced models and account for the emerging behavior. An important concern is whether the adaptation decisions made during this period of time (i.e., using an inaccurate model) could worsen the violated goals or not. Figure 7 shows the normalized impact of enabling F_3 on metrics M_{G1} and M_{G3} in the first observation for each of the four scenarios of Figure 6. Recall from Figure 6 that the first observation for IF and DOS correspond to a situation when there is a high-level of inaccuracy. In all cases, FUSION disables F_3 with the purpose of increasing M_{G1} and reducing M_{G3}. While due to the inaccuracy of the induced model FUSION fails to predict accurately the magnitude of impact on these metrics, it gets the general direction of impact (i.e., positive vs. negative) correctly. This result is reasonable since a given feature typically has a similar general impact on metrics. For instance, one would expect an authentication feature to improve the system's security, while degrading its performance. Hence, even in the presence of inaccurate knowledge, FUSION does not make decisions that worsen the situation. Instead it makes decisions that are good, but not necessarily optimal, until the knowledge base is refined.

8.3 Overhead of Learning

FUSION enables adjustment of the system to changing conditions by continuously incorporating observation records in the learning process. An important concern is the execution overhead of the online learning. One of the principle factors affecting learning overhead is the number of observations required to make accurate inductions. Table 3 lists the execution time for a given number of observations. Simple linear regression takes insignificant amount of time with large number of observations, which makes it an appealing choice when there is a large number of observation (e.g., initial training at design-time, when large number of observations can be obtained). In our experiments FUSION performed online learning on a maximum of 10 observations, which from Table 3 could be verified to have presented an insignificant overhead of less than 20 milliseconds. This

Table 3. Induction execution time in milliseconds

# of Observations	50	500	528	822	903	1227	1809
M5 Model Tree	60	110	130	130	130	160	230
Linear Regression	20	30	30	50	60	70	80

efficiency is due to the pruning of the feature space and significance test described in Section 5.

8.4 Quality of Feature Selection

We evaluate the quality of solution (feature selection) found by FUSION against two competing technqiues. The first technique is *Traditional Optimization* (*TO*), which maximizes the global utility of the system, and includes all of the feature variables and goals in the optimization problem. The second technique is *Constraint Satisfaction* (*CS*), which finds a feature combination that satisfies all of the goals, regardless of the quality of the solution. As you may recall from Section 6, FUSION adopts a middle ground with two objectives: (1) find solutions with comparable quality to those provided by TO, but at a fraction of time it takes to executing TO, and (2) find solutions that are significantly better in quality than CS (i.e., stable fix), but with a comparable execution time.

Figure 8 plots the global utility obtained from running the optimization at 3 different points in time for each of the 4 evaluation scenarios discussed earlier. Each data point represents the global utility value (recall the objective function in Section 6.2) obtained for each experiment. FUSION produces solutions that are only slightly less in quality than TO in all of the experiments. Note that TO finds the optimal solution. This demostrates that our feature space pruning heuristics do not significnatly impact the quality of found solutions. Table 4 shows the average number of features that are considered for solving each of the experiments, which is only a small fraction of the entire feature space. Figure 8 also shows that FUSION find solutions that are significantly better than CS. In turn, this corroborates our assertion in Section 6.2 that FUSION produces a stable fix to goal violations by placing the system in a near-optimal configuration. On the other hand, since CS may find borderline solutions that barely satisfy the goals, due to slight fluctuations in the system, goals may be violated and thus frequent adaptations of the system ensue.

Finally, we should point out that the quality of solutions found in all of the methods, including FUSION's approach, depends on the accuracy of induced model. In particular, FUSION's feature space pruning heuristics depend on this. In fact, the spike in the number of DoS features that are considered in Table 4 demonstrates that feature space pruning heuristics were not as successful as other scenarios where a more accurate knowledge base was available.

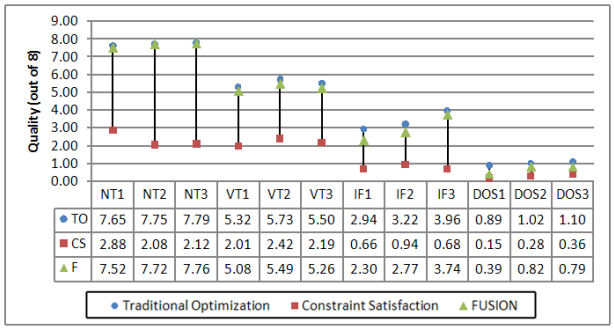

Figure 8. Global utility for different scenarios.

Table 4. Effect of feature reduction heuristics.

# of Features Considered	NT 1	NT 2	NT 3	VT 1	VT 2	VT 3	IF 1	IF 2	IF 3	DoS 1	DoS 2	DoS 3
FUSION	3	1	2	3	4	5	3	4	5	6	13	11
CS / TO	78											

8.5 Efficiency of Optimization

In Section 6.2 we described how FUSION achieves efficient anaysis by using the knowledge base to dynamically tailor an optimization problem to the violated goals in the system. In comparison, TO conducts a full optimization problem where the complexity of the problem is $O(2^F)$. Figure 9 shows the execution time for solving the optimization problem in FUSION, TO, and CS for the same instances of TRS as those shown in Figure 8 and Table 4. Note that the execution time of FUSION is comparable to CS and is significanly faster than TO. This in turn along with the results shown in the previous section demonstrates that FUSION is not only able to find solutions that are comparable in quality to those found by TO, but also achieves this at a speed that is comparable to CS. Note that since TO runs exponentially in the number of features, for systems with slighlty larger number of features, TO could take several hours for completion, which would make it inapplicable for use at run-time.

9. RELATED WORK

Over the past decade, researchers and practitioners have developed a variety of methodologies, frameworks, and technologies intended to support the construction of self-adaptive systems [2]. We provide an overview of the most notable research in this area and examine them in light of FUSION.

Architecture-based adaptation. IBM's Autonomic Computing initiative advocates a reference model known as MAPE [10], which is structured as hierarchical levels of feedback-control loop consisting of the following activities: Monitor, Analyze, Plan, and Execute. Oreizy et al. pioneered the architecture-based approach to run-time adaptation and evolution management in their seminal work [16]. Garlan et al. present Rainbow framework [5], a style-based approach for developing reusable self-adaptive systems. Rainbow monitors a running system for violation of the invariant imposed by the architectural model, and applies the appropriate adaptation strategy to resolve such violations. All of the above approaches, including many others (e.g., see [2,15]), share three traits: (1) use analytical models for making adaptation decisions, and (2) rely on architectural models for the analysis, and (3) effect a new solution through architecture-based adaptation. These works have clearly formed the foundation of our work and have guided our research as manifested by the key role of architecture in FUSION. However, unlike these approaches, FUSION adopts a

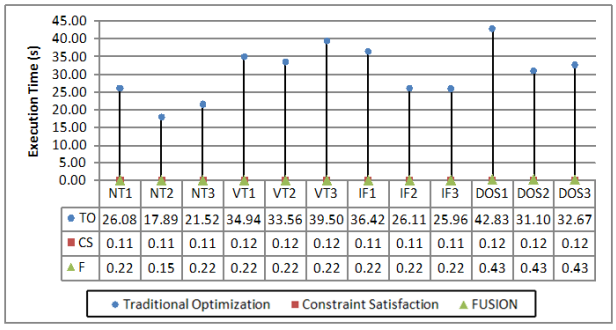

Figure 9. Optimization execution time for different scenarios.

feature-based approach to analysis and adaptation, which not only makes learning feasible, but also makes the analysis efficient and reduces the effort required in applying FUSION to existing systems. Moreover, unlike them, FUSION is capable of coping with unanticipated changes through learning.

Policy-based adaptation. Related to our research are adaptation frameworks that employ logic and policy based methods of induction. Sykes et al. [18] present an online planning approach to architecture-based self-managed systems. Based on a three-layer model for self-management [13], their work describes plan (i.e., a set of condition-action rules) generation with respect to a change in the environment or a system failure. Georgas and Taylor [6] present a knowledge-based approach, such that the adaptation polices are specified as logic rules, which are in turn leveraged to induce new policies. These approaches bear resemblance to our work in their use of induction. While policy-based approaches have been shown useful in some settings (e.g., ensuring certain properties hold in the system), they cannot be used for making quantitative analysis of QoS trade-offs. These approaches may also suffer from conflicting rules in the knowledge base.

Reinforcement learning adaptation. Finally, related to our work are autonomic approaches that have employed reinforcement learning. Kim and Park [11] propose a reinforcement learning-based approach to online planning for robots. Their work focuses on improving the robot's behavior by learning from prior experience and by dynamically discovering adaptation plans in response to environmental changes. Tesauro et al. [19] have proposed a hybrid approach that combines queueing network with reinforced learning to make resource allocation decisions in data centers. FUSION provides a general-purpose framework for self-adaption of any feature-oriented application software, which is fundamentally different from these domain-specific solutions.

10. CONCLUSION

We presented FUSION, a new method of engineering self-adaptive systems that combines feature-orientation, learning, and dynamic optimization to alleviate some of the crucial challenges in this setting. Instead of relying on analytical models that are unwieldy for use and subject to wrong assumptions, FUSION uses online learning to analyze and self-tune the adaptive behavior of the system to unanticipated changes. Learning is enabled by a dynamic feature-oriented representation of the system that incorporates the engineer's knowledge of the application and its domain. Learning in turn enables FUSION to dynamically tailor the optimization problem to the violated goals, and hence achieve efficiency of analysis without trading accuracy. Using a prototype implementation of the system and a travel reservation system we have extensively validated the approach and its properties.

In our future work, we intend to investigate opportunistic self-training as a way to detect emerging behaviors before adaptation decisions are made. We are exploring a self-training method that takes place using a shadow clone of the running system during periods of low utilization. In addition, we intend to empirically compare FUSION against other self-adaptation frameworks.

11. ACKNOWLEDGMENTS

This work is partially supported by grant CCF-0820060 from the National Science Foundation.

12. REFERENCES

[1] Carroll, D.J. 2002. *Statistics Made Simple for School Leaders: Data-Driven Decision Making*. ScarecrowEducation.

[2] Cheng, B., et al. 2009. Software Engineering for Self-Adaptive Systems: A Research Roadmap. *Software Engineering for Self-Adaptive Systems, LNCS*. 1-26.

[3] Edwards, G., Malek, S., Medvidovic, N. 2007. Scenario-Driven Dynamic Analysis of Distributed Architectures. *Int'l Conf. on Fundamental Approaches to Software Engineering* (Braga, Portugal, March 2007), 125.

[4] Friedman, J.H. and Roosen, C.B. 1995. An introduction to multivariate adaptive regression splines. *Statistical Methods in Medical Research*. 4, 3 (Sep. 1995), 197-217.

[5] Garlan, D., Cheng, S.W. et al. 2004. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *IEEE Computer*. 37, 10 (Oct. 2004), 46-54.

[6] Georgas, J.C. and Taylor, R.N. 2004. Towards a knowledge-based approach to architectural adaptation management. *Workshop on Self-healing Systems* (Newport Beach, California, October 2004), 59-63.

[7] Gomaa, H. 2004. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison-Wesley Professional.

[8] Gross, D. and Harris, C.M. 1985. *Fundamentals of queueing theory (2nd ed.)*. John Wiley & Sons, Inc.

[9] Jordan, M.I. and Jacobs, R.A. 1994. Hierarchical mixtures of experts and the EM algorithm. *Neural Comput.* 6, 2 (1994), 181-214.

[10] Kephart, J.O. and Chess, D.M. 2003. The Vision of Autonomic Computing. *IEEE Computer*. 36(1), 41-50.

[11] Kim, D. and Park, S. 2009. Reinforcement learning-based dynamic adaptation planning method for architecture-based self-managed software. *Workshop on Softw. Eng. For Adaptive and Self-Managing Systems* (Vancouver, Canada, May 2009), 76-85.

[12] Kleppe, A., Warmer, J. et al. 2003. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Professional.

[13] Kramer, J. and Magee, J. 2007. Self-Managed Systems: an Architectural Challenge. *Int'l Conf. on Software Engineering* (Minneapolis, MN, May 2007), 259-268.

[14] Malek, S., et al. 2005. A Style-Aware Architectural Middleware for Resource-Constrained, Distributed Systems. *IEEE Trans. Softw. Eng.* 31, 3 (2005), 256-272.

[15] Menascé, D.A., Ewing, J.M. et al. 2010. A framework for utility-based service oriented design in SASSY. *Joint WOSP/SIPEW Int'l Conf. on Performance engineering* (San Jose, CA, January 2010), 27-36.

[16] Oreizy, P., Medvidovic, N., Taylor, R. 1998. Architecture-based runtime software evolution. *Int'l Conf. on Software Engineering* (Kyoto, Japan, April 1998), 177-186.

[17] Poladian, V., et al. 2004. Dynamic Configuration of Resource-Aware Services. *Int'l Conf. on Software Engineering* (Scotland, UK, May 2004), 604-613.

[18] Sykes, D., et al. 2008. From goals to components: a combined approach to self-management. *Int'l Workshop on Software Engineering for Adaptive and Self-Managing Systems* (Leipzig, Germany, May 2008), 1-8.

[19] Tesauro, G., et al. 2006. A Hybrid Reinforcement Learning Approach to Autonomic Resource Allocation. *Int'l Conf. on Autonomic Computing* (Dublin, Ireland, June 2006), 65-73.

[20] WEKA. *http://www.cs.waikato.ac.nz/ml/weka/*.