A Learning-Based Framework for Engineering Feature-Oriented Self-Adaptive Software Systems

Naeem Esfahani, Ahmed Elkhodary, and Sam Malek, *Member, IEEE*

Abstract—Self-adaptive software systems are capable of adjusting their behavior at runtime to achieve certain functional or quality of service goals. Often a representation that reflects the internal structure of the managed system is used to reason about its characteristics and make the appropriate adaptation decisions. However, runtime conditions can radically change the internal structure in ways that were not accounted for during their design. As a result, unanticipated changes at runtime that violate the assumptions made about the internal structure of the system could degrade the accuracy of the adaptation decisions. We present an approach for engineering self-adaptive software systems that brings about two innovations: (1) a feature-oriented approach for representing engineers' knowledge of adaptation choices that are deemed practical, and (2) an online learning-based approach for assessing and reasoning about adaptation decisions that does not require an explicit representation of the internal structure of the managed software system. Engineers' knowledge, represented in feature-models, adds structure to learning, which in turn makes online learning feasible. We present an empirical evaluation of the framework using a real world self-adaptive software system. Results demonstrate the framework's ability to accurately learn the changing dynamics of the system, while achieving efficient analysis and adaptation.

Index Terms— Self-Adaptive Software, Autonomic Computing, Feature-Orientation, Machine Learning

1 Introduction

THE unrelenting pattern of growth in size and complexity of software systems that we have witnessed over the past few decades is likely to continue well into the foreseeable future. As software engineers have developed new techniques to address the complexity associated with the construction of modern-day software systems, an equally pressing need has risen for mechanisms that automate and simplify the management and modification of software systems after they are deployed, i.e., during runtime. This has called for the development of *self-adaptive software systems* [1]. A self-adaptive software system is capable of modifying itself at runtime to achieve certain functional or Quality of Service (QoS) objectives. While over the past decade researchers have made significant progress with methodologies and frameworks [1–6] that target the development of such systems, numerous challenges remain [7]. In particular, engineering the adaptation logic poses the most difficult challenge as further discussed below.

The state-of-the-art [1], [7] in engineering self-adaptive software systems is to employ an architectural representation of the software system (e.g., component-and-connector view [8]) for reasoning about the adaptation decisions. We refer to this as the *white-box approach*, since it requires knowledge of the managed system's internal structure. The adaptation decisions are thus made at the architectural level, often in terms of structural changes, such as adding, removing, and replacing software components, changing the system's *architectural style* [9], rebinding a component's interfaces, and so on. This paradigm is commonly referred to as *architecture-based adaptation* [1], [6], [8]. At design-time engineers create analytical models using this architectural representation. The analytical model is then used to assess the system's ability to satisfy an objective using the monitoring data obtained at runtime. The result produced by an analytical model thus serves as an indicator for making adaptation decisions. For instance, Queuing Network models [10] and Hidden Markov models [11] have been used previously for assessing the system's performance and reliability properties, respectively.

While state-of-the-art approaches have achieved noteworthy success in many domains, they suffer from key shortcomings when faced with the following issues:

1. **Concept drifts.** White-box approaches, in general, make simplifying assumptions or presume certain properties of the internal structure of the system that may not bear out in practice. They cannot cope with the runtime changes (i.e., *concept drifts* [12], [13]) that were not accounted for during their formulation. In practice, the internal structure of a managed software system may not be completely known at design time. Even when known, runtime conditions may radically change the structure or properties of the system in ways that were not accounted for during design. Thus, unanticipated changes at runtime that violate the design-time assumptions could make the analysis and hence the adaptation decisions inaccurate.
2. **Dependencies.** To make the construction of self-adaptive systems manageable, majority of the existing approaches assume adaptation can be localized in atomic structural changes that can be carried out independently, while in practice changes in different parts of the architecture need to occur

• *N. Esfahani, A. Elkhodary, and S. Malek are with the Department of Computer Science, George Mason University, Fairfax, VA 22030-4444, USA. E-mails: { nesfaha2, aelkhoda, smalek}@gmu.edu*

in concert. Thus, ensuring the correct functioning during and after the adaptation is difficult using such approaches, in particular when changes crosscut the software system.

3. **Efficiency.** The 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 runtime. But, often searching for an optimal configuration (i.e., solution) at the architectural-level is computationally very expensive. In fact, many architecture-based optimization algorithms are shown to be NP-hard [5], [14]. We argue this is because architectural models do not provide an effective medium for representing the engineer's knowledge of practical alternatives, hence forcing the automated analysis to explore a large number of invalid configurations.

In this paper, we provide a comprehensive description of a *black-box approach* for engineering self-adaptive systems that was first introduced in our prior work [15]. By *black-box* we mean that the adaptation decisions are made using abstractions that do not require knowledge of the internal structure of the software system. From the perspective of Kramer and Magee's reference architecture for self-management [1], a black-box approach results in a clear separation of models used for *goal management* and those used for *change management*. Although to map the abstractions used for goal management to those used for change management, knowledge of the system's internal structure is likely to be needed.

The approach brings about two innovations for solving the aforementioned challenges: (1) a new method of modeling and representing self-adaptive software systems that builds on the notion of *feature-orientation* from the product line literature [16], and (2) a new method of assessing and reasoning about adaptation decisions through online learning [17]. The result of this research has been a framework, entitled FeatUre-oriented Self-adaptatION (FUSION), which combines feature-models with online machine learning. Domain expert's knowledge, represented in feature-models, adds structure to online learning, which in turn improves the accuracy and efficiency of adaptation decisions. The key contributions of the FUSION framework are as follows:

1. FUSION copes with the changing dynamics of the system, even those that are unforeseen at design time, through incremental observation and induction (i.e., online learning).
2. By encapsulating the engineer's knowledge of the inter-dependencies among the system's constituents, FUSION can ensure stable functioning and protect system goals during and after adaptation.
3. FUSION uses features and inter-feature relationships to significantly reduce the configuration space of a sizable system, making runtime analysis and learning feasible.

This paper describes several new non-trivial extensions to the preliminary version of FUSION described in [15]: (1) a more expressive feature-modeling language that incorporates several additional commonly used feature-modeling notations, (2) incorporation of additional machine learning techniques, in particular one that is applicable to discrete values, (3) a brand new algorithm that uses the learned knowledge to manage the adaptation of software, while minimizing disruptions to the system, (4) a prototype of an environment that supports building of self-adaptive software systems that are managed via FUSION, and (5) additional empirical evaluations to assess the new algorithms and capabilities. On top of these technical contributions, the paper provides an in-depth description of FUSION, such as a section on alternative means of realizing features in a software system, a detailed discussion on how feature-oriented adaptation makes learning possible, and a revamped discussion of FUSION in the context of related research.

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 detail FUSION's feature-oriented model of adaptation, learning method, and adaptation planning, respectively. Sections 7, 8, and 9 present the implementation of FUSION, experiment setup, and evaluation details respectively. The paper concludes with a discussion of the threats to validity, an overview of the related work and avenues of future research.

2 Motivation

For illustrating the concepts in this paper, we use an online Travel Reservation System (TRS) that provides a web portal for making travel reservations remotely. Figure 1c shows a subset of this system's software architecture using the traditional component-and-connector view [8]. TRS aims to provide the best airline ticket prices in the market. To prepare a price quote for the user, TRS takes the trip information, and then discovers and queries the appropriate travel agent services. The travel agents reply with their itinerary offers, which are then sorted and presented in ascending order of the quoted price.

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

A system such as TRS needs to be self-adaptive to deal with unanticipated situations, such as traffic spikes or security attacks. To that end, the adaptation logic of TRS would need to select from the available adaptation choices. For instance, enable caching to improve performance during a traffic spike, strengthen authentication to thwart a security attack, and adjusting the logging level to ensure non-repudiation of transactions (i.e., accountability). To do so, heterogeneous analytical models are required. For example, security engineers may use attack graphs to prevent intrusions and find the best counter measures, while performance engineers may use queuing network models to assess the latency goals. For a complex system, engineers may need to connect analytical models at mul-

tiple layers of abstraction (i.e., network, software, user, etc.) to characterize the system's behavior. The construction of adaptation logic for a system in this way is challenging for the following reasons.

Concept Drifts. Consider a Queuing Network (QN) model that quantifies the impact of an adaptation decision on the response time of receiving price quotes from travel agents (thick arrows 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 a particular architectural layout is assumed, such a model may be unaware of the delay/overhead of communication and estimate the response time simply as summation of the execution time associated with the participating components. A more elaborate model would also include the hardware layer details, but potentially for a presumed architectural layout (e.g., physical hardware versus Virtual Machines deployed on a shared pool of hardware), and so on. Since the underlying characteristics of complex dynamic systems change at runtime, design-time assumptions on the structure of the system may not hold, making the analysis and hence the adaptation decisions inaccurate.

Dependencies. Ensuring the correct functioning of the software system during and after the adaptation is a challenging task. This is often dependent on the application and cannot be represented effectively in the general-purpose architectural description languages [18]. For instance, consider the problem of representing a constraint in TRS that requires the same authentication protocol to be used on an end-to-end execution flow from the *Web Portal* all the way to the *Travel Agent* and back (thick lines in Figure 1c). Prior to switching to a new protocol, the system is required to negotiate new credentials among all of the components involved in the execution flow. The fact that this authentication protocol crosscuts multiple components is difficult to represent and enforce using architectural constructs (i.e., by introducing separate components and connectors).

Efficiency. To satisfy multiple goals, self-adaptation logic needs to search in a configuration space that is equivalent to the combined complexity of all possible architectural choices. As an example, consider how a hypothetical system would make use of N authentication components for authenticating the network traffic between its M software components, which may be deployed on P different hardware platforms. In this case, analyzing the impact of authentication alone on the system's goals would require exploring a space of $(M^P$ *possible deployments)* $^{N \text{ possible ways of authentication}} = M^{NP}$ *possible configurations*. Such a problem is computationally expensive to solve at runtime for any sizable system. This is while authentication is only one concern out of many in a typical system.

These difficulties motivate our work, which instead of a pre-specified analytical model uses a feature-oriented representation of the system to continuously learn the impact of adaptation choices on system's goals and adjust the induced models.

3 FUSION Overview

Figure 2 depicts the framework as it adapts a running system composed of a number of features. We assume the running system is variable in the sense that features can be "selected" and "deselected" on demand. FUSION makes new feature selections 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. The details of how features and goals are modeled are discussed in Section 4.

As depicted in Figure 2, FUSION makes such adaptation decisions using a continuous loop, called *adaptation*

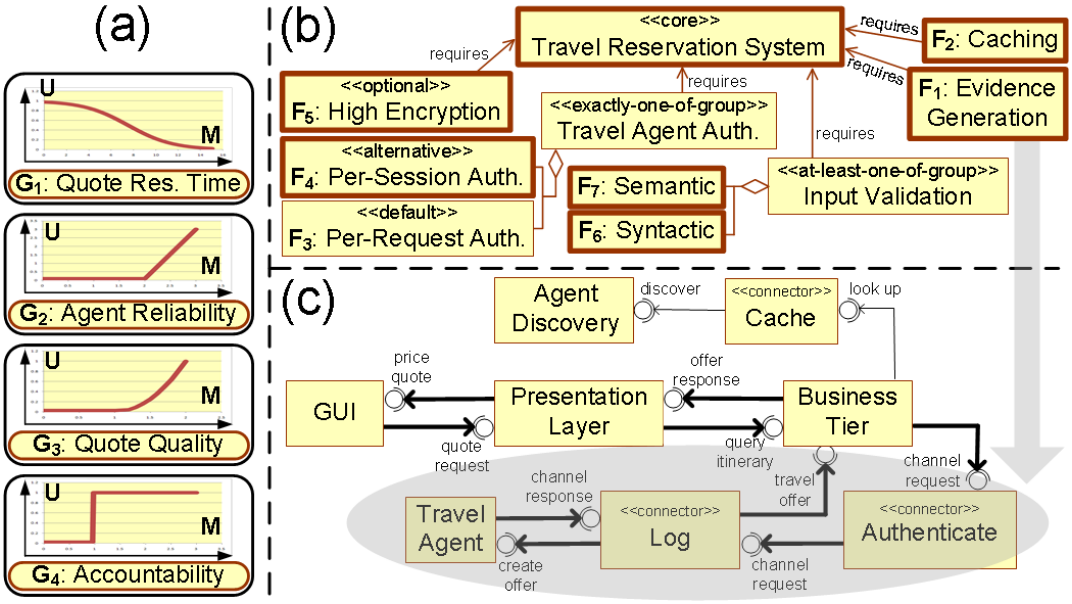

Figure 1. Travel Reservation System: (a) goals are quantified in terms of utility obtained for a given level of metric; (b) subset of available features, where features with thick borders are selected; and (c) software architecture corresponding to the selected features, where the thick lines represent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

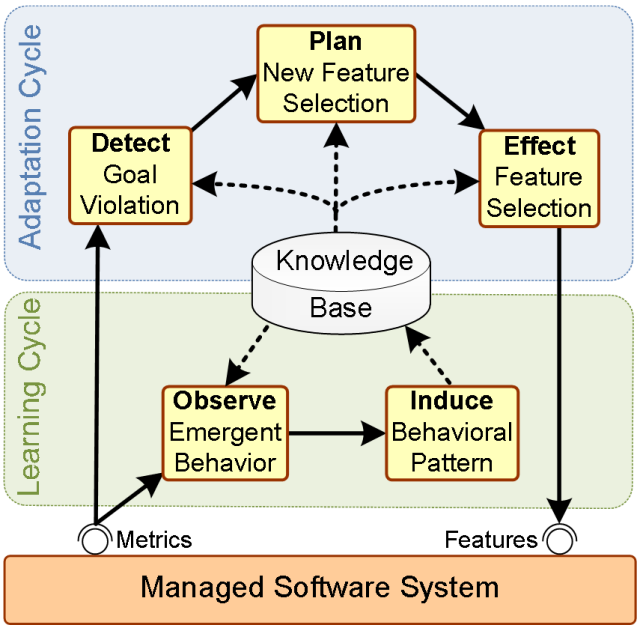

Figure 2. Overview of the FUSION framework.

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 (i.e., feature selection) that maximizes the overall system utility.
- Given a new feature selection, *Effect* determines a set of adaptation steps (i.e., enabling/disabling of features) to minimize disruptions that negatively impact the system's goals.

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 are collected. This data is used to train FUSION to induce a preliminary model of the system's behavior.

At runtime, 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 the following two activities in sequence:

- Based on the measurements collected from the system, *Observe* detects any emerging patterns of behavior. An emergent pattern is detected when predictions set wrong expectations (i.e., inaccurate forecast of the impact of adaptation on utility).
- *Induce* learns the new behavior by applying machine learning on recently collected data and stores a refined model of the behavior in the *knowledge base*, which is then used to make (more) informed adaptation decisions in future cycles.

The input to the various activities in both learning and adaptation cycles is the *knowledge base*, which is comprised of all the models relating to the managed system, including the current configuration (feature selection), QoS goals, and automatically inferred functions relating the impact of features on metrics. In the following three sections, we describe FUSION's underlying model, learning cycle, and adaptation cycle.

4 FUSION Model

We first describe FUSION's modeling methodology, which forms the centerpiece of our approach. As we demonstrate later in this paper, FUSION's feature-oriented models enable effective learning and analysis by allowing engineers to specify key factors in the software system that affect the system goals. Such factors can be at the domain, architecture, or execution platform levels.

4.1 Feature-Oriented Adaptation

In FUSION, the unit of adaptation is a *feature*. A feature is an abstraction of a capability provided by the system. A feature is traditionally used during the requirements engineering phase to model a variation point in the software system [16]. During software construction, the engineer develops a mapping for each feature to a part of the software system that realizes it. A feature may also affect the system's non-functional properties (e.g., response time). We propose an additional role for features at runtime. We use features as the units of adaptation.

A feature provides a granular abstraction of an adaptation point (i.e., runtime variability) in the software system, and since it may crosscut the software system's implementation, it could be used to address the consistency issues during the adaptation. Moreover, since a feature model incorporates the engineer's knowledge of the system (i.e., the interrelationships among the system's functional capabilities), it could be used to reduce the space of valid configurations, and thus make the runtime analysis very efficient. In that sense, features in our approach belong to the solution domain (i.e., software design and construction phase) rather than the problem domain (i.e., requirements phase), where they have traditionally been used in the software product line literature [16]. This notion of features belonging to the solution domain is closely aligned with how features have been used in the dynamic software product line literature [19–21].

The use of feature as an abstraction makes the FUSION framework independent of how adaptation choices are realized, and thereby allows FUSION to treat the managed system as a black-box. For example, features may correspond to configuration parameters that are expressed in configuration files as in Figure 3a. Features may be realized using aspects that are weaved to the running system dynamically when the corresponding feature

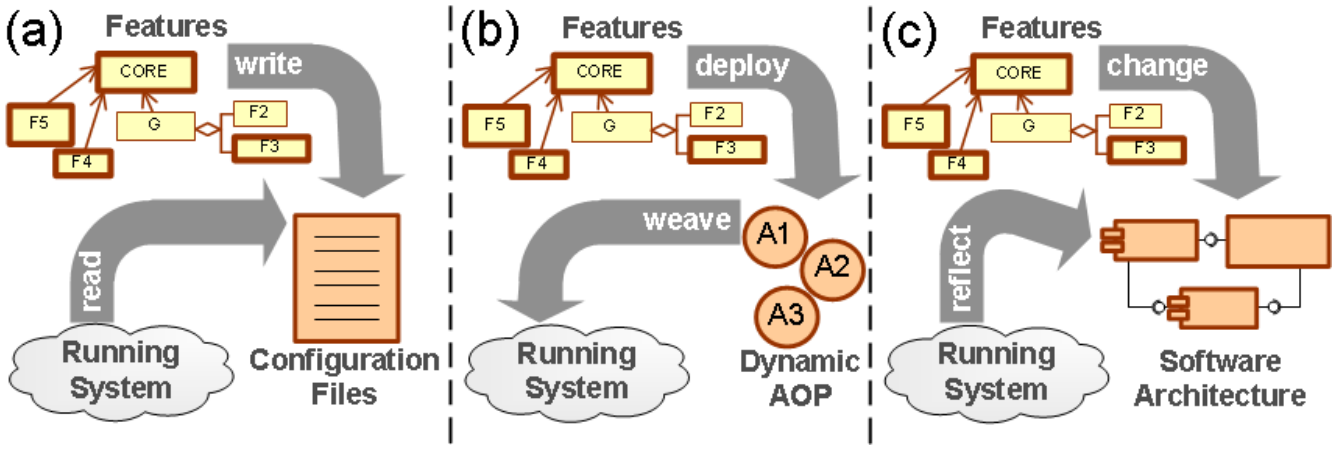

Figure 3. Features can be realized in many ways: (a) configuration parameters modified by the self-management layer at runtime and read by running system; (b) aspects that are weaved into the running system when the corresponding feature is enabled; and (c) architecture mapping, which then modifies the running system by adding/removing components.

is enabled [22] as in Figure 3b. In this paper, we adopt a particular realization of a feature: a feature represents an extension of the architecture at well-defined *variation points* as in Figure 3c. 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 architecture, which then maps to the platform specific representations of the running system.

Figure 1b shows a simple feature model for TRS. There are seven 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. Several other types of feature relationships supported in FUSION are zero-or-all-of, zero-or-one-of, at-least-one-of, and exactly-one-of. Interested reader may find description of these relationships in [16]. Section 7 demonstrates how features are specified and mapped to the underlying software architecture in a tool support chain.

At runtime, 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" and disabled features are set to "0". For example, one possible configuration of TRS would be "1101111", which means that all features from Figure 1b would be enabled except *Per-Request Authentication*.

4.2 Goals

In FUSION, 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. We revisit the issue of how metrics can be obtained from the running system in Section 7.

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). FUSION places one constraint on the range of utility functions: they need to return a value less than zero for the metric values that are not acceptable to the user. As will be discussed in Section 6.1, when a utility is less than or equal to zero, FUSION considers it as the violation of the associated goal and initiates adaptation.

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 [23], [24], and considered to be outside the focus of this paper. FUSION is independent of the type of utility functions and the approach employed in eliciting them. We rely on these works in the development of FUSION's support for the elicitation of user's QoS preference. Some possible methods of eliciting user's preferences include: (1) discrete—select from a finite number of options (e.g., a certain level of QoS for a given service is excellent, good, bad, or very bad), (2) relative—a simple relationship or ratio (e.g., 10% improvement in a given QoS has 20% utility), and (3) constant feedback—input preferences based on the delivered QoS at runtime (e.g., given a certain QoS delivered at runtime ask for better, same, or ignore).

4.3 Contextual Factors

In FUSION, a *contextual factor* is a property of the computational environment that affects the system goals. A typical example of a contextual factor is the system's workload. Workload affects performance related QoS goals, such as response time. Security related goals may also be sensitive to contextual factors. For instance, in the TRS example, the location of a travel reservation agency is used to change the type of authentication. In Section 5.2, we describe how FUSION incorporates the impact of such contextual factors in the management of software.

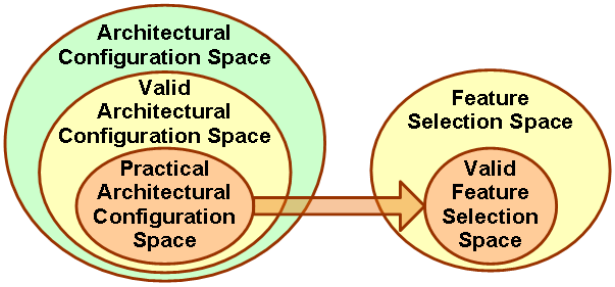

Figure 4: FUSION uses a feature model to incorporate the engineers' knowledge of configurations for the software system that are both valid and practical.

4.4 Implications of Feature-Oriented Adaptation

In FUSION, the features serve as the interface between the adaptation logic and the managed system. Figure 4 depicts how feature-orientation reduces the software adaptation space. In the conventional architecture-based adaptation (i.e., white-box approach), adaptation logic operates on the full *architectural configuration space*, where a vast majority of the configurations are either invalid or simply not practical. Often the *architectural configuration space* is exponential. For instance, recall from Section 2 that a system with N authentication protocols, M software components, which may be deployed on P different hardware platforms, is comprised of $(M^P$ *possible deployments)* $^{N \text{ possible ways of authentication}} = M^{NP}$ possible configurations. Learning in such a large exponential space is infeasible. That is because the number of possible configuration is very large, applying machine learning techniques to infer a predictive model that can determine the impact of a given configuration on multiple quality attributes becomes impossible. Moreover, a large number of configurations in such a setting are not even valid or practical.

In FUSION, instead of operating on the full architectural configuration space, we use features to expose only the configurations that the engineer deems practical and eliminate others. The engineer represents the sensible variation points as features. As mentioned in Section 4.1, features take on boolean values (i.e., a feature can be either "1" = enabled or "0" = disabled). Thus, the full adaptation space is limited to 2^F, where F represents the set of all the features exposed in the system. In addition, feature relationships are used to constrain the space to those combinations that are valid. This produces the *valid feature selection space* that provides a codification of the engineer's knowledge of practical adaptation choices.

In practice, the *valid feature selection space* is several orders of magnitude smaller than the set of all possible configurations for a software system. The key benefit of this reduction is that learning becomes possible. On the other hand, the feature-oriented adaptation limits the scope of runtime adaptation to variations points that have been deemed practical by the engineer prior to system's deployment. Therefore, feature-oriented adaptation exposes only a subset of all possible ways in which a software system can be adapted. These differences are due to the tradeoffs between white-box and black-box adaptation, and each approach has its own advantages. Since our objective in FUSION has been the ability to automatically learn the impact of adaptations on system's properties, we have chosen the black-box adaptation model.

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, along with any other relevant contextual variables, on a metric. In the subset of TRS depicted in Figure 1, 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 and relevant contextual variables as input and produces an estimated value for the metric as output.

Learning is typically a computationally intensive process. In particular, learning 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 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 significantly reduce the *feature selection space* down to the invalid ones, further aiding FUSION to learn their tradeoffs with respect to goals.

For instance, the feature model in Figure 1b yields a space of *48* valid feature selections. Without considering the inter-feature relationships to prune the invalid selections, the space of feature selection would have been $2^{\text{number of features}} = 2^7 = 128$.

The rest of this section describes the two activities that take place to populate and fine-tune the knowledge base.

5.1 Observe

As depicted in Figure 2, *Observe* starts the learning cycle. *Observe* is a continuous execution of two activities: (1) normalize raw metric values to make them suitable for

TABLE 1. NORMALIZED OBSERVATION RECORDS.

Independent Variables					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	..
1	0	0	1	..	0.650	0.513	1.371	2	..
0	1	0	1	..	-1.470	-0.719	1.378	1	..
0	0	1	0	..	-0.132	-0.103	0.740	0	..
0	0	0	1	..	-0.736	-1.335	1.103	1	..
1	0	1	0	..	1.574	1.951	0.550	2	..
1	1	0	1	..	0.153	0.513	1.090	2	..
1	1	1	0	..	0.804	-0.513	0.562	2	..
..	..	..	..	..	..	..	..	..	..

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 when outliers are present. For instance, consider the fact that some metric readings obtained from executing the software system under a given feature selection may be starkly different from the normal values due to a temporary usage spike.

To address this issue, *Observe* takes raw metric data through an automated normalization process prior to storing them as observation records. Normalized data captures the relative relationship between configurations with respect to a given metric, and thus reduces the effect of outliers. Many normalization techniques can be applied to transform the learning inputs into a representation that is less sensitive to such temporary fluctuations. In Table 1, observation records were normalized using *studentized residual* [25] 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 residuals 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 metric using the learned functions and actual observed value. For that purpose, we use the *learning error ratio* provided by the learning algorithm itself. Note that the majority of learning algorithms provide an error ratio 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 running the learning algorithm frequently is too costly for example. 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, *Induce* (recall Figure 2) constructs several functions that estimate the impact of making a feature selection on the corresponding metrics at a given execution context. *Induce* executes two steps.

TABLE 2. LEARNED METRIC FUNCTIONS. AN EMPTY CELL MEANS 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	..
F_1	1.553	1.137		2	..
F_2	-0.673	-0.938			..
F_3	0.709		-0.672		..
F_4		-0.174		1	..
F_5				4	...
F_6			0.244		...
F_7			0.591		...
F_1F_3	0.163				..
..	..	..	..	..	..

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 that learning needs to consider for each metric (also known as feature extraction). After the significance test, we apply the learning, which derives relationships between metrics and features using the normalized observations.

FUSION is not tied to a particular learning algorithm. One particular algorithm that we have extensively used in our implementation and evaluation is M5 model tree (MT) [26], which is a machine learning technique with three important properties: (1) ability to eliminate insignificant features automatically, (2) fast training and convergence, and (3) robustness to noise. Here, we describe the approach assuming the use of MT, but later revisit situations in which the metrics are discrete and such algorithm is not applicable.

Table 2 shows an example of how the induced functions look like. For the moment we have eliminated contextual factors (e.g., workload) from the table to simplify the demonstration of the approach, but revisit this later in the section. The empty cells correspond to insignificant features. The information in this table can be represented as a set of functions. For instance, a function estimating the impact of features on 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 (i.e., feature selection is "1010000") can be calculated as follows:

$$M_{G1}=\mathbf{1.553}\times 1-\mathbf{0.673}\times 0+\mathbf{0.709}\times 1+\mathbf{0.163}\times 1\times 1-\mathbf{0.843} = \mathbf{1.482} \quad (2)$$

When making adaptation decisions, values obtained from the induced functions (e.g., 1.482 from Eq. 2 above) would need to be 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 *Induce* could also learn 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 effect as follows. F_1 introduces a delay by adding a mediator connector, called *Log*, which 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-request 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. 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):

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

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. These equations do not necessarily need to be linear, and may be of the type (e.g., linear, sigmoid, exponential, etc.) that best captures the impact of context on the system.

Other methods of representing feature-metric relationships (i.e., induced functions) are needed in the case of discrete metrics. Here, classification-based algorithms [27] are more suitable, as they can efficiently represent such relationships in the form of *decision trees* [28]. For example, in TRS, the accountability metric (G_4 in Figure 1a) can take on five discrete values: *Very-Low=0, Low=1*, *Medium=2*, *High=3,* and *Very-High=4*. FUSION uses a classification-based learning algorithm, such as decision trees, which produce a set of rules in the form of implications. In order to be able to apply a utility function to such metrics, they must be converted to a branch function:

$$M_{G4} = \begin{cases} (F_5 = 1) \rightarrow 4 \\ (F_1 = 1) \rightarrow 2 \\ (F_4 = 1) \rightarrow 1 \\ else \rightarrow 0 \end{cases}$$

In the above example, the learning algorithm has inferred that selecting features *F5*, *F1*, and *F4* set the accountability metric to values *Very-High=4*, *Medium=2*, and *Low=1*, respectively, while other features have no significant impact on accountability. Given the step function representing the corresponding utility function (M_{G4} in Figure 1a), selection of *F5*, *F1*, or *F4* results in the same outcome, as they all achieve the maximum utility of 1. When we have a combination of continuous and discrete metrics, FUSION leverages sophisticated learning algorithms, such as CART [29] and MARSplines [30], which automatically combine the decision tree functions with regressed functions to arrive at regression tree functions that can be used to simultaneously reason about both continuous and discrete metrics.

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: *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). 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 runtime 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 runtime 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 adaptation, effecting solutions that provide a temporary fix is not a desirable approach. We would like FUSION to minimize recurrent adaptation of the system caused by 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 or a negative value greater than "– 1" to indicate a violated goal, and (2) when the metrics satisfy the minimum, returns a positive value less than "1" to indicate the user's preference for improvement. Therefore, utility in FUSION has dual purpose: not only is it used to initiate adaptation, but to also perform tradeoff analysis between 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 an optimization problem tailored to the running software:

TABLE 3. CONSTRAINTS FOR ENCODING THE FEATURE RELATIONSHIPS IN THE OPTIMIZATION PROBLEM GENERATED BY PLAN.

Feature Relation	Optimization Constraint	Variability Type
«zero-or-one-of-group»	$\sum_{\forall f_c \in zero-or-one-of-group} f_c \leq 1$	Optional
«exactly-one-of-group»	$\sum_{\forall f_c \in exactly-one-of-group} f_c = 1$	Mandatory
«at-least-one-of-group»	$\sum_{\forall f_c \in at-least-one-of-group} f_c \geq 1$	Mandatory
«zero-or-all-of-group»	$\sum_{\forall f_c \in zero-or-all-of-group} f_c \bmod N = 0$	Optional
Feature Dependency	$\forall f_{child} \in SharedFeatures, f_{parent} - f_{child} \geq 0$	Based on type of "*Child*" feature

- 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 features F_3, F_5, F_6, and F_7 since they have 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 functions. 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 problem in the running software system. The objective is to find a selection of *Shared Features*, F^*, that maximizes the system's overall utility for the *Conflicting Goals* given the set of all features F:

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

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:

$$\forall g \in ConflictingGoals,\ U_g(M_g(F)) > 0$$

Note that we do not need to include the goals that are unaffected by *Shared Features*. We then apply feature model constraints to the optimization problem. Table 3 demonstrates formulation of additional feature model constraints. For example, to prevent feature selections that violate the mutual exclusion, we specify the following constraint for each *exactly-one-of-group*:

$$\sum_{\forall f_c \in exactly-one-of-group} f_c = 1$$

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. Similarly, we ensure the dependency relationship as follows:

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

The above inequality does not hold if a 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 $U_{G1}(M_{G1}(F)) > 0$

$$U_{G2}(M_{G2}(F)) > 0$$

$$U_{G4}(M_{G4}(F)) > 0$$

$$F_3 + F_4 \leq 1$$

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

$$M_{G4} = \begin{cases} (F_5 = 1) \rightarrow 4 \\ (F_1 = 1) \rightarrow 2 \\ (F_4 = 1) \rightarrow 1 \\ else \rightarrow 0 \end{cases}$$

By eliminating F_3, F_5, F_6, and F_7, as well as U_{G3} from the optimization problem, we obtain a smaller problem that is tailored to the violated goals. The customized problem has less number of features and goals than the original problem. As will be shown in Section 8, in large software systems, by pruning the features and goals from the optimization problem, FUSION achieves significant performance gains with negligible degradation in the quality (accuracy) of adaptation decisions. By representing each feature with a binary decision variable, we solve the optimization problem using well-known *Integer Programming Solvers* [31], which provision the optimal solution. Stochastic algorithms, such as *greedy* and *genetic* [32], that rely on FUSION specific heuristics can also be used for providing near-optimal solutions very fast. However, stochastic algorithms are outside the scope of this paper.

Note that feature interaction terms (e.g., F_1F_3) make the optimization problem nonlinear. If such terms are present in the tailored problem, we use conventional techniques, such as replacing them with auxiliary variables (e.g., [33]), to obtain a linear version of the problem.

6.3 Effect

Once the *Plan* activity has found a new feature selection, it is passed to *Effect* for placing the system in the target configuration (recall Figure 2). *Effect* is responsible for choosing a path containing several adaptation steps (i.e., enable/disable features) towards the new feature selection. Depending on how features are mapped to the running software system, each step in turn may require making several changes to the running software. Since there are many possible paths to reach a target feature selection, *Effect* is responsible for picking a path that satisfies feature model constraints in addition to system goals. In our prior work [15], the managed system was transitioned to the target configuration without considering the implications on the goals during the adaptation process. This resulted in transient degradation of metrics and subsequently violating the goals until the system reached the target configuration. In this section, we present a novel algorithm, based on A* search algorithm [32], that uses the learned knowledge to find a path that altogether eliminates, and if not possible minimizes the extent of, goal violations during the adaptation process.

Figure 5 presents the *Effect* algorithm—a heuristics-based search algorithm that finds a suitable adaptation path. The function calls that are underlined are either simple functions with straightforward implementation and descriptive names (i.e., *createNode* and *backtrack*) or described in the following paragraphs (i.e., *expand*).

There are many possible paths to reach a target feature selection. Some of these paths may create inconsistent feature selections. For instance, in the TRS 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, dependent features should never be enabled without their prerequisites being enabled. The *Effect* algorithm uses *expand* to create candidates for each step of the path. Since *expand* adheres to feature model constraints, there are no invalid feature selection along the path calculated by *Effect*.

More formally, each step has a different type ($t \in T$) such as, *enable* or *disable* for an optional feature, or *swap of two features* for a mutually exclusive group. Recall from Section 6.2, set F corresponds to the adaptation units. In turn, we define the set of all possible adaptation steps as $S = T \otimes F$. Note that some adaptation steps may not be valid for a given feature selection. For instance, when an optional feature is already enabled it cannot be enabled again. We call a set of consecutive adaptation steps an *adaptation path*, denoted as π, which transitions the current feature selection towards another feature selection: $\pi = \sigma_1\sigma_2\sigma_3 \dots \sigma_{|\pi|} : (\sigma_i \in S)$.

As shown below, we can reduce the problem of finding such path to a graph search problem. To model the problem as a graph, we first define the set of nodes (V) and edges (E). We define each node $v \in V$ to be a feature selection. Similar to Table 1, we encode each feature selection as a binary string $b_1 b_2 \dots b_{number\ of\ features}$, where

```
//src: Source feature selection
//dst: Destination feature selection
//insig: Insignificant features
//featureModel: The feature model
List<Node> effect(src, dst, insig, featureModel) {
  visited = new Set<Node>
  queue = new PriorityQueue<Node>

  Node source = createNode(src, featureModel)
  Node destination = createNode(dst, featureModel)

  source.parent = null
  queue.push(source)

  while (! queue.isEmpty()) {
    Node extracted = queue.pop()

    if (extracted.equals(destination))
      return backtrack(extracted)

    //expand preserves the feature model constraints
    //it does not consider insignificant features
    List<Node> neighborhood =
        expand(extracted, featureModel, insig)

    for each (Node v in neighborhood) {
      if (visited.contains(v))
        continue

      if (! queue.contains(v)) {
        v.parent = extracted
        queue.push(v)
      } else {
        Node n = queue.get(v)

        if (g(extracted) + 1 < g(n)) {
          queue.remove(n)
          n.parent = extracted
          queue.push(n)
        }
      }
    }

    visited.push(extracted)
  }

  return failure
}
```

```
Node{
  String features
  Node parent
  boolean equals(that) {
    return this.features.equals(that.features)
  }
}
```

Figure 5. Effect heuristics-based path search algorithm.

each bit reflects the status of corresponding feature ($b_j = 1$ if F_j is enabled and $b_j = 0$ if F_j is disabled). We define edge $e \in E$ to be an adaptation step that transitions one feature selection to another: $e = v_{src} \xrightarrow{\sigma} v_{dst}$, where $\sigma \in S$. A simple path through this graph is defined as $v_{src} \xrightarrow{\sigma_1} v_1 \dots v_{k-1} \xrightarrow{\sigma_k} v_{dst}$, assuming none of the nodes are repeated (we are not interested in cycles or paths with cycles in them). If we call this path π, we have $\pi = \sigma_1\sigma_2 \dots \sigma_k$ and $|\pi| = k$. In other words, a path through this graph corresponds to an *adaptation path*, which transitions the system from the feature selection encoded in the source node v_{src} towards the feature selection encoded in the destination node v_{dst}.

To search through the graph, we have developed an algorithm with an *admissible* heuristic. A heuristic is admissible (also known as optimistic) if it is no more than the lowest-cost path to the goal [32]. As a result, although our algorithm is heuristic-based and does not search the

entire space, it is guaranteed to find a solution that is optimal. The search is based on the following values:

1. *Minimum distance* $h(v_i)$ represents the total number of features required in the best case to go from the current feature selection v_i to target v_{dst}. We can simply calculate this by counting the number of ones after applying bitwise XOR operator on the binary representations of those feature selections. For example, given v_i= 1110111 and a target configuration v_{dst}= 0010111, h(1110111)=2, since 1110111⊕ 0010111=1100000.
2. A*daptation cost* $g(\pi)$ represents the inconvenience due to adaption of a software system. One such inconvenience is the interruption to the operations of a running software system, e.g., delay due to the queuing of messages sent to the component being adapted. For the sake of simplicity, we define the cost of a path to be the number of its steps: $g(\pi) = |\pi|$. However, FUSION could be extended to use more complicated path cost functions.
3. *Utility loss* $Y(\pi)$ represents the extent to which system goals are violated during the adaptation path π. Based on the knowledge we obtained in the learning cycle (recall Table 2), we can calculate the utility loss as follows: $y(v_i) = |\sum_{\forall g \in Violated\ Goals} U_g(v_i)|$, where *Violated Goals* are goals that have a utility less than zero at vertex v_i. Intuitively, y takes the absolute value of the sum of all negative utilities at a given feature selection. We define the utility loss of a path to be the utility loss of the worst adaptation step in that path: $Y(\pi) = \max_{0 \le i \le |\pi|} y(v_i)$.

We let $f(v_i) = g(\pi_i) + h(v_i)$ represent the admissible heuristic for exploring the graph. Intuitively, $f(v_i)$ represents the overall cost of adaptation, where $g(\pi_i)$ represent the cost so far (i.e., up to vertex v_i), while $h(v_i)$ represents the expected minimum future cost (i.e., after vertex v_i). Alternatively, if the actual overhead of adaptation steps are known (e.g., through benchmarking of the system), one could use the actual adaptation costs in g and h. For the sake of simplicity, however, we assume all adaptations have the same weight, and as becomes clear later in this section, we use this to stir the algorithm away from adaptation paths that are very long.

The feature selections that are steps of the path (i.e., nodes) toward the target feature selection are inserted in a priority queue. The queue uses the following policy to priorities the vertexes in the queue: *Vertex* v_i *has a higher priority than vertex* v_j*, if* $Y(\pi_i) < Y(\pi_j)$*. However, if* $Y(\pi_i) = Y(\pi_j)$*, vertex* v_i *has a higher priority if* $f(v_i) < f(v_j)$*. Otherwise, in the case of a tie (i.e.,* $Y(\pi_i) = Y(\pi_j)$ *and* $f(v_i) = f(v_j)$*), one of the nodes is non-deterministically selected.*

The first item in this order is always the head of the priority queue. The above heuristic gives priority to finding a path that does not degrade the utility of the system, while aiming to reduce the amount of changes required to software system when there are no utility tradeoffs. Currently, and given the way $Y(\pi)$ is defined, FUSION aims to reduce the most severe utility degradation in a path. An alternative approach would be to reduce the accumulated goal violation in a path.

Effect algorithm creates the graph gradually (directed by the heuristic) as it progresses and does not assume a complete graph to be present. Given the *Conflicting Goals*, FUSION uses the knowledge base (recall Table 2) to eliminate the insignificant features. We can eliminate them, since they are not going to have a significant impact on the *Conflicting Goals*. This drastically reduces the neighborhood size of each vertex, which as will be shown in Section 8 improves the performance of the algorithm substantially.

Interested reader may find a detail example illustrating the steps taken by Effect algorithm in managing TRS in Appendix A.

7 Implementation Environment

Figure 6 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. Each part in Figure 6 is developed using a dedicated editor as described in the remainder of this section. As mentioned in Section 4.1, although other realizations are possible, a feature in our approach is realized as an extension of the software architecture at well-defined variation points. For example, *Adhoc Reports* feature in Figure 6b is realized using the architectural fragment *Adhoc Reports* in Figure 6c, which extends the TRS's base architecture. The TRS base architecture itself (depicted on the left side of Figure 6c) is the realization of *Travel Reservation System* core feature, which by default is present in all possible configurations of TRS.

FUSION's tool support is developed according to the three-layer reference architecture from [1]. These layers target the three kinds of concern in a self-adaptive software system: *goal management*, *change management*, and *component control* [1]. FUSION itself fits in the goal management layer (i.e., Figure 6a and Figure 6b). We have realized support for the other two layers by building on tools developed in the prior work. The change management layer is realized on top of XTEAM [34]—an extensible architectural description and analysis environment (a screen shot of which is depicted in Figure 6c). Finally, the component control layer is realized on top of Prism-MW [35]—a middleware environment aimed at architecture-based software development, with extensive support for runtime monitoring and runtime adaptation of the software.

Since self-adaptation concerns reside at different levels of abstraction, it is impractical to use a single modeling language to capture all self-adaptation concerns. To that end, our tool suite uses Online Model Driven Architecting (Online MDA) to relate concepts at different layers of abstraction. The overall architecture of tool-suite is depicted in Figure 7, where models and transformations are used at runtime to realize a hierarchical control loop. For example, enabling the *AdHoc Reports* feature in Figure 6b triggers a transformation rule that effects the corresponding *AdHoc Reports* architectural fragment in Figure 6c,

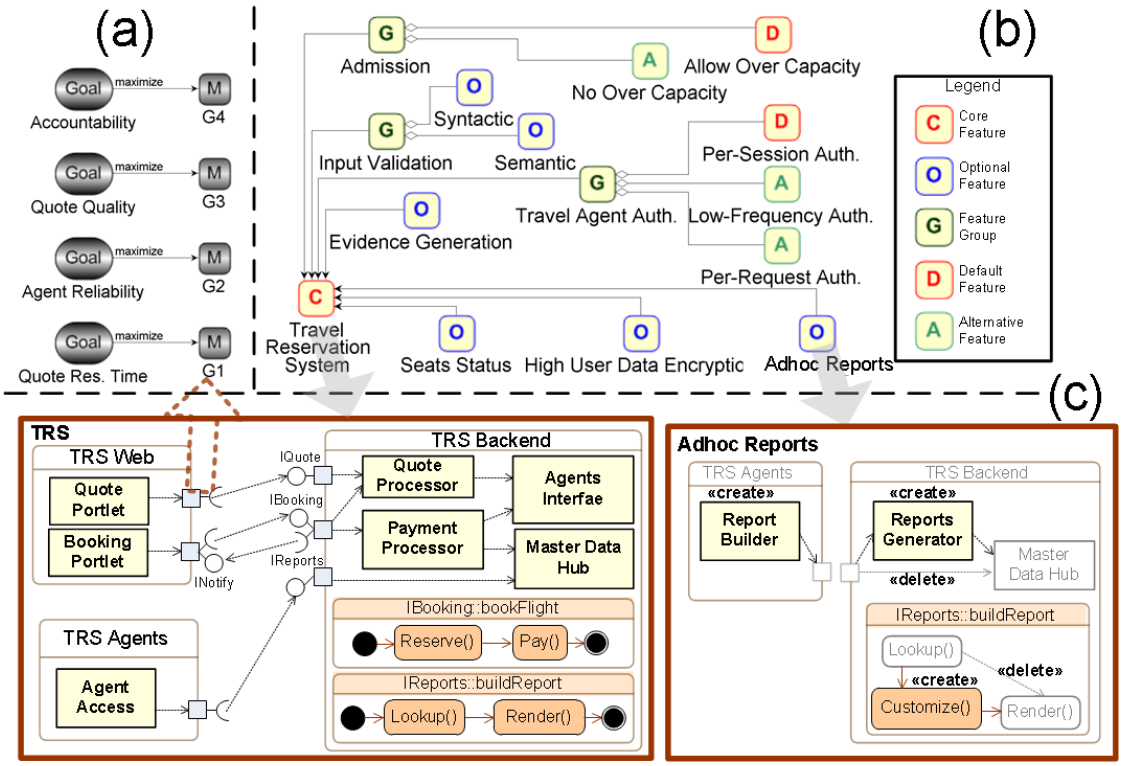

Figure 6. Subset of TRS in our prototype implementation of FUSION: (a) goals and metrics; (b) feature model; and (c) implementations of Core and Adhoc Reports features.

which then deploys the corresponding components and connectors into the running system.

The following subsections describe the tool-suite in more detail. We first provide an overview of the meta-models representing the underlying semantics of the models of the three layers of self-adaptation. Afterwards, we describe how the changes propagate among the models through transformation rules that are defined at the meta-model level (i.e., *online transformations* in Figure 7). Interested reader could find additional details about the tool support and the downloadable artifacts on FUSION's home page [44].

FUSION's tool-suite has been developed to the extent necessary to manage a few targets. We believe some effort would be required to extend our tool support for managing applications that are different from those used as case studies. For instance, the meta-models presented in the following subsections could be customized and extended with the additional properties that would need to be captured in different application domains.

7.1 Goal Management

As alluded to earlier, FUSION provides the goal management capabilities. Figure 8 depicts a portion of its meta-model. An example of the corresponding concrete syntax for the TRS system is depicted in Figure 6a and b. We have realized FUSION's modeling capabilities on top of the Generic Modeling Environment (GME) [36]—a configurable toolkit for creating domain-specific modeling and program synthesis environments. GME provides an extensible meta-meta-modeling language that could be used to define a meta-model describing the semantic and syntax of a domain-specific modeling language. GME also provides the ability to compile the meta-models constructed in this way into an instantiated environment, where models complying to the rules of the constructed language can be developed using an editor. Domain-specific models constructed in this way can be programmatically accessed and manipulated using GME's API.

The feature model editor developed in this way and depicted in Figure 6b allows the engineer to develop the feature models following the approach in [16]. FUSION's feature modeling language is comprised of *Feature* and *Feature Group* elements. From Figure 8, we can see that each *Feature* is associated with an attribute, called *selected*, which is used to capture the runtime state of that feature. If the feature is enabled, the *selected* attribute takes the value of "1"; otherwise, it takes the value "0".

A *Feature* can be of one of the following types: (1) *Core*:

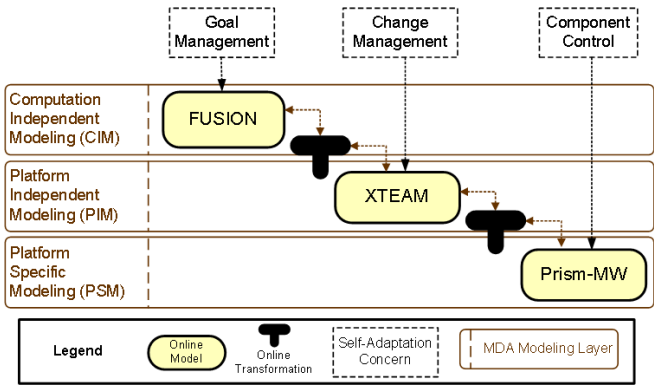

Figure 7. Self-Adaptation tool support using Online MDA.

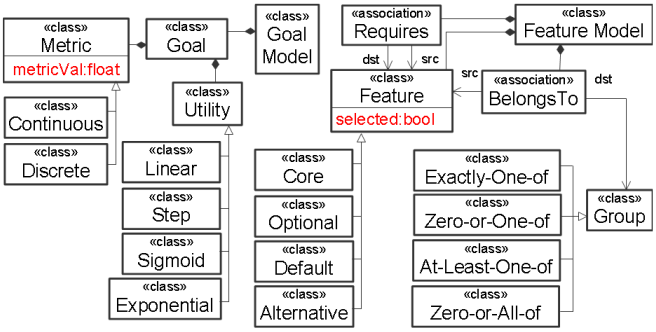

Figure 8. FUSION goal management meta-model.

must be enabled at all times; (2) *Optional*: may be enabled/disabled at runtime; (3) *Alternative*: can be enabled only when all features under mutual exclusive relationship are disabled; (4) *Default*: a special kind of *Alternative* feature that is enabled by default at system start up time.

Feature Group model elements are used to capture the following feature inter-relationships: (1) *Exactly-one-of-group*: captures a mutual exclusion relationship among all features associated with it, where one-and-only-one feature must be enabled; (2) *Zero-or-one-of-group*: also captures a mutual exclusion relationship except in this case the system can operate without any of the features enabled; (3) *At-least-one-of-group*: captures a mandatory variation at which multiple features may be enabled; (4) *Zero-or-all-of-group*: captures a mutual inclusion relationship, that is either all features are enabled or none.

In addition, a number of consistency rules apply to the feature model both at design time and runtime. Such consistency rules are captured using the Object Constraint Language (OCL). Some examples of design time consistency rules that apply to the feature model are: *core* feature cannot depend on an *optional* feature, *Exactly-one-of* groups must include one *Default* feature, *Optional* features shall not be part of an *Exactly-one-of* group, etc.

The goal model editor (depicted in Figure 6a) supports two types of model elements. The first type of modeling element is *Goal*, which can be of a number of subtypes (i.e., Linear, Step-Function, Exponential, and Sigmoid), corresponding to commonly used utility function templates. The second type of modeling element is *Metric*, which can be either *Continuous* or *Discrete*. While the FUSION metal-model depicted in Figure 8 is rich enough to represent the goals used in case studies in which FUSION has been applied, it may not be enough to capture goals that may arise in other application domains. In such a case, the FUSION meta-model would need to be extended using the GME's meta-model editor.

7.2 Change Management

We have realized the support for Change Management by extending XTEAM, an architectural modeling and analysis environment developed in [34] for representation of the system's software architecture. XTEAM supports modeling of a system's software architecture using heterogeneous Architectural Description Languages (ADLs) [18], where each ADL is suitable for representing a particular concern. For instance, XTEAM supports Finite State Processes (FSP) [37] and eXtensible Architectural Description Language (xADL) [38] for modeling the behavioral and structural properties of software system, respectively.

Notable portions of XTEAM's meta-models are depicted in Figure 9. The models are organized in *ArchitectureFolders*. Each *ArchitectureFolder* may contain one or more *Architectures*. The internal structure of the architecture is defined in terms of *Components* (i.e., independently deployable software units) that interact using *Connectors* (i.e., communication links). A snapshot of XTEAM's model for a subset of TRS is shown in Figure 6c. The *TRS's ArchitectureFolders* are: *TRS Web*, *TRS Agents*, and *TRS Backend*. The *TRS Backend* architecture has three interfaces: *IQuote*, *IBooking*, and *IReport*. Internally, TRS Backend is composed of four components, which are *Quote Processor*, *Payment Processor*, *Agents Interface*, and *Master Data Hub*. Components interact through connector links. Each *Component* implements several Finite State Processes. An *FSP* contains a number of *Activities* that process incoming messages from an interface of the component. *Activities* essentially represent the system's functionalities. For example, as shown in Figure 6c, the *TRS Backend* architecture contains two FSPs representing the *bookFlight* and *buildReport* processes, handling messages coming from the *IBooking* and *IReports* interfaces, respectively.

We have extended XTEAM to provide the means for change management. A *Fragment*, for instance, is an architectural snippet that can be weaved into the core architecture at well-defined variation points using change management operators (i.e., <<*create*>> and <<*delete*>>). The architectural fragment uses references to the model elements in the core architecture to specify the insertion points for the changes using the technique developed in MATA [22]. For example, Figure 6c shows the impact of weaving the *Adhoc Reports'* fragment on the core architectural model, which results in the addition of *Report Builder* and *Report Generator* components, some connector modifications, and the new *Customize()* step in the *buildReport* activity of *TRS Backend* architecture. The stereotypes <<*create*>> and <<*delete*>> in Figure 6c are realizations of the two *ChangeAction* types depicted in Figure 9.

We have also extended XTEAM to capture runtime state information. For instance, the *enacted* attribute of a

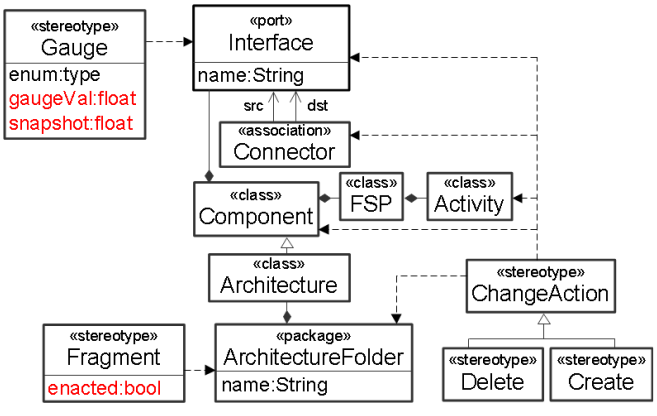

Figure 9. XTEAM change management meta-model.

Fragment (runtime state information marked red in Figure 9) is used to identify if the fragment is weaved into the base architecture. When weaved, it takes the value "1"; otherwise, it takes the value "0". Similarly, *Interfaces* that are associated with the *Gauge* stereotype capture runtime state information. Figure 6c highlights the *Quote Response Time* interface, which acts as a *Gauge* that collects round-trip time for request-response message exchange in the architecture. The gauge captures round-trip response time using the *gaugeVal* attribute depicted in Figure 9. This value comes from the underlying component control layer, as discussed next.

7.3 Component Control

We have provided support for component control layer using Prism-MW [35]. Prism-MW is an *architectural middleware* platform, meaning it provides one-to-one mapping between the architectural concepts and their programming-level counterparts. It provides programming-level constructs for realizing the architectural concepts, such as components, connectors, ports, architectural styles, etc. It also serves as a container for managing their lifecycle, with operations for creating, manipulating, and destroying instances of those objects.

These programming-level abstractions enable direct mapping between a system's software architectural model and its implementation. To implement a software component, the developer extends the *Component* class and provides the application logic by overwriting some of its abstract methods. Prism-MW also provides numerous off-the-shelf facilities, such as *Connectors*, *Ports*, and event handling and routing capabilities that the developer can use in the construction of a software system. To bootstrap a software system, the developer first instantiates an *Architecture* object, which acts a container for the system, and then uses its *add* and *weld* methods to create and connect software components, respectively. A partial meta-model of Prism-MW is depicted in Figure 10. For a more detailed description of Prism-MW and its facilities, we refer the interested reader to [35].

Prism-MW provides three key capabilities that make it a suitable runtime platform for our research. First, it provides support for architecture-based development, which makes it straightforward to map from XTEAM's architectural models to a software system executing on top of Prism-MW. Second, it provides architecture-level dynamism (e.g., adding/removing software components/connectors), as well as support for ensuring those adaptations do not jeopardize the system's functionality [39]. Third, it provides extensive instrumentation and probing capabilities to monitor the system's execution.

7.4 Runtime Integration Architecture

Each layer of self-adaptation has an online model based on the corresponding meta-model that we discussed in Sections 7.1, 7.2, and 7.3. As demonstrated in Figure 7, the interactions between these layers are through transformations defined between these online models. Adaptation is achieved through downward transformations, while monitoring is done through upward transformations. As a result, the platforms used for each layer of self-adaptation do not interface directly with each other.

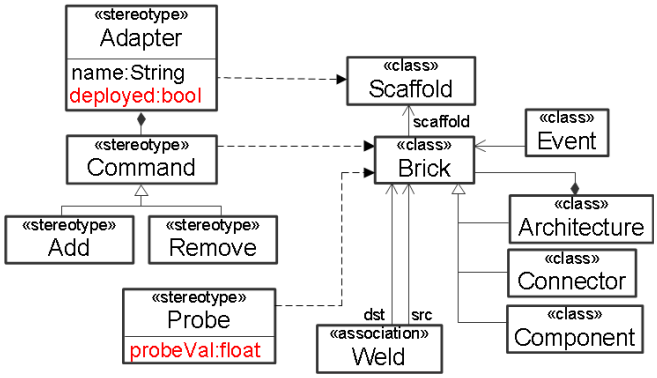

Figure 10. Prism-MW component control meta-model.

Each transformation automatically detects changes in its source model and incrementally propagates the changes to the target model. Whenever the tagged-value that corresponds to the state of a feature in FUSION model (i.e., *selected* attribute) is changed, a transformation rule is triggered. The change is propagated down to XTEAM by changing the state of the corresponding architectural fragment (i.e., *enacted* attribute). As a result, XTEAM weaves the new architectural elements that belong to the fragment into the core architecture. The change further cascades down to Prism-MW and results in modification of the running system. Monitoring works in the reverse order. When *probeVal* (recall Figure 10) changes, *snapshot*, and in turn, *gaugeVal* (recall Figure 9) are updated. Finally, as a result of change in *gaugeVal*, *metricVal* (recall Figure 8) is also changed to represent the most up-to-date measurements.

The connection between goal management and change management layers is realized through model-to-model transformations, while the connection between change management and component control layers is realized through model-to-code transformations.

Model-to-model transformations are defined at the meta-model level using QVT-Relations [40]. QVT-Relations language enables a source model to be transformed into a target model by applying a set of transformation rules. Each rule is expressed declaratively as a *relation* between two patterns (i.e., source and target patterns). A pattern match is accomplished, if and only if, all variables depicted in the pattern are bound to elements in the models. The *when* construct is used to express a precondition for the execution of the transformation rule. Transformation rules can be either one-way (forward or backward) or bi-directional. The efficiency of such incremental transformations and their feasibility for use at runtime has been shown in [41].

Figure 11 depicts *FUSION-to-XTEAM*, which is the transformation bridge that maps goal management concepts to change management concepts in the two meta-models. *FeatureToFragment* relation maps a Feature in FUSION to a Fragment in XTEAM. As a precondition, the *name* of the ArchitecureFolder associated with the fragment must match the name of the feature. For example, the *Adhoc Reports* feature in Figure 6b maps to the *Adhoc*

Reports fragment in Figure 6c, since they both have the same name. Note that the name of a fragment comes from the name of the *ArchitectureFolder* to which the stereotype is attached. Similarly, the *MetricToGauge* relation maps a Metric in the FUSION models (e.g., *M1: Quote Response Time* in Figure 6a) to a Gauge in the XTEAM models if the names match.

The *adapt* relation in Figure 11 maps the *selected* attribute in FUSION to *enacted* attribute in XTEAM. To make sure that the attributes on both sides correspond to each other, a precondition is defined to verify that the owning feature matches the owning fragment by chaining *adapt* relation to *FeatureToFragment* relation. Note that *adapt* is defined as an incremental transformation; it is triggered to execute dynamically whenever the *selected* attribute of a feature is modified without affecting other parts of the model. As a result, only the corresponding *enacted* attribute in the XTEAM model will be updated to have the same value.

Similarly, the *monitor* relation in Figure 11 maps the *guageVal* attribute of a Gauge in XTEAM to a *metricVal* attribute in FUSION, when the owning fragment and feature match. The key difference between *monitor* and *adapt* is that *monitor* operates in the reverse direction. The relation listens to modifications of *gaugeVal* in XTEAM so that it can propagate them to FUSION.

Table 4 shows some of the model-to-code transformation rules for connecting change management layer (i.e., XTEAM) to component control layer (i.e., Prism-MW). For instance, applying the <<*create*>> stereotype to *Report Builder* component in the *Adhoc Reports* fragment corresponds to generating a command in Prism-MW that adds the component to the *TRS Agent*. The transformation also keeps the model and the code in sync with respect to runtime state information. When Prism-MW updates the value of *probeVal* to capture the most recent readings, the transformation passes the value upward to the *snapshot* attribute in the XTEAM's online model.

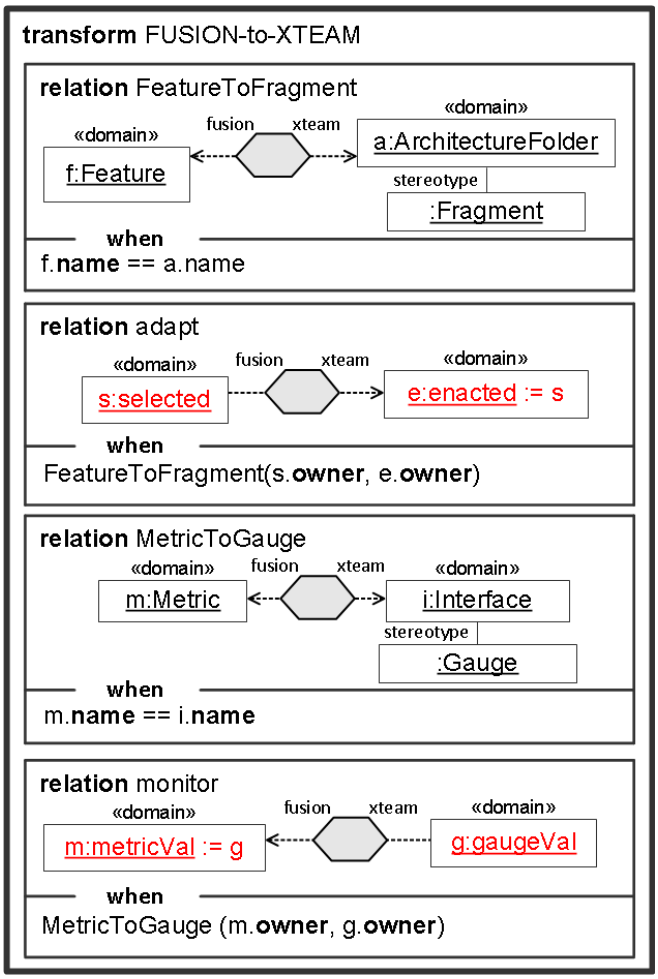

Figure 11. FUSION-to-XTEAM transformation definition.

TABLE 4. TRANSFORMING A SUBSET OF XTEAM'S CHANGE MANAGEMENT OPERATORS TO PRISM-MW CODE FRAGMENT.

Architecture Adaptation		Prism-MW Operation
Adaptation Operator	Type of Artifact	
<<create>>	Component	Architecture.add(Component c)
<<delete>>	Component	Architecture.remove(Component c)
<<create>>	Connector	Architecture.weld(Component a, Component b)
<<delete>>	Connector	Architecture.unweld(Component a, Component b)

The online models at each layer of self-adaptation are programmatically accessible. The activities depicted in Figure 2 (e.g., *Induce*, *Plan*, *Effect*) are realized as independently deployable programming modules that operate on the online FUSION model, as well as the knowledge base stored in a relational database. These modules also interact with a number of tools. For instance, *Induce* uses WEKA API [42], which provides an open-source implementation of learning algorithms. Moreover, the transformation from *snapshot* to *gaugeVal* in the XTEAM model is achieved by a programming module, which implements a running average algorithm.

8 EXPERIMENT SETUP

We conducted our experiments on an extended version of TRS, which consists of 78 features and 8 goals. Figure 12 depicts a portion of the system. TRS offers services in five major business process areas (i.e., *Flights*, *Hotels*, *Car Rental*, and *Account Management*). Each of the first three business process area (i.e., *Flights*, *Hotels*, and *Car Rentals*), involves four use cases that execute consecutively as follows:

1. *Price Quotes*: Collects trip details from the customer, discovers travel agent service providers to get a price quote, and returns a filtered list of quotes to the customer.
2. *Booking*: Allows the user to select one of the quotes, which is obtained from the previous use case, and then proceeds with selecting detailed preferences. In some cases, customer preferences, such as choosing aisle/window seat, may involve several rounds of interaction with the travel agent.
3. *Payment Processing*: Collects customer payment information and interacts with a credit card processing service provider. Successful transactions result in updating the user profile, which is maintained for each customer, to determine future discounts and travel packages.

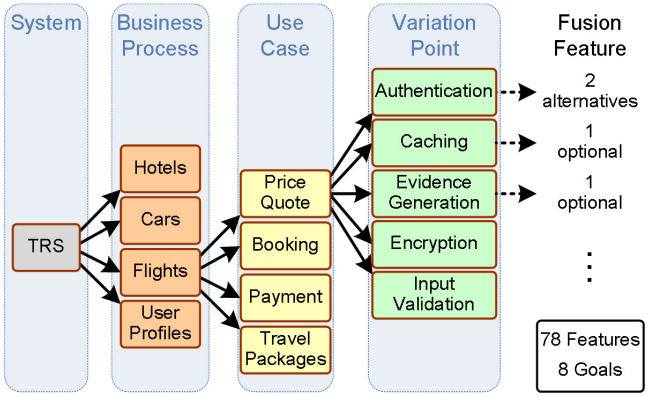

Figure 12. Structure of adaptation choices (i.e., Features) in TRS.

4. *Travel Packages*: Offers comprehensive travel packages (i.e., from other business process areas) to the customer based on trip destination and user profile history. Frequent customers qualify for additional discounts on travel packages.

For two of the business process areas (i.e., Flights and Hotels), we identified four QoS goals that are critical for the stakeholders of the system. Table 5 depicts the QoS goals for the *Flights* business process area along with their key characteristics, which are further described as follows:

1. *Quote Response Time*: aims to minimize the total roundtrip time for obtaining a filtered list of quotes for the customer (i.e., from initial request by the customer to delivery of list of quotes).
2. *Travel Agent Reliability*: aims to maximize compliance of travel agents to the price quotes issued by them. Compliance is measured by comparing Price at Quote time (Price@Quote) to Price at Booking (Price@Booking) time. For instance, travel agents that increase ticket at booking time force the customer to go back to search again for other quotes and, thus, result in customer dissatisfaction. The utility function of this goal is designed to welcome any decrease in the price at booking time and not vice versa.
3. *Quotes Quality*: aims to maximize maturity of reservations (i.e., price quotes that end with a purchase/payment). Thus, a key indicator of customer satisfaction of the *Price Quote* given to them is to continue the process through *Booking* and eventually to final *Payment Processing*. Customers that stop at the *Price Quote* use case and do not continue the process to the next step, which is *Booking*, are most likely dissatisfied about the list of quotes issued by the system. Therefore, TRS tracks the progress of customers in the business process and ranks quality of quotes accordingly (i.e., *Quote Produced*=0, *Booked*=1, *Payment Processed*=2).
4. *Accountability of Travel Agents*: aims to maximize the level of evidence generation by ensuring that the parties are collecting logs of transaction. Collecting evidence of transactions prevents any future repudiation by both parties in case of a dispute [43]. Therefore, transaction-level logs should be generated for each transaction that involves external parties, such as Travel Agents. Ideally, evidence generation involves establishing a trust chain within which pairwise evidence can be validated by a commonly trusted 3rd party. However, due to the overhead involved in such protocol, this is not always feasible. A widely acceptable compromise is to obtain pairwise evidence with both parties confirming receipt of log and acceptance of its content. Hence, absence of pairwise evidence renders the overall accountability of the system low.

TABLE 5. QOS DIMENSIONS CONSIDERED FOR THE FLIGHTS BUSINESS PROCESS.

QoS Goal	Metric	Utility
Quote Response Time	M = Roundtrip Latency (ms)	$U = -0.0755M + 1.13$
Travel Agent Reliability	M = Price@Quote/Price@Booking	$U = 5.17M - 4.17$
Quote Quality	$M = \begin{cases} 0, & Quote\ Produced \\ 1, & Booked \\ 2, & Payment\ Processed \end{cases}$	$U = \begin{cases} 0, & M = 0 \\ 0.5, & M = 1 \\ 1, & M = 2 \end{cases}$
Accountability	$M = \begin{cases} 0, & Local\ Log \\ 1, & Local\ Measures \\ 2, & Pairwise \\ 3, & 3rd\ Party \\ 4, & Common\ 3rd\ Party \end{cases}$	$U = \begin{cases} 0, & M = 0 \\ 0.2, & M = 1 \\ 0.5, & M = 2 \\ 0.9, & M = 3 \\ 1, & M = 4 \end{cases}$

Goals for the *Hotel* business process operate in a very similar manner and can be found in FUSION's home page [44].

For each goal, we analyzed each use case and identified practical adaptation choices (variations in the architectural configuration) that have a significant impact on the system's goals. To evaluate FUSION's ability to learn and adapt under a variety of conditions, we set up a controlled environment. We used a prototype of the implementation environment described in Section 7. We developed stubs in Prism-MW 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. In all of our experiments, FUSION was running on a dedicated Intel Quad-Core processor machine with 5GB of RAM.

We evaluated FUSION under four different execution scenarios, which we believe correspond to one of the four situations in which FUSION may find itself:

(NT) Similar context – the system is placed under workload settings that are comparable to those the system would face. 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 that is 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., impact of adaptation on

metrics) that can be learned. We use a scenario, called database *Indexing Failure (IF)*, in which the index of one database table used by the *Agent Discovery* component during the execution of the make quote workflow (recall Figure 1) unexpectedly fails, and forces full table scan for some parts of the database.

(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 totally randomized traffic, representative of an online Denial of Service attack. The traffic does not follow a typical skewed curve (i.e., exponential distribution).

9 Evaluation

In this section, we provide an empirical evaluation of FUSION's learning and adaptation cycles in terms of accuracy and efficiency. Unless otherwise stated, wherever 95% confidence interval has been reported, it has been established by executing the experiments 30 times.

9.1 Accuracy of Learning

Throughout this section, we use the term *observation* to indicate an adaptation decision made by FUSION and its effect on the system properties. Therefore, an observation 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 impact of feature selection on that metric. We refer to this as *Absolute Difference Percentage (ADP)*, defined as: $|(A_i - P_i)/A_i| * 100$, where i is an adaptation decision, P_i is the predicted value of the metric for that decision, and A_i is the actual value that is collected from the running system after the decision is effected. In the experiments reported here, learning is initiated if the average ADP in *10* most recent observations is more than *5%*. Other learning initiation policies could have been selected, each of which would present a tradeoff (i.e., overhead versus accuracy).

Figure 13 shows the error rate of observations for the *Quote Response Time* metric in the four scenarios described earlier. Each data point corresponds to an observation error at a particular point in time in the four evaluation scenarios. For this particular metric, measurements are collected based on the round trip time taken for a given request from the point of entering the system to the point of exiting. Then, a fixed control interval of 600 milliseconds is used to compute the average metric value which is eventually used to calculate ADP. The Y-axis represents ADP at a given observation.

We compared the results of FUSION against Queueing Network (QN) models of the system. QN is representative of conventional analytical models for reasoning about

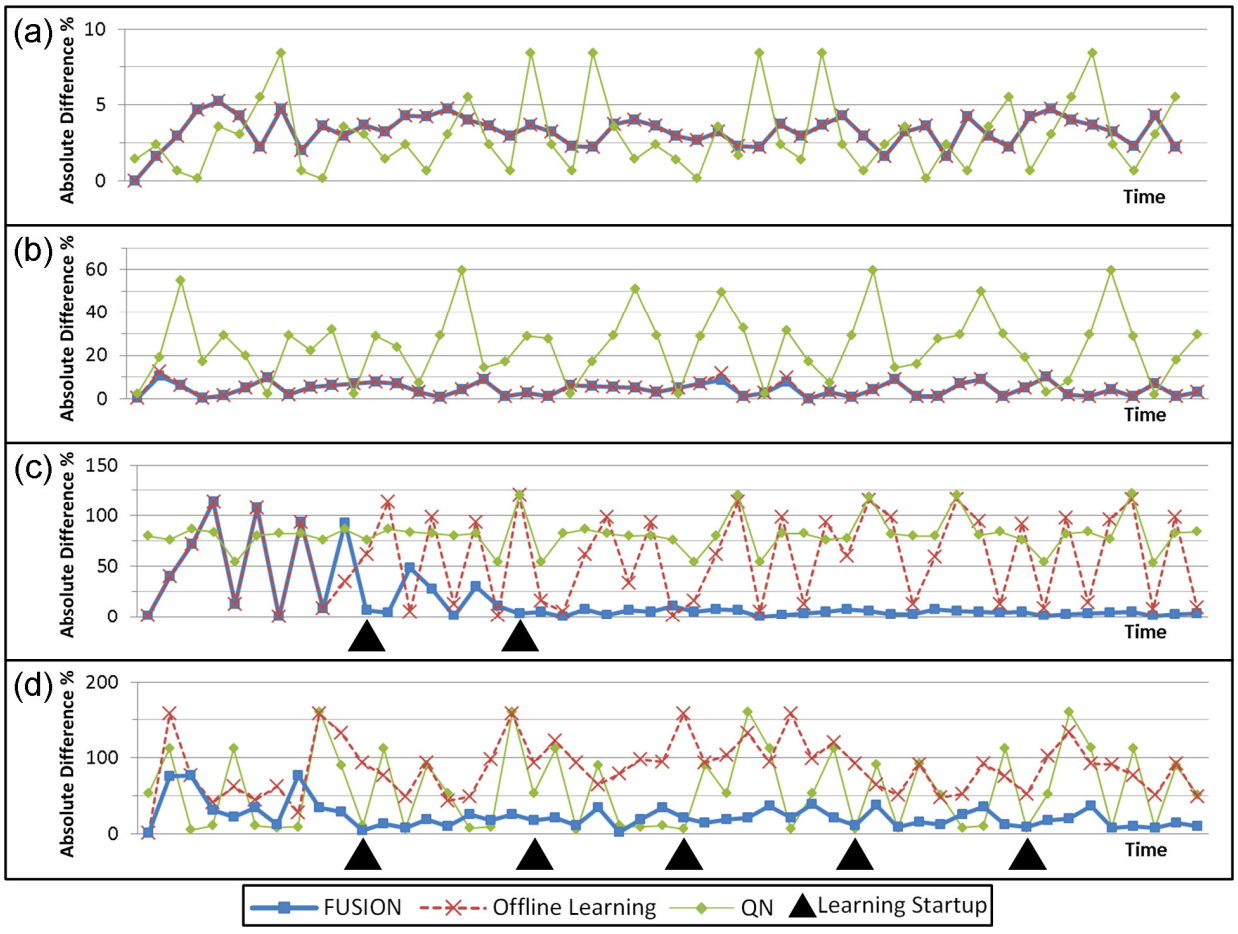

Figure 13. Accuracy of learned functions for "Quote Res. Time" metric (i.e., ADP): (a) Normal Traffic; (b) Varying Traffic; (c) Database Indexing Failure; and (d) Randomized DoS Traffic.

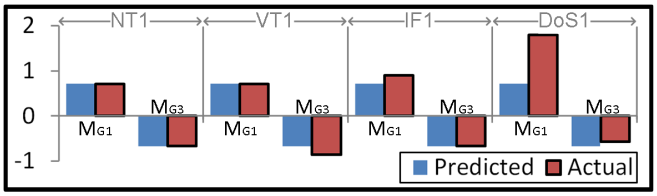

Figure 14. Impact of Feature Per-Request Authentication on Metrics Quote Response Time and Quote Quality.

performance of the system. 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 would 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 building self-adaptive systems by constructing tailored analytical models. In the accuracy comparisons reported below for TRS we constructed a subset of QN models that correspond to the feature selections made by FUSION.

Concept drifts further exacerbate the problem as they require generating a new QN model for a given feature selection, whenever a concept drift takes place. To address this issue, we employed online Queueing Network models in our research, which take parametric changes in the system into account (i.e., changes in workload). But even online QN models cannot handle concept drifts that are due to structural changes in the system, as opposed to parametric changes.

Figure 13a shows the TRS system under the NT scenario, where ADP for both FUSION and QN come within 5%, and often less. As expected, this indicates that both FUSION and QN achieve good level of accuracy under the expected conditions. QN's level of accuracy was within an average ADP of 2.9% and some spikes of 5-8%. This is due to the fact that some service demands in TRS are not fully compliant with the assumptions of the model.

Figure 13b shows the TRS system under the VT scenario. This shows that even when the workload changes frequently, FUSION's ADP remains within 5% on average. As a result, a new behavioral pattern sufficient for runtime learning never emerges. On the contrary, in the case of QN, operating outside of steady state condition combined with the wrong assumptions about some of the service demands exacerbate the prediction errors.

Figure 13c shows the TRS system under the IF scenario. It shows the fact that FUSION is capable of learning the new behavior, when concept drifts exist in the system. FUSION's ADP increases up to an average of 54% for the first 10 observations. This error is attributed to the fact that the model did not anticipate the impact of database access and associated software contentions, when the table scans were taking place. Software contentions were estimated to be responsible for 35% of FUSION's average ADP. Gradually, FUSION fine-tunes the coefficient of *Caching* and other features in the learned functions. As a result, average ADP goes down to less than 5%. In contrast, the average ADP of QN reaches 80%, since the QN model formulation presumes the existence of a functioning DB indexing system.

Note that it would be difficult to implement active monitoring of changes for service demands that are associated with database access. Active monitoring in such cases relies on probing a dedicated table with a functioning index reserved for measuring service demand time, which could give misleading readings if the DB indexing systems fails partially on a selective subset of tables. Consequently, online QN models may give wrong predictions due to false service demand readings.

Figure 13d shows the TRS system under the DoS scenario. The random nature of network traffic makes it impossible for FUSION to converge to an induced model that can consistently predict the behavior of the system within 5% average ADP. 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. Even though FUSION does not reach the same level of accuracy as in the other execution scenarios, it is still capable of masking transient effects and reducing errors significantly. This can be attributed to the fact that FUSION is benefiting from the continuous tuning, although it loses accuracy in the absence of a stable pattern.

9.2 Adaptation in Presence of Concept Drift

Clearly the quality of adaptation decisions depends on the accuracy of induced model. However, when the unforeseen behaviors emerge, the model is forced to make some adaptation decisions under inaccuracy, 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 further exacerbate the violated goals or not. Figure 14 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 13. Recall that the first observation for VT, IF, and DoS corresponds to a situation where 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 direction of 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 goal violations as it has learned the semantic underlying each feature. In other words, FUSION makes decisions that improve the system's properties, but not necessarily optimal, until the knowledge base is refined as will be demonstrated in the next section.

TABLE 6. SIZE OF SIGNIFICANT SPACE IN NUMBER OF FEATURES.

Experiment	NT1	NT2	NT3	VT1	VT2	VT3	IF1	IF2	IF3
FUSION	5	5	5	5	7	7	5	7	8
CS/TO	78								

9.3 Quality of Feature Selection

We evaluate the quality of solution (feature selection) found by FUSION against two competing techniques. 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. That is, it does not use feature reduction heuristics that result from significance testing. 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 15a plots the global utility obtained from running the optimization at the 3 different points in time for each of the NT, VT, and IF evaluation scenarios discussed earlier. We do not show the results for DoS case, because as mentioned in Section 9.1, it represents the case where learning is not possible due to the random behavior caused by the simulated denial of service attack. 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. The minor difference in quality is due to impact of features that are deemed to be insignificant. This demonstrates that our feature space pruning heuristics do not significantly impact the quality of found solutions. Table 6 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 15a also shows that FUSION finds solutions that are significantly better than CS. In turn, this corroborates our assertion in Section 6 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.

9.4 Efficiency of Optimization

In Section 9.3, we compared the quality of FUSION with two approaches: Traditional Optimization (TO) and Constraint Satisfaction (CS). In this section, we evaluate the performance of FUSION's planning algorithm. As you may recall from Section 6.2, FUSION achieves efficient planning 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 15b 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 15a. Note that the execution time of FUSION is comparable to CS and is significantly faster than TO. This in turn along with the results shown in the previous section demonstrate that FUSION is not only able to find solutions that are comparable in quality to those found by TO, but achieves this at the speed that is comparable to CS. Note that since TO runs exponentially in the number of features, for systems with slightly larger number of features, TO could take several hours for completion, which would make it inapplicable for use at runtime.

9.5 Protecting System Goals During Adaptation

FUSION leverages its knowledge base to find adaptation paths that minimize violation of system goals during adaptation. We compare FUSION against two alternatives.

The first technique, referred to as *Feature Constraints* (*FC*), uses a path search formulation that enforces feature model constraints during adaptation, but does not leverage the knowledge base to prune the search space. The

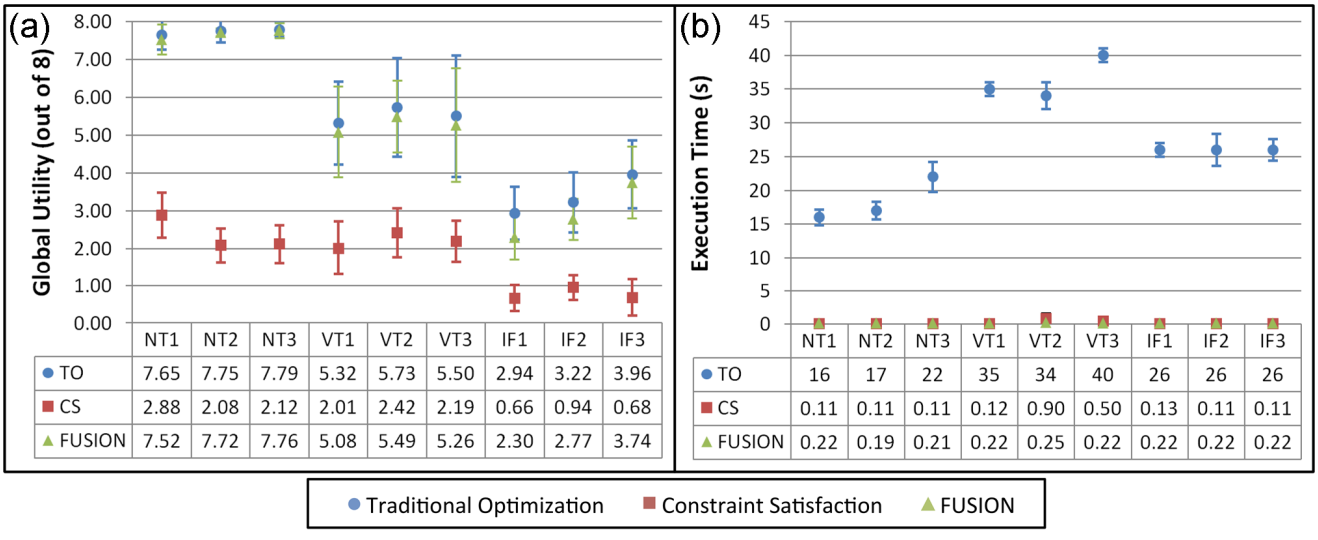

Figure 15. The result of optimization for different scenarios: (a) impact on global utility and (b) execution time.

second technique, referred to as *Knowledge and Feature Constraints* (*K+FC*), uses a path search formulation that is identical to *FC*, except it leverages the knowledge base *(K)* to prune the search space. As you may recall from Section 6.3, FUSION exhibits three characteristics: (1) It finds a path that satisfies feature constraints, (2) It leverages knowledge base to prune the search space, and (3) It picks a path that minimizes utility loss.

Figure 16a shows the utility loss from the paths obtained using the three alternative formulations. Each data point represents utility loss in the system at the designated point in time. In the NT scenario, all approaches are comparable in terms of utility loss, since most adaptation steps in the shortest path do not violate additional goals. However, FC and K+FC produce paths that worsen the utility loss and violate additional goals in scenarios VT and IF. FUSION produces paths that minimize utility loss in all 3 scenarios NT, VT, and IF. This demonstrates the FUSION's ability to leverage the inferred knowledge to produce adaptation paths that minimize the utility loss while enforcing feature model constraints.

9.6 Efficiency of Path Search

We evaluate the performance of FUSION's path search algorithm described in Section 6.3. As the reader may recall, FUSION achieves efficient planning by using the knowledge base to dynamically tailor a search problem that is relevant to the violated goals. FUSION also takes into consideration the objective of minimizing utility loss during adaptation, which is ignored in the FC and K+FC approaches.

Figure 16b shows the execution time of the path search for the same experiments reported in Figure 16a. Incorporating knowledge improves the efficiency of the path search process by as much as 10 times in NT and IF scenarios. In the VT scenario, the path search process becomes more extensive due to the number of features being changed.

Note that execution time of FUSION is not far behind K+FC despite the inclusion of utility loss function as a second objective. Utility loss function adds the burden of minimizing goal violation throughout the adaptation path. For instance, in *VT3*, the system is running at a peak workload in which no adjacent feature selection satisfies all goals. As a result, a long adaptation path comprised of additional neighboring feature selections had to be explored. Yet, FUSION execution time remains comparable to K+FC and clearly superior to FC. This in turn demonstrates that finding paths that minimize utility loss can be achieved efficiently.

TABLE 7. INDUCE EXECUTION TIME IN MILLISECONDS.

# of Observations	50	500	528	822	903	1227	1389
Linear Regression	20	30	30	50	60	70	80
M5Model Tree	60	110	130	130	130	160	230
SVM Regression	190	2310	3230	7330	8740	18700	29830

9.7 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 develop accurate models of system behavior. Table 7 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 the number of observations is large (e.g., initial training at design-time). In our experiments FUSION performed online learning using the M5 Model Trees algorithm on a maximum increment of 30 observations, which from Table 7 could be verified to have presented an insignificant overhead of less than 60 ms. This efficiency is due to the pruning of the feature space and significance test described in Section 5.

10 THREATS TO VALIDITY

In this section, we discuss the threats to the validity of our experimental results by studying factors that could impact the performance and accuracy of FUSION. This discussion in turn helps us frame the types of systems that could benefit from FUSION, and explicitly describe the assumptions and limitations of the approach.

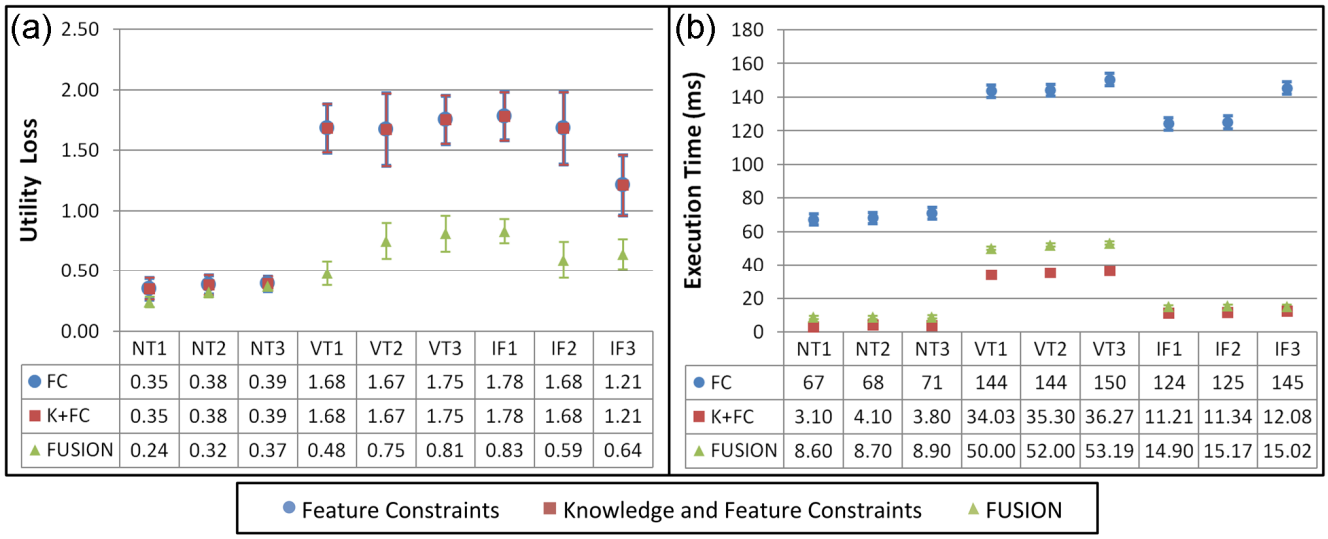

Figure 16. The result of path search for different scenarios: (a) impact on utility loss and (b) execution time.

10.1 Impact of Feature-Metric Coupling

As you may recall from Figure 2, the behavior of the managed software system is modeled in the knowledge base. As we elaborated in earlier sections, these models are mainly in terms of feature-metric relationships. Consequently, in FUSION, the level of coupling between features and metrics is a key indicator of the level of complexity in the behavior of the system.

Figure 17 depicts the lowest (Figure 17a) and highest (Figure 17b) levels of coupling in a hypothetical system. In the simplest case, the value of each metric is determined by only one feature. On the other hand, in the most complex case, the value of each metric is determined by considering all features in the system. Feature-metric coupling, which represents the extent to which metrics are affected by features, impacts several parts of FUSION, as discussed in the remainder of this section.

This coupling directly impacts the *Induce* activity (recall Section 5.2). When the number of features impacting the value of a given metric increases, the complexity of the corresponding function predicting the impact of features on metrics increases. Therefore, learning algorithms will need more observations and more time to converge (i.e., reduce the error to an acceptable threshold). In such cases, FUSION may need to adopt more complex (e.g., nonlinear) models for learning the functions.

Note that increasing the number of metrics does not increase the complexity of the learning exponentially. This is due to the fact that inducing the function corresponding to each metric is an independent task.

Coupling also affects *Plan* activity. As you may recall from Section 6.2, the first step in fixing a violated goal is to build the set of features that affect it. This set is called *Shared Features* and determines the size of the optimization problem for resolving that goal violation. When coupling increases, the difference between *Shared Features* and set of all features converges to an empty set. This means that for any goal violation the entire system would need to be optimized, which increases the execution time of the planning algorithm.

The second step for fixing a goal violation is to find other goals that are also affected by the same features. As you may recall from Section 6.2, the set of these goals is called *Conflicting Goals*. When coupling increases, the difference between *Conflicting Goals* and set of all goals converges to an empty set. In other words, for fixing any goal violation, trade-offs between all the goals should be considered. In addition to forcing optimization of the entire system in the extreme cases, this also decreases the number of viable solutions for fixing a goal violation.

Since *Effect* (recall Section 6.3) is invoked after *Plan* activity, it is impacted by the coupling in the same two ways. First, increase in the size of *Shared Features* decreases the size of *insig* parameter passed to the Effect algorithm (recall Figure 5). As a result of this, the number of viable options at each step of the search increases. Second, the increase in the size of *Conflicting Goals* will make it more likely to have *Utility loss* (i.e., $Y(\pi) > 0$). This means that the chance of getting a longer path for solution increases. In other words, the amount of disruption for executing the plan as well as the chance of temporary goal violation during the transition increases.

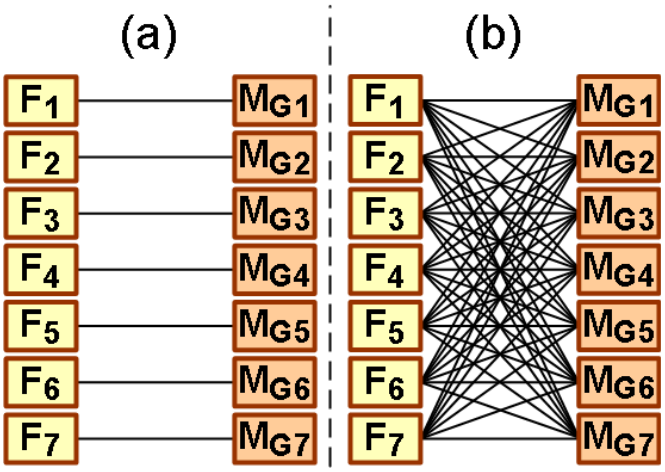

Figure 17. The lowest (part a) and highest (part b) levels of feature-metric coupling in a hypothetical system with 7 features and 7 metrics.

Extreme cases, such as Figure 17b, also challenge other existing approaches. Manually deriving quantitative models in such settings is exacerbated when there is a high-degree of coupling between adaptation choices and affected metrics. Our experiences (e.g., [4], [33], [39], [45–48]) with the construction of numerous adaptive software systems over the past decade, in the context of this project as well as others, indicate that the coupling in most systems is not as extreme as that depicted in Figure 17b, but rather somewhere in between that and Figure 17a.

By doing the significance test before running the learning algorithm (recall Section 5.2), FUSION already reduces feature-metric coupling. FUSION could be extended to use more complex learning algorithms that produce more complex functions, although this has not been necessary in the systems that we have come across so far. Finally, in extreme cases, using heuristic-based algorithms (e.g., greedy or genetic algorithms [32]), which provide near-optimal solutions very fast, could help with improving the performance at the expense of accuracy.

10.2 Impact of Feature Model's Structure

Feature model is the key enabler of learning in FUSION, as feature inter-relationships (e.g., dependency, mutual exclusion) could be used to represent the engineer's knowledge of the valid adaptation choices.

To illustrate the impact of feature model's structure on FUSION, consider the two hypothetical feature models in Figure 18. One of the features is the core feature, which is always enabled in the system. The remaining 10 features are optional that can be either enabled or disabled. The two feature models are different in terms of dependencies between the features. All optional features in Figure 18a are dependent on the core feature and can be enabled/disabled independent of each other. Therefore, feature model constraints do not reduce the feature space, and hence, the total number of features combinations for feature model of Figure 18a is 2^{10}=1024. On the other hand, optional features in Figure 18b form a chain of de-

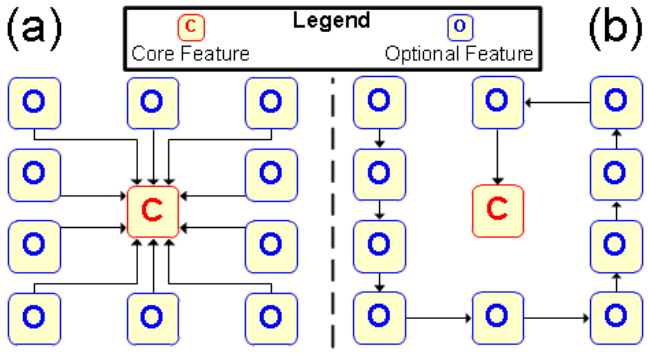

Figure 18. Two Feature Models with the same number of features but different relationships among them.

pendencies and cannot be enabled/disabled independent of each other. As a result, the feature space is reduced to 11 total feature combinations, which is almost a ten-fold reduction.

The size of feature space influences same aspects of FUSION that was discussed in Section 10.1 (i.e., *Induce*, *Plan*, and *Effect*). Smaller number of valid feature combinations implies a smaller learning space. As a result, for learning the behavior of the system fewer observations would be required. Moreover, it is more likely for the learning algorithms to converge in a short amount of time. In the case of *Effect*, smaller number of valid feature combinations implies smaller number of nodes in the search tree. Therefore, the total memory and time, which is required for finding an adaptation path, is reduced.

The reduction in the size of the feature space in *Plan* is achieved through adding more constraints in the optimization problem (recall Section 6.2). At first blush, this seems to imply smaller solution space and faster optimization. However, in some cases, additional constraints may transform the shape of the solution space into an irregular convex. As a result, finding the optimal solution becomes more complex, and in turn, will need more time [49]. Therefore, making a general statement about the influence of the size of the feature space on *Plan* is not possible.

The two feature models in Figure 18 are intended to show the two extremes, and are unlikely to occur in practice. Feature models usually have many interdependencies between the features, which reduce the number of valid feature combinations, although not to the extent of forming a chain. As a result, a typical feature model fits somewhere between these two extremes.

10.3 Impact of Feature Interactions

The effects of features on a given metric are not always independent of each other. As you may recall from Section 5.2, features may have a combined effect on a metric, also known as feature interaction.

During learning, FUSION models feature interactions that pass the significance test as separate variables (recall Table 2). This in turn means that more observations are necessary to accurately learn the impact of features on metrics. Hence, *Induce* will need more time to converge.

As we mentioned in Section 6.2, feature interactions make the optimization problem in *Plan* nonlinear. Therefore, FUSION needs to transform them into linear problems by introducing auxiliary variables and additional constraints. Larger solution space (more variables) and more constraints increase the complexity of the optimization problem. In other words, feature interactions increase the execution time of the planning algorithm.

Our observations indicated that usually limited number of feature interactions per metric pass the significance test and become significant variable. Moreover, the interaction of more than two features rarely has significant effect on metrics. Therefore, although feature interactions affect complexity in FUSION, in practice the increase in the complexity is usually very limited. Finally, note that feature interactions do not impact *Effect*.

10.4 Assumptions and Limitations

There are several assumptions underlying FUSION that delineate the scope of its applicability, which we describe explicitly in this section:

- FUSION relies on precise and accurate collection of metrics from the managed software system. Otherwise, the behavior of the system cannot be observed, and in turn learned in terms of feature-metric couplings. We are assuming that the system is running in an environment in which metrics of interest can be observed and collected. Such basic facilities are usually provided at the system level and/or by the modern middleware platforms, one of which was described in our implementation prototype (recall Section 7.3).
- FUSION assumes a mapping from the features to software elements that realize the features is established, and that changes in the software can be exercised at runtime. Recall from Figure 3 that FUSION is independent of how this mapping is realized, as long as the change management and component control layers provide the means for effecting those changes in a consistent fashion. In our implementation prototype, we have developed one approach to realizing this mapping, and used existing tools for exercising those changes at runtime. However, the problem of mapping features to the software artifacts is one that deserves additional attention to increase the scope of systems where FUSION can be employed. The research in dynamic software product lines (e.g., [19], [20]) has made significant progress through tools and techniques for establishing the mappings between feature variability and implementation-level artifacts.
- Another assumption in FUSION is that sufficient data is available about the system's execution under different feature configuration to allow for machine learning techniques to infer an accurate model of the system's behavior. For instance, in the case of TRS case study, we first benchmarked the software system to collect this data, before deploying it. The runtime learning is mainly intended to fine-tune the knowledge base as opposed to construct it from scratch. We believe this is a prac-

tical assumption, as most systems can be benchmarked prior to deployment and changes at runtime are not going to be so drastic that the entire knowledge base becomes useless. If that is not the case, then other techniques would need to be employed for the management of software.

- During a concept drift (i.e., at a point where the knowledge base is not accurate with respect to unanticipated changes that have occurred in the system) FUSION makes adaptation decisions based on stale knowledge. As demonstrated in Section 9.2, decisions made using stale knowledge typically do not exacerbate violation of goals, as a given feature maintains the same general direction of impact on metrics. Here, we assume the system could be managed sub-optimally, until enough data is collected from the modified system to allow for refinement of knowledge base. In systems where this is not acceptable, such as high-risk and mission critical systems, FUSION as well as other prior techniques would be inapplicable.
- Finally, FUSION assumes prior to system's deployment engineers are able to identify useful features that could resolve the issues that may arise at runtime. This means that FUSION's scope of management is limited to issues that can be addressed with a set of preconceived features. FUSION may not be able to resolve an issue that could be resolved through runtime adaptation, simply because the engineers have not included the appropriate features.

11 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 [1], [7], [50], [51]. We provide an overview of the most notable research in this area and examine them in the light of FUSION.

11.1 Modeling Software Adaptation Space

Ryutov et al. [52] provide a security framework that supports adaptive access control and trust negotiation through parameterization. Similarly, Bennani et al. [53] propose a parameterized model (i.e., an online analytical model) for estimating the system's performance by incorporating system characteristics (e.g., workload) that become known at runtime. While parameterization has the advantage of simplicity, it lacks support for large-scale adaptation at the system level.

Component-based adaptation [1], [5], [6], [8], is at the architectural level, often in terms of structural changes, such as adding, removing, and replacing software components, changing the system's *architectural style*, rebinding a component's interfaces, and so on. This enables large-scale adaptation at the system-wide level by swapping distributed components at runtime. Since these works serve as the foundation for our work, we further discuss this paradigm and how it relates to architecture-based reasoning in Section 11.3.

Aspect Oriented Modeling is used to build adaptation paradigms for addressing crosscutting concerns that are related to non-functional requirements. In past work [22], we presented MATA, a UML aspect-oriented modeling technique, that uses graph transformations to change a system's software architecture at runtime. Aspects reduce the configuration space significantly and make runtime analysis feasible. Aspects are causally connected; however, aspect-oriented approaches do not support specifying user-defined inter-aspect relationships and interactions.

Feature-orientation has been used as a method of modeling the requirements of a dynamic product line [19]. Lee et al. [20] break the existing feature model of a system into several subsets, which correspond to the major functionalities of the system. They call these subsets feature binding units. By enabling and disabling these units, they reconfigure the system at runtime. Cetina et al. [21] show how adaptations can be made possible by reusing the existing feature models at runtime. Although these works have adopted a similar model of adaptation as FUSION, none has explored the opportunities it presents for online learning and decision making.

DiVA project [54] has resulted in a framework for building adaptive systems. Two branches from this project are relevant to FUSION. The first branch, which is based on aspect oriented modeling, uses aspects as course-granular units of adaptation [55] to tame the combinatorial explosion in the configuration space. The second branch, which is based on MDA, uses feature oriented representations to model variability in the system and its context [56]. The feature model is also used to model relationships in the domain. These two branches are merged to bring the best of aspect oriented modeling and MDA together [57]. FUSION adopts a similar modeling methodology. However, in addition to this, features are units of runtime learning and reasoning in FUSION.

11.2 Approaches for Analytical Modeling

Analytical models differ in the way knowledge about the behavior of the system is represented. From this perspective, analytical models fall under two broad categories: white-box and black-box. The former requires an explicit model of the internal structure of the software system (i.e., typically an architectural model), while the latter does not require such knowledge.

Queuing Network (QN) [10] is a mathematical model used for performance analysis of a software system, represented as a collection of *Queues* (i.e., system resources) and *Customers* (i.e., user requests). Markov model [11] is often used for reliability analysis. It is comprised of a stochastic model that captures the state of the system using random variables that change overtime according to the probability distribution of the previous state. These white-box approaches require an explicit model of the internal structure of the software system. Such models are typically used at design time to analyze the tradeoffs of different architectural decisions before implementation, but recently these models are used at runtime to dynami-

cally analyze the system properties [58]. However, the structure of these models cannot be easily changed at runtime in ways that were not accounted for during their formulation (e.g., addition of new queues in a QN due to emerging software contentions).

Artificial Neural Network (ANN) is an effective way of solving a large number of nonlinear estimation problems. Model tree is based on regression trees and associate leaves with multiple regression models (i.e., M5 Model Trees [59] and Multivariate Adaptive Regression Splines (MARS) [30]). As a black-box analytical modeling family, these approaches do not require knowledge of the internal structure of the system. But, they require sufficient sampling of the input/output parameters to construct an approximation of the relationship between the inputs and outputs. A key advantage of black-box approaches is that they can be used to detect concept drifts (i.e., changes to the underling properties of a software system over time). FUSION follows the black-box approach.

11.3 Architecture-Based Reasoning

Kramer and Magee [1] state that software architectures provide an appropriate level of abstraction for modeling and reasoning about dynamic adaptation. They define a three-layer model (component control, change management, and goal management) to address the challenges associated with the development of self-adaptive systems.

Oreizy et al. pioneered the architecture-based approach to runtime adaptation and evolution management in their seminal work [6], [60]. Runtime adaptation is facilitated using the C2 architectural style that uses connectors to route messages among components through implicit invocation, thus minimizing interdependencies.

Garlan et al. presented the Rainbow framework [2], a style-based approach for developing reusable self-adaptive systems. In Rainbow, an architectural model is used to monitor and detect the need for adaptation in a system. The self-adaptation language describes rule-like constructs (condition-action). When the condition is met, the appropriate action is executed to adapt the system.

Malek et al. [33] provide a generic framework for improving the QoS of a distributed software system by changing its deployment architecture at runtime. Deployment architecture denotes the allocation of software components to the hardware nodes. Domain experts express QoS properties using a generic architecture-based representation of the system.

Inverardi et al. [61] tackle the variability associated with context and evolution of requirements in context-aware adaptive systems. To that end, they use features for consistent evolution of component-based service-oriented systems, where a feature is a dynamic unit representing the smallest part of a service that the user can perceive.

All of the above approaches, including many others (e.g., see [7]), share three traits: (1) use white-box analytical models for making adaptation decisions, and (2) rely on architectural representation for the analysis, and (3) effect a new solution through architecture-based adaptation. As manifested by the key role of architecture in FUSION, the above approaches form the basis of our work. However, similar to Inverardi et al., FUSION adopts a feature-oriented black-box approach to reasoning and adaptation, which not only makes the runtime analysis efficient, but also reduces the effort required in applying FUSION to existing systems. Moreover, unlike these approaches, FUSION is capable of coping with unanticipated changes through online learning.

11.4 Machine Learning Approaches

Gambi et al. [62] proposed the use of online machine learning using surrogate models to limit violations of SLA of software applications within Virtualized Data Centers (VDCs). However, their approach does not apply any state space pruning heuristics to reduce the learning space and improve runtime convergence.

Tesauro et al. [63] have proposed a hybrid approach that combines white-box analytical modeling (i.e., QN models) with Reinforcement Learning. Online learning uses a simplified black-box representation of the running system, while the white-box QN model is used as a training facility. A key assumption of the work is that white-box QN model of the system is available and can accurately predict its behavior under different adaptation decisions. The approach is also focused on the problem of managing redeployment of applications in Data Centers.

Kim and Park [64] propose a reinforcement learning 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.

Zhao et al. [65] use a hybrid Supervised Reinforcement Learning approach that combines the merits of Supervised Learning and Reinforcement Learning to develop an adaptive real-time cruise control system.

The work by Rieck et al. [66] and Sabhnani et al. [67] demonstrate the use of several machine-learning algorithms on cyber-attacks. The algorithms were used to detect the drift of a system from its normal patterns of behavior, which is typically a sign of misuse of the system.

FUSION's objective is to provide a general-purpose approach for self-adaption of the software systems, which is fundamentally different from the above works that are concerned with a specific problem. Due to the malleability of software applications, adaption space can be enormously large. Thus, FUSION combines a number of techniques (i.e., feature-based knowledge, significance testing, heuristic search, etc.) to make online reasoning feasible.

12 Conclusion

We presented a *black-box approach* for engineering self-adaptive systems that brings about two innovations for solving the aforementioned challenges: (1) a new method of modeling and representing a self-adaptive software systems that builds on the notions of *feature-orientation* from the product line literature, and (2) a new method of assessing and reasoning about adaptation decisions through online learning. FUSION uses features and inter-feature relationships to significantly reduce the configura-

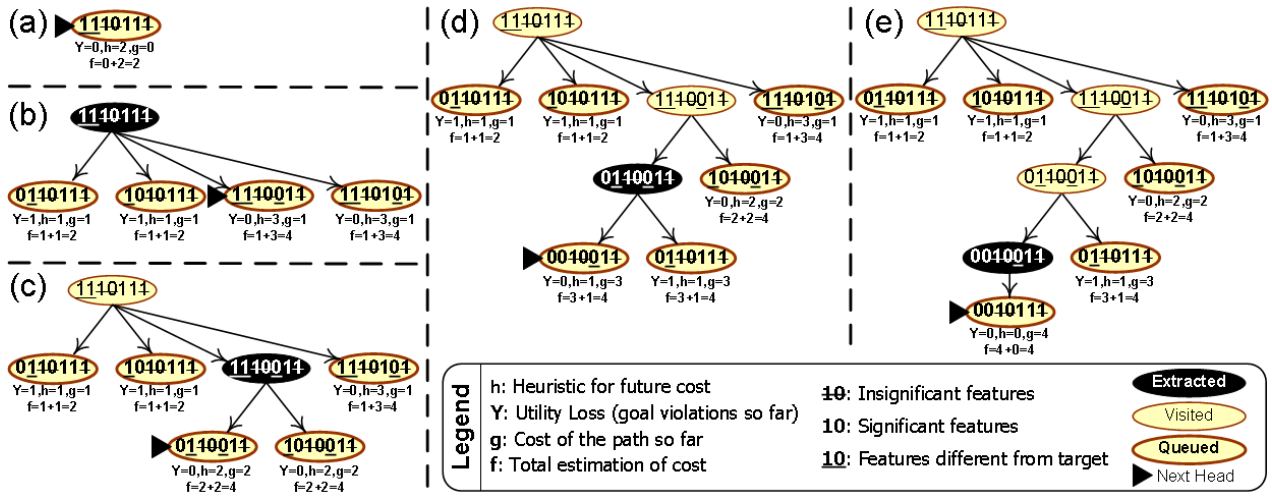

Figure 19: Effect finds the best path within the significant feature space.

tion space of a sizable system, making runtime analysis and learning feasible.

We also presented an empirical evaluation of the approach using a real-world self-adaptive software system to demonstrate the feasibility of FUSION and the quality and efficiency of learning and adaptation decisions.

Currently we are working on two extensions to FUSION: (1) in some real world systems, waiting until a goal violation occurs and then reacting to it may be very costly, thus we are investigating techniques for proactive adaptation on top of Fusion; and (2) we are extending the *Effect* algorithm to use additional information (e.g., actual cost of each adaptation step) in the path search process. In addition, currently FUSION assumes a single stakeholder. An interesting avenue of future research would be to extend the framework to model multiple users' preferences in terms of multi-dimensional utility functions.

APPENDIX A, EFFECTING CHANGE EXAMPLE

This appendix demonstrates the application of *Effect's* heuristic-based path search algorithm (recall Figure 5) via an example. It shows a situation in which the initial feature selection of the TRS system before adaptation is "1110111", meaning that all of the features expect F_4 (i.e., *Per-Session Auth.*) are enabled, and the target feature selection is *0010111*.

Figure 19a shows the start of the algorithm at the current feature selection, which is added to a priority queue. In each iteration, a vertex, indicated by black oval is extracted from the head of the queue and its neighboring vertexes (i.e., reachable with one valid adaptation step and calculated by the call to *expand* in Figure 5) are added to the queue. If any of the recently added vertexes to the queue is the target, the algorithm stops as the target is reached. The extracted vertex is then added to the visited list. We also keep a record of the visited vertexes to be able to backtrack the path.

In Figure 19, assuming we are concerned about G_2 and G_3 (i.e., *Agent Reliability* and *Quote Quality*), we can prune F_3, F_4 and F_7 (i.e., *Per-Request Auth.*, *Per-Session Authentication*, and *Semantic*), as they do not impact those goals. Insignificant features in Figure 19 are indicated using a strikethrough line.

ACKNOWLEDGEMENT

This work is supported in part by awards W911NF-09-1-0273 from the Army Research Office, D11AP00282 from the Defense Advanced Research Projects Agency, and CCF-1252644 and CCF-1217503 from the National Science Foundation. We would like to thank the anonymous reviewers for their constructive feedbacks which led to the improvement of the paper.

REFERENCES

[1] J. Kramer and J. Magee, "Self-Managed Systems: an Architectural Challenge," in *Int'l Conf. on Software Engineering*, Minneapolis, Minnesota, 2007, pp. 259–268.

[2] D. Garlan, S. W. Cheng, A. C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure," *IEEE Computer*, vol. 37, no. 10, pp. 46–54, Oct. 2004.

[3] J. O. Kephart and D. M. Chess, "The Vision of Autonomic Computing," *IEEE Computer*, vol. 36, no. 1, pp. 41–50, Jan. 2003.

[4] S. Malek, G. Edwards, Y. Brun, H. Tajalli, J. Garcia, I. Krka, N. Medvidovic, M. Mikic-Rakic, and G. S. Sukhatme, "An architecture-driven software mobility framework," *Journal of Systems and Software*, vol. 83, no. 6, pp. 972–989, Jun. 2010.

[5] D. A. Menascé, J. M. Ewing, H. Gomaa, S. Malek, and J. P. Sousa, "A framework for utility-based service oriented design in SASSY," in *Joint WOSP/SIPEW Int'l Conf. on Performance engineering*, San Jose, California, 2010, pp. 27–36.

[6] P. Oreizy, N. Medvidovic, and R. N. Taylor, "Architecture-based runtime software evolution," in *Int'l Conf. on Software Engineering*, Kyoto, Japan, 1998, pp. 177–186.

[7] B. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. M. Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. M. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. A. Muller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, and J. Whittle, "Software Engineering for Self-Adaptive Systems: A Research Roadmap," in *Software Engineering for Self-Adaptive Systems*, 2009, pp. 1–26.

[8] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*. Wiley, 2009.

[9] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *Softw. Eng. Notes*, vol. 17, no. 4, pp. 40–52, Oct. 1992.

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

[11] L. R. Rabiner, "A tutorial on hidden Markov models and selected applications in speech recognition," *Proceedings of the IEEE*, vol. 77, no. 2, pp. 257–286, Feb. 1989.

[12] J. Z. Kolter and M. A. Maloof, "Dynamic Weighted Majority: An Ensemble Method for Drifting Concepts," *J. Mach. Learn. Res.*, vol. 8, pp. 2755–2790, Dec. 2007.

[13] G. Widmer and M. Kubat, "Learning in the presence of concept drift and hidden contexts," *Mach. Learn.*, vol. 23, no. 1, pp. 69–101, Apr. 1996.

[14] S. Malek, N. E. Beckman, M. Mikic-Rakic, and N. Medvidovic, "A Framework for Ensuring and Improving Dependability in Highly Distributed Systems," in *Wrkshp. on Architecting Dependable Systems*, Florence, Italy, 2004, pp. 173–193.

[15] A. Elkhodary, N. Esfahani, and S. Malek, "FUSION: A Framework for Engineering Self-Tuning Self-Adaptive Software Systems.," in *Int'l Symp. on the Foundations of Software Engineering*, Santa Fe, New Mexico, 2010, pp. 7–16.

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

[17] E. Alpaydin, *Introduction to Machine Learning*. The MIT Press, 2004.

[18] N. Medvidovic and R. N. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages," *IEEE Trans. Softw. Eng.*, vol. 26, no. 1, pp. 70–93, Jan. 2000.

[19] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid, "Dynamic software product lines," *IEEE Computer*, vol. 41, no. 4, pp. 93–95, 2008.

[20] J. Lee and K. C. Kang, "A Feature-Oriented Approach to Developing Dynamically Reconfigurable Products in Product Line Engineering," in *Int'l Software Product Line Conf.*, Baltimore, Maryland, 2006, pp. 131–140.

[21] C. Cetina, P. Giner, J. Fons, and V. Pelechano, "Autonomic Computing through Reuse of Variability Models at Runtime: The Case of Smart Homes," *Computer*, vol. 42, no. 10, pp. 37–43, Oct. 2009.

[22] J. Whittle, P. Jayaraman, A. Elkhodary, A. Moreira, and J. Araújo, "MATA: A Unified Approach for Composing UML Aspect Models Based on Graph Transformation," in *Transactions on Aspect-Oriented Software Development VI*, 2009, pp. 191–237.

[23] P. deGrandis and G. Valetto, "Elicitation and utilization of application-level utility functions," in *Int'l Conf. on Autonomic Computing*, Barcelona, Spain, 2009, pp. 107–116.

[24] J. P. Sousa, R. K. Balan, V. Poladian, D. Garlan, and M. Satyanarayanan, "User Guidance of Resource-Adaptive Systems," in *Int'l Conf. on Software and Data Technologies*, Porto, Portugal, 2008, pp. 36–44.

[25] C. Chatfield, *The Analysis of Time Series: An Introduction, Sixth Edition*, 6th ed. Chapman and Hall/CRC, 2003.

[26] D. P. Solomatine, "Data-driven modelling: paradigm, methods, experiences," in *Int'l Conf. on Hydroinformatics*, Cardiff, UK, 2002, pp. 1–5.

[27] M. Kantardzic, *Data Mining: Concepts, Models, Methods, and Algorithms*, 1st ed. Wiley-IEEE Press, 2002.

[28] J. R. Quinlan, "Induction of decision trees," *Machine Learning*, vol. 1, no. 1, pp. 81–106, 1986.

[29] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen, *Classification and Regression Trees*, 1st ed. Chapman and Hall/CRC, 1984.

[30] J. H. Friedman, "Multivariate adaptive regression splines," *The Annals of Statistics*, vol. 19, no. 1, pp. 1–67, Mar. 1991.

[31] L. A. Wolsey, *Integer Programming*, 1st ed. Wiley-Interscience, 1998.

[32] S. J. Russell and P. Norvig, *Artificial intelligence: a modern approach*, 3rd ed. Prentice hall, 2009.

[33] S. Malek, N. Medvidovic, and M. Mikic-Rakic, "An Extensible Framework for Improving a Distributed Software System's Deployment Architecture," *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 73–100, Feb. 2012.

[34] G. Edwards, S. Malek, and N. Medvidovic, "Scenario-Driven Dynamic Analysis of Distributed Architectures," in *Int'l Conf. on Fundamental Approaches to Software Engineering*, Braga, Portugal, 2007, pp. 125–139.

[35] S. Malek, M. Mikic-Rakic, and N. Medvidovic, "A Style-Aware Architectural Middleware for Resource-Constrained, Distributed Systems," *IEEE Trans. Softw. Eng.*, vol. 31, no. 3, pp. 256–272, Mar. 2005.

[36] "GME." [Online]. Available: http://www.isis.vanderbilt.edu/Projects/gme/. [Accessed: 08-Mar-2009].

[37] J. Magee and J. Kramer, *Concurrency: State Models and Java Programs*, 2nd ed. Wiley, 2006.

[38] E. M. Dashofy, A. van der Hoek, and R. N. Taylor, "An infrastructure for the rapid development of XML-based architecture description languages," in *Int'l Conf on Software Engineering*, Orlando, Florida, 2002, pp. 266–276.

[39] N. Esfahani and S. Malek, "Utilizing architectural styles to enhance the adaptation support of middleware platforms," *Journal of Information and Software Technology*, vol. 54, no. 7, pp. 786–801, Jul. 2012.

[40] "QVT Final Specification." [Online]. Available: http://www.omg.org/cgi-bin/doc?ptc/2007-07-07. [Accessed: 06-Mar-2011].

[41] H. Giese and R. Wagner, "Incremental Model Synchronization with Triple Graph Grammars," in *International Conference on Model Driven Engineering Languages and Systems (MoDELS 2006)*, 2006, vol. 4199, pp. 543–557.

[42] "WEKA." [Online]. Available: http://www.cs.waikato.ac.nz/ml/weka/. [Accessed: 04-Mar-2010].

[43] M. Gunestas, D. Wijesekera, and A. Elkhodary, "An evidence generation model for web services," in *IEEE International Conference on System of Systems Engineering, 2009. SoSE 2009*, 2009, pp. 1 –6.

[44] "FUSION Project." [Online]. Available: http://www.sdalab.com/projects/fusion. [Accessed: 14-Feb-2013].

[45] S. Malek, "A user-centric approach for improving a distributed software system's deployment architecture," Univ. of Southern California, 2007.

[46] D. Cooray, S. Malek, R. Roshandel, and D. Kilgore, "RESISTing Reliability Degradation through Proactive Reconfiguration," in *Int'l Conf. on Automated Software Engineering*, Antwerp, Belgium, 2010, pp. 83–92.

[47] N. Esfahani, E. Kouroshfar, and S. Malek, "Taming Uncertainty in Self-Adaptive Software," in *Int'l Symp. on the Foundations of Software Engineering*, Szeged, Hungary, 2011, pp. 234–244.

[48] D. A. Menasce, J. P. Sousa, S. Malek, and H. Gomaa, "QoS Architectural Patterns for Self-Architecting Software Systems," in *Int'l Conf. on Autonomic Computing*, Washington, DC, 2010, pp. 195–204.

[49] S. Onn, "Convex Discrete Optimization," in *Encyclopedia of Optimization*, 2nd ed., vol. 1, C. A. Floudas and P. M. Pardalos, Eds. Springer, 2009, pp. 513–550.

[50] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Trans. Auton. Adapt. Syst.*, vol. 4, no. 2, pp. 1–42, May 2009.

[51] R. de Lemos, H. Giese, H. A. Muller, M. Shaw, J. Andersson, L. Baresi, B. Becker, N. Bencomo, Y. Brun, B. Cikic, R. Desmarais, S. Dustdar, G. Engels, K. Geihs, K. M. Goeschka, A. Gorla, V. Grassi, P. Inverardi, G. Karsai, J. Kramer, M. Litoiu, A. Lopes, J. Magee, S. Malek, S. Mankovskii, R. Mirandola, J. Mylopoulos, O. Nierstrasz, M. Pezze, C. Prehofer, W. Schafer, W. Schlichting, B. Schmerl, D. B. Smith, J. P. Sousa, G. Tamura, L. Tahvildari, N. M. Villegas, T. Vogel, D. Weyns, K. Wong, and J. Wuttke, "Software Engineering for Self-Adpaptive Systems: A second Research Roadmap," in *Software Engineering for Self-Adaptive Systems*, Dagstuhl, Germany, 2011.

[52] T. Ryutov, L. Zhou, C. Neuman, T. Leithead, and K. E. Seamons, "Adaptive trust negotiation and access control," in *ACM Symp. on Access control models and technologies*, Stockholm, Sweden, 2005, pp. 139–146.

[53] M. N. Bennani and D. A. Menasce, "Assessing the Robustness of Self-Managing Computer Systems under Highly Variable Workloads," in *Int'l Conf. on Autonomic Computing*, New York, New York, 2004, pp. 62–69.

[54] "DiVA Project." [Online]. Available: http://www.ict-diva.eu/DiVA. [Accessed: 23-Oct-2012].

[55] B. Morin, O. Barais, G. Nain, and J.-M. Jezequel, "Taming Dynamically Adaptive Systems using models and aspects," in *Int'l Conf. on Software Engineering*, Vancouver, Canada, 2009, pp. 122–132.

[56] F. Fleurey and A. Solberg, "A Domain Specific Modeling Language Supporting Specification, Simulation and Execution of Dynamic Adaptive Systems," in *Int'l Conf. on Model Driven Engineering Languages and Systems*, Denver, Colorado, 2009, pp. 606–621.

[57] B. Morin, F. Fleurey, N. Bencomo, J.-M. Jézéquel, A. Solberg, V. Dehlen, and G. Blair, "An Aspect-Oriented and Model-Driven Approach for Managing Dynamic Variability," in *international conference on Model Driven Engineering Languages and Systems*, Toulouse, France, 2008, pp. 782–796.

[58] D. Ardagna, C. Ghezzi, and R. Mirandola, "Rethinking the Use of Models in Software Architecture," in *Int'l Conf. on Quality of Software-Architectures: Models and Architectures*, Karlsruhe, Germany, 2008, pp. 1–27.

[59] Y. Wang and I. H. Witten, "Induction of model trees for predicting continuous classes," in *European Conf. on Machine Learning*, Prague, Czech Republic, 1997.

[60] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf, "An Architecture-Based Approach to Self-Adaptive Software," *IEEE Intelligent Systems*, vol. 14, no. 3, pp. 54–62, May 1999.

[61] P. Inverardi and M. Mori, "Feature oriented evolutions for context-aware adaptive systems," in *Joint ERCIM Wrkshp on Software Evolution and Int'l Wrkshp on Principles of Software Evolution*, Antwerp, Belgium, 2010, pp. 93–97.

[62] A. Gambi, G. Toffetti, and M. Pezze, "Protecting SLAs with surrogate models," in *Proceedings of the 2nd International Workshop on Principles of Engineering Service-Oriented Systems*, Cape Town, South Africa, 2010, pp. 71–77.

[63] G. Tesauro, "Reinforcement Learning in Autonomic Computing: A Manifesto and Case Studies," *IEEE Internet Computing*, vol. 11, no. 1, pp. 22–30, Jan. 2007.

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

[65] D. Zhao and Z. Hu, "Supervised adaptive dynamic programming based adaptive cruise control," in *Adaptive Dynamic Programming And Reinforcement Learning*, Paris, France, 2011, pp. 318 –323.

[66] K. Rieck and P. Laskov, "Language models for detection of unknown attacks in network traffic," *Journal in Computer Virology*, vol. 2, no. 4, pp. 243–256, Feb. 2007.

[67] M. Sabhnani and G. Serpen, "Application of machine learning algorithms to KDD intrusion detection dataset within misuse detection context," in *Int'l Conf. on Machine Learning: Models, Technologies, and Applications*, Las Vegas, Nevada, 2003, pp. 209–215.

Naeem Esfahani is a PhD candidate in the Department of Computer Science at George Mason University (GMU). His current research mainly focuses on software architecture, autonomic computing, and mobile/distributed software systems. Esfahani received his MS degree in Computer Engineering with Software Engineering major from Sharif University of Technology (SUT) in 2008 and his BS degree in Electrical and Computer Engineering with Software Engineering major from University of Tehran (UT) in 2005. Esfahani is a member of ACM SIGSOFT.

Ahmed Elkhodary is currently working in Islamic Development Bank (IDB). Elkhodary received his PhD in Information Technology from George Mason University (GMU) in 2011. He got his bachelor's degrees in Computer Engineering from King Abdul-Aziz University (KSA). He also holds a Master of Science in Software Engineering from George Mason University (GMU). His current research mainly focuses on Software Architecture, Software Product Lines Engineering, Autonomic Computing, and Online Machine Learning.

Sam Malek is an Associate Professor in the Department of Computer Science at George Mason University (GMU). He is also the director of GMU's Software Design and Analysis Laboratory, a faculty associate of the GMU's Center of Excellence in Command, Control, Communications, Computing and Intelligence, and a member of the Defense Advanced Research Projects Agency's Computer Science Study Panel. Malek's general research interests are in the field of software engineering, and to date his focus has spanned the areas of software architecture, autonomic software, and software dependability. Malek received his PhD and MS degrees in Computer Science from the University of Southern California in 2007 and 2004, respectively, and his BS degree in Information and Computer Science from the University of California, Irvine in 2000. He has received numerous awards for his research contributions, including the National Science Foundation CAREER award (2013) and the GMU Computer Science Department Outstanding Faculty Research Award (2011). He is a member of the ACM, ACM SIGSOFT, and IEEE.