On the Role of Features in Analyzing the Architecture of Self-Adaptive Software Systems

Ahmed Elkhodary, Sam Malek, Naeem Esfahani

Department of Computer Science
George Mason University
{aelkhoda, smalek, nesfaha2}@gmu.edu

Abstract. In traditional software families, feature-orientation has been shown effective for bridging the semantic gap between a software system's requirements and its architecture. Over the past few years, the emergence of self-adaptive software systems, which are significantly more challenging to build than traditional systems, has gained the attention of the software engineering research community. In this paper, we show that using features at runtime could alleviate some of the key challenges of building such systems. The underlying insights are that: (1) features allow representation of the engineer's knowledge about some facets of the system that can be used to enhance the adaptation logic, and (2) features can serve as an abstraction to deal with the heterogeneity of the underlying architectural models, analytical algorithms, and implementation platforms. We describe the role of features in a self-adaptive framework that we have developed, entitled FeatUre-oriented Self-adaptatION (FUSION). We also report on our preliminary experience with FUSION that demonstrates the benefits of using features in different stages of self-adaptation.

Keywords: Self-Adaptive Systems, QoS Analysis, Feature-Oriented Modeling

1 Introduction

Feature-orientation has shown to be an effective paradigm for achieving systematic evolution and large-scale reuse in traditional software families [1-3]. In particular, it leverages feature modeling as an intuitive formalism to bridge the semantic gap between end-user requirements and software architecture.

From an end-user's perspective, a feature model decomposes the system's requirements into meaningful units of functionality, known as *features*. A feature serves as an abstraction that is independent of how the functionality is realized by the system. From a software architectural perspective, feature modeling abstracts low-level architectural variability into coarse-grained features that are easier to manage. In turn, it maximizes the reuse potential in the construction of software families. It also helps to ensure the validity of software family members, since features have a

mapping to the low-level architectural constructs and each mapping is functionally validated by the engineer.

In parallel with and largely unaffected by advances in feature-orientation and software product line research, we have witnessed the emergence of self-adaptive systems [4]. Such systems are capable of changing their behavior at runtime to achieve certain functional and QoS goals, which are often specified by the users. Building self-adaptive software systems is significantly more challenging than traditional software systems. In particular, finding the right abstractions that can bridge the gap between end-user goals on one end and their dynamic realization in the software architecture on the other end is challenging.

Given the central role feature-orientation has played in the development of traditional software systems, it is natural to believe its importance to only grow in the even more complex domain of self-adaptive systems. Features are often used during the requirements engineering phase to model the variation points in the software system. At design-time, the engineer develops a mapping for each feature to part of the underlying software architecture that realizes it. This mapping often crosscuts the different parts of the architecture [5]. We advocate an additional role for features that manifests itself at runtime. We believe features provide an appropriate abstraction for modeling the adaptation points (i.e., runtime variability) in the software system [12]. Particularly, features are used in our approach to incorporate the engineer's knowledge of some facets of the system (e.g., the semantic relationship between functional capabilities, QoS properties of concern), which augment the traditional software architectural models to mitigate the challenges of achieving self-adaptation.

In this paper, we describe the role of features in a self-adaptive framework that we have developed, entitled FeatUre-oriented Self-adaptatION (FUSION). Our preliminary experience with FUSION has shown the advantages of using features in the different stages of self-adaptation:

- Features are intuitively understood by both end-users and engineers, making them a convenient medium for eliciting adaptation preferences.
- FUSION's analysis operates on a feature-based representation of the system, decoupling it from the heterogeneity of architectural and analytical models, application domain, and implementation platform.
- Features allow FUSION to correlate results obtained from multiple analytical models to discover interactions and conflicts in the system.
- FUSION uses inter-feature relationships to reduce the configuration space significantly and make the analysis efficient.
- By encapsulating the engineer's knowledge in the mapping of features to the architecture and enforcing feature model constraints, FUSION ensures correct functioning of the system during and after the adaptation.

The rest of this paper is organized as follows. Section 2 uses a motivating example to present some of the key challenges our approach intends to resolve. Section 3 provides a high-level overview of the FUSION framework. Section 4 describes FUSION's underlying feature-oriented model. Sections 5 to 8 describe respectively how features affect the monitoring, analysis, planning, and execution activities in FUSION. Finally, the paper concludes with an overview of our future research.

2 Challenges

We illustrate the concepts in this paper using an online Travel Reservation System (TRS). Fig. 1c shows the software architecture of TRS using the traditional component and connector view. TRS aims to provide the best airline ticket prices in the market. To make a price quote for the user, TRS takes trip information from the users, and then discovers and queries the appropriate travel agent services. The travel agents reply with their price quotes, which are sorted and presented in an ascending order. In addition to the functional goals, the system is required to attain a number of QoS goals such as performance, reliability and security. To that end, solutions for each QoS perspective were developed, e.g., caching for performance, redundancy for reliability, and checkpoints for security.

A system such as TRS needs to be self-adaptive to deal with unexpected situations, such as traffic spikes or security attacks. Therefore, the self-adaptation logic of TRS needs to select from the available adaptation choices. For instance, enable caching to improve performance during a traffic spike or enable authentication to prevent a security attack. To do so, heterogeneous analytical models are required. For example, security engineers may use attack graphs [6] 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 of multiple layers of abstraction (i.e., network, software, user, etc.) to characterize software behavior.

Therefore, applying the existing models of adaptation in the development of self-adaptive systems, such as TRS, is challenged by the following:

Challenge 1: There is no effective mechanism for identifying the interactions and conflicts among the goals in a system using the results obtained from several independent analytical models. For instance, consider the conflict between

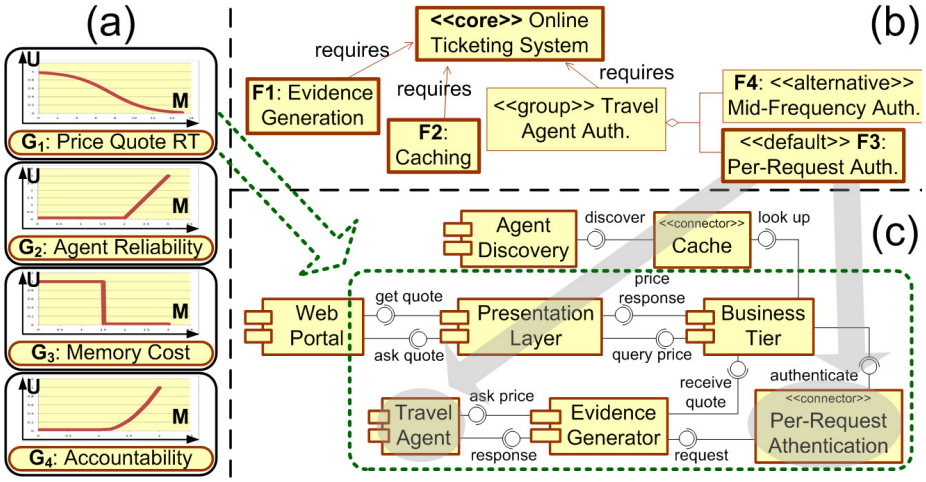

Fig. 1. Travel Reservation System: (a) goals, (b) features, thick border indicates a feature that is enabled, (c) software architecture corresponding to the enabled features.

authentication and the quality of price quotes in TRS. *Business Tier* component waits for a limited time to receive quotes from the *Travel Agent* components before timing out. Since the authentication protocol introduces an additional delay, a heavy authentication protocol may force more timeouts on the *Business Tier*, and hence reduce the number of quotes received by TRS. Building analytical models that could relate the interaction among the system's capabilities and their impact on the system's conflicting goals is often infeasible, as they require representation of complex real-world entities, such as users, networks, service providers, and so on.

Challenge 2: To satisfy multiple goals, self-adaptation logic needs to search in a configuration space that is equivalent to the combined complexity of all analytical models involved. As an example, consider how TRS 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 $(M^P \text{ possible deployments})^{N \text{ possible ways of authentication}} = M^{NP}$ *possible configurations*. Such problem is computationally expensive to solve at runtime for any sizable system. This is while authentication is only one concern out of many in any typical system.

Challenge 3: 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 modeling languages. For instance, consider the problem of representing a constraint in TRS that requires the same authentication protocol to be used on the end-to-end execution flow from the *Web Portal* all the way to the *Travel Agent* and back (depicted in Fig. 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 abstract and represent at the architectural level.

Challenge 4: Effecting a new architecture for a running system may require making changes at the different levels of system stack (e.g., application, middleware, and network). For instance, when a specific authentication protocol is used at the application layer, security engineers may recommend the use of certain IP services at the network layer. In addition, since the recommended IP services come with a performance hit, the engineers may prefer to leave that as an option.

These four challenges have been the prime motivation for our work. As discussed in the remainder of this paper, by adopting a feature-oriented approach, we are able to mitigate these challenges.

3 Overview of Feature-Oriented Self-Adaptation

Changes in the system or its environment trigger the process of self-adaptation. Fig. 2 depicts a high-level overview of FUSION's four main activities: *Monitor*, *Analyze*, *Plan* and *Execute*. These activities are consistent with existing self-adaptive framework's that are based on the feedback control loop reference model [4,7].

However, unlike the majority of existing approaches [8-11] that base the analysis and adaptation on the architectural models, we adopt a feature-based model of adaptation.

At runtime, these activities are performed in the following logical flow:

- *Monitor*: Collects data through instrumentation of the running system. If a functional failure or a violation of QoS objective is detected, it correlates the data into symptoms that can be analyzed.
- *Analyze*: When a problem is detected, it searches for a configuration that resolves it. It may perform a trade-off analysis between multiple conflicting goals.
- *Plan*: Chooses a path of adaptation steps towards the target configuration. The path has to abide by the system constraints. In addition, adaptation steps must not cause further failures in the system.
- *Execute*: Takes the required actions to effect the changes delineated in the plan. This may require adding, removing, and replacing the components and the way they are interconnected in the running architecture.

In the remainder of this paper we describe how using feature-orientation affects and improves the behavior of these activities. Each activity addresses one of challenges introduced in Section 2. The *Feature Based Models*, shown in the middle of Fig. 2, is how the engineer's knowledge of the system's characteristics and its domain is captured and provided for the activities.

4 Feature-Based Models

A feature is an abstraction of a capability provided by the system. A feature may affect either the system's functional (e.g., ticket discounts) or non-functional (e.g., authentication protocol) properties.

Conceptually, features elicited for runtime variability serve a different purpose than traditional ones. The main motivation behind a runtime feature is to account for variability in the system's execution context rather than the end-user requirements. That is, to give the system enough flexibility to cope with an environment where no one solution works perfectly at all times. The goal is to identify critical features required for the system given such variability in the context.

The proposed features are in essence variation points in the architecture rather than requirements. Exposing them as features makes them independent of a particular implementation platform or application domain. For example, in a rule-based system a feature may correspond to a set of rules, in a service-oriented system it may correspond to a set of services in a workflow, in an adaptive system it may correspond to a set

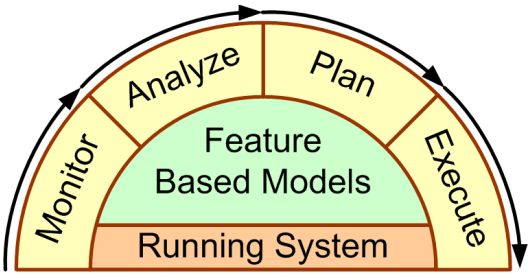

Fig. 2. High-level overview of FUSION.

of adaptation strategies, and so forth. Fig. 1b shows a particular realization of features: a feature is an abstract representation of an architectural variant. As depicted in Fig. 1b, features map to a subset of the system's software architecture. In other words, features crosscut the system's software architecture.

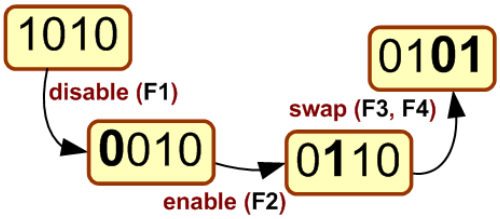

Fig. 3. Feature-based adaptation.

4.1 Runtime Variability

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

In FUSION, at runtime we use the feature model to identify the current system configuration in terms of a feature-selection string. In a feature-selection string, enabled features are set to "1"; disabled features are set to "0". For example, one possible configuration of TRS would be "1101", which means that all features from Fig. 1b are enabled except *Per-Request Authentication* (i.e., F_3).

The adaptation of a system in FUSION is modeled as a transition from one feature-selection string to another (more details in section 7). Each transition takes one of the three forms: enable and disable an optional feature, or swap two mutually exclusive features. Fig. 3 shows three transitions that take the TRS system from feature selection "1010" to "0101".

4.2 Goals

In FUSION, system failure is defined as inability to satisfy one or more system goals. We have adopted a simple, yet very expressive, approach for modeling the system's goals. A goal has a *utility* function for which a system quality *metric* can be optimized. The metric is a measurable quantity (e.g., response time) that can be obtained from a running system. The *utility* function expresses the engineer's preferences for the *metric*. For instance, G_1 (*Price Quote Response Time*) in Fig. 1a specifies a response time metric value to be collected from sensors in the system. The corresponding utility function specifies the user's preferences for different values of price quote response time.

FUSION calculates the system's expected utility for a new feature selection F' for the system as follows:

$$\sum_{g \in G} U_g\left(M_g(F')\right)$$

, where U returns the utility associated with achieving a given metric M of goal g.

A utility function can be used to express hard constraints. In that case the utility function would be a step-function such as the utility of G_4 depicted in Fig. 1a. A utility function may take on more advanced forms (e.g., sigmoid curve), and express more complex preferences, such as G_1, G_2, and G_3.

FUSION places one constrain on the specification of utility functions: they need to return zero for the range of metric values that are not acceptable to the user. When a utility associated with a goal reaches zero, FUSION considers that goal to be violated and initiates adaptation

5 Monitor

As mentioned in challenge 1 of Section 2, quantifying the impact of adaptation choices on the system's conflicting goals are typically difficult (e.g., recall the trade-off between the authentication protocol and the quality of price quotes). We believe this difficulty is partially due to the gap between the system's goals and the low-level units of adaptation (e.g., add/remove component) at the architecture-level. In other words, the adaptation occurs through low-level architectural changes, while the goals are high-level concerns. Achieving a particular goal may require a series of low-level changes at the architecture level. As a result, identifying the impact of low-level changes on the system's goals becomes extremely difficult.

In FUSION, the units of adaptation are features, which are inherently less granular than low-level architectural constructs. In turn, since *Monitor* collects the data at a higher level (i.e., feature level), it is significantly easier to observe and identify the conflicts among goals. In particular, the monitored data in FUSION can be used to determine two kinds of interactions:

1. *Goal interactions* with respect to one feature. A goal interaction occurs when two goals are affected by enabling a feature. For instance, F_1 (*Evidence Generation*) has a positive effect on G_4 (*Accountability*) and negative effect on G_1 (*Price Quote Response Time*), since *Evidence Generation* adds a mediator component to witness the exchange of messages between TRS and travel agents.
2. *Feature interactions* with respect to one goal. A feature interaction occurs when enabling two features modifies the behavior of one or both features. For example, enabling both features F_1 (*Evidence Generation*) and F_3 (*Per-Request Authentication*) has a negative ramification on G_1 (*Price Quote Response Time*) that is beyond the individual impact of each. *Per-Request Authentication* changes the behavior of *Evidence Generation,* since it causes additional overhead in mediating exchange of authentication credentials between TRS and travel agents.

6 Analyze

Analyze conducts runtime analysis to find a configuration of the system that resolves the violated goals. As mentioned in challenge 2 of Section 2, performing such analysis at the architectural-level is often computationally very expensive for any sizable system. FUSION uses features to encode the engineer's knowledge of the adaptation choices that are practical. In turn, *Analyze* operates on the feature selection space, which is significantly smaller than the architecture selection space.

For instance, in the TRS example, the engineer has exposed only the authentication strategies that are foreseen to be useful as features. Fig. 1b shows the two authentication strategies that are modeled as features in the TRS: F_3 and F_4. This automatically reduces the configuration space from M^{NP} (recall example of challenge 2) to 2^F, where F is the number of variant features that affect the authentication concern in the system. Clearly it is reasonable to assume that $M >> 2$ and $N \times P >> F$ for any sizable system.

In addition, using the inter-feature relationships (e.g., mutual exclusions, dependencies) we can further reduce the feature selection space. For instance, Fig. 1b shows a mutual exclusive relationship between F_3 and F_4. This relationship captures the engineer's application knowledge that applying two authentication protocols to the same execution scenario is not a valid configuration. Such relationships reduce the space of valid feature selections significantly.

We can further scope down the analysis to only the features that affect the violated goals. *Analyze* first finds features that have a significant impact (positive or negative) on the violated goal. It then finds any other goals that are affected by the selected features. As a result, FUSION's feature-based analysis is significantly more efficient than the alternative of assessing all of the system goals for the entire space of adaptation choices at the architectural level.

7 Plan

As you may recall from challenge 3 in Section 2, adaptation planning is a major source of difficulty, due to its application dependent nature. This is one of the key shortcomings of existing self-adaptation frameworks, which either ignore or revert to ad-hoc techniques during the planning stage. In FUSION, the engineer models this knowledge in terms of features and their dependencies. This is used to devise a plan that ensures the system's correct functioning during and after the adaptation.

Fig. 3 shows an *adaptation plan* in FUSION, which consists of a series of transitions from the current feature selection to a new one. Since many paths can be traversed to reach a target feature selection, *Plan* uses the feature model to pick a path that abides by feature model constrains in every intermediate step. In TRS for 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 were both features are enabled. Similarly, a dependent feature should not be enabled without its prerequisite. In other

words, the path should not cause transition to an invalid feature selection that could jeopardize the system's functionality.

In addition, guided by utility functions, *Plan* can pick a path that minimizes violation of goals as much as possible. For instance, suppose that enabling F_1 causes 5% decrease in the utility of G_1. If G_1 is already 1% away from violating its constraint, enabling F_1 right away will cause a violation. In such a case, the adaptation plan first enables another feature, suppose F_2, to increase G_1's utility (e.g. up to more than 5% away from the constraint) before enabling F_1.

8 Execute

Execute carries out the process of changing the system's configuration. However, as mentioned in challenge 4 of Section 2, effecting a new architecture may require making changes at different levels of system stack (e.g., application, middleware, network). FUSION uses features as platform-independent effectors. Each feature is associated with a *feature mapping*, which relates the feature to a part of the running system. A feature mapping is a set of rules that specify the changes that need to take place in the lower levels of system stack. For mutual exclusive features, one mapping is created for each mutual group.

Fig. 4 shows how FUSION integrates with the system using a feature mapping interface. In part (a), the feature mapping interacts with multiple platforms at the application level. In part (b), the feature mapping rules extend to different levels of system stack. In both cases, the role of *Execute* is limited to invoking one feature-mapping interface at a time (i.e., enable/disable/swap a feature) regardless of *how* and *where* changes are taking place. For example, enabling a feature may correspond to delopying new components in the application, selecting a new resource allocation policy in the middleware, switching off certain network interfaces, and so on. The feature mapping interface invokes effectors in the running system to apply the changes as specified.

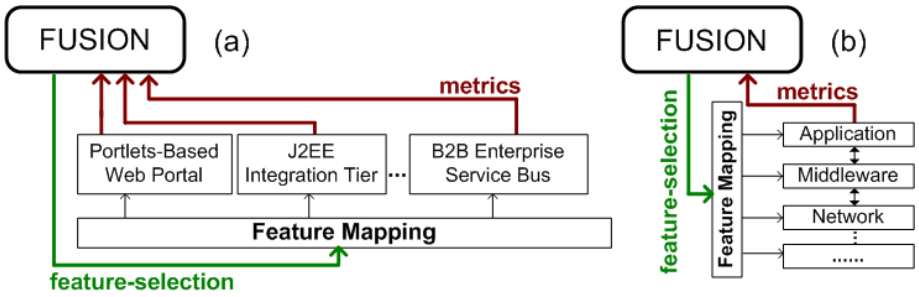

Fig. 4. FUSION uses feature-mapping to integrate with (a) heterogeneous implementation platforms, and (b) different levels of system stack.

9 Conclusion

We described the role of features in a self-adaptive framework, called FUSION. We showed how feature modeling alleviates some of the key challenges of building self-adaptive systems. The underlying insight guiding our research is that: (1) by using features to incorporate the engineer's knowledge of some aspects of the system we can enhance the adaptation logic, and (2) features can serve as an abstraction to deal with the heterogeneity of the underlying architectural models, analytical algorithms, and implementation platforms. As part of our future work we intend to empirically evaluate and compare the FUSION framework against other self-adaptation frameworks. In particular we plan to quantitatively assess the benefits and drawbacks of using feature abstractions for self-adaptation in the context of real-world applications.

Acknowledgments. This work is partially funded by contract W9132V-07-C-0006 with US Army Geospatial Center, as well as grant CCF-0820060 from National Science Foundation. We would like to thank Mark Pullen for his guidance and support in this research.

References

1. Gomaa, H.: Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures, Addison-Wesley Professional, 2004.
2. Kang, K.C., et al.: Feature-oriented domain analysis (FODA) feasibility study. Carnegie-Mellon University, Pittsburgh, PA, Software Engineering Institute, 1990.
3. Kang, K.C., et al.: FORM: A feature-oriented reuse method with domain-specific reference architectures. In: Annals of Software Engineering, vol. 5, 1998, pp. 143–168.
4. Cheng, B. H. C. et al., Software Engineering for Self-Adaptive Systems: A Research Roadmap. In: Software Engineering for Self-Adaptive Systems, Lecture Notes on Computer Science Hot Topics, 2009, pp. 1-26.
5. Lee, K., Kang, K., Kim, M., Park, S.: Combining feature-oriented analysis and aspect-oriented programming for product line asset development. In: 10th International Software Product Line Conference, 2006, pp. 10 pp.-112.
6. Foo, B., Wu, Y., Mao, Y., Bagchi, S., Spafford, E.: ADEPTS: adaptive intrusion response using attack graphs in an e-commerce environment. In: Dependable Systems and Networks, 2005. DSN 2005. Proceedings International Conference on, 2005, pp. 508-517.
7. Andersson, J., de Lemos, R., Malek., S., and Weyns, D.: Modeling Dimensions of Self-Adaptive Software Systems. In: Software Engineering for Self-Adaptive Systems, Lecture Notes on Computer Science Hot Topics, 2009.
8. Garlan, D., Cheng, S., Huang, A., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure, Oct. 2004.
9. Oreizy, P., et al.: An Architecture-Based Approach to Self-Adaptive Software. In: IEEE Intelligent Systems, vol. 14, 1999, pp. 54--62.
10. Kramer, J., Magee, J.: Self-Managed Systems: an Architectural Challenge. In: International Conference on Software Engineering, 2007, pp. 259-268.
11. Malek, S., Mikic-Rakic, M., Medvidovic, N.: An extensible framework for autonomic analysis and improvement of distributed deployment architectures. In: Proceedings of the workshop on Self-managed systems, ACM New York, NY, USA, 2004, pp. 95-99.
12. Lee, J., Kang, K.: "A feature-oriented approach to developing dynamically reconfigurable products in product line engineering," Software Product Line Conference, 2006 10th International, 2006, pp. 10 pp.-140.