A LEARNING-BASED FRAMEWORK FOR ENGINEERING FEATURE-ORIENTED SELF-ADAPTIVE SOFTWARE SYSTEMS

by
Ahmed Elkhodary

A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
in Partial Fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Information Technology

Committee:

____	Dr. Sam Malek, Dissertation Director
____	Dr. Hassan Gomaa, Committee Member
____	Dr. Alexander Levis, Committee Member
____	Dr. Daniel Menascé, Committee Member
____	Dr. Daniel Menascé, Senior Associate Dean
____	Dr. Lloyd J. Griffiths, Dean, Volgenau School of Engineering
Date: ____________________________________	Fall Semester 2011 George Mason University Fairfax, Virginia

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

A Dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University

By

Ahmed Elkhodary
Master of Science
George Mason University, 2006
Bachelor of Science
King Abdul Aziz University, 2004

Director: Sam Malek, Assistant Professor
Department of Computer Science

Fall Semester 2011
George Mason University
Fairfax, VA

Dedication

To the greatest parents in the world (*Mohamed Elkhodary* & *Wijdan Alshanti*).

Acknowledgements

I would like to thank my parents, Mohammad and Wijdan, for their continuous and unlimited support and prayers.

I would like to thank my dissertation director, Dr. Sam Malek, for his scholarly direction, teaching, encouragement, and support.

I would like to thank my Ph.D. committee (Dr. Hassan Gomaa, Dr. Alex Levis, and Dr. Daniel Menasce) for their scholarly direction and guidance throughout the Ph.D. program.

I would like to thank Dr. Daniel Menascé, Senior Associate Dean of the Volgenau School of Engineering, for his generous support, in the form of PhD Fellowships, and PhD Completion Grant.

I would like to thank my friends and school mates, Dr. Naeem Esfahani, Mohammad Abu-Matar, Ashraf Abu-Sharekh, Ahmed Abu-Jbarah, and Mohammad Ababneh for their good company, intellectual discussions, and help.

Last but not least, I would like to thank my family members Lama, Hani, May, and Ala Elkhodary for their selfless help and continuous prayers.

Table of Contents

List of Tables

List of Figures

Abstract

A LEARNING BASED FRAMEWORK FOR ENGINEERING FEATURE-ORIENTED SELF-ADAPTIVE SOFTWARE SYSTEMS

Ahmed Elkhodary, PhD

George Mason University, 2011

Dissertation Director: Dr. Sam Malek

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, in practice, self-adaptive software systems are often complex, dynamic, and the structure of managed system may not be completely known at design time. In addition, 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 make the adaptation decisions inefficient and inaccurate.

In this dissertation, 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 of 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 online learning, which in turn makes online learning feasible.

We present an empirical evaluation of a proof-of-concept implementation 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.

Chapter 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* [33]. 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 [18][20][21][30][33][39][48][52][65] that target the development of such systems, numerous challenges remain [10]. In particular, engineering the adaptation logic poses the greatest difficult as further discussed below.

1.1 Problem Statement

The state-of-the-art [10][33] in engineering self-adaptive software systems is to employ an architectural representation of the software system (e.g., component-and-connector view [66]) for reasoning about the adaptation decisions. We refer to this as the *white-box approach*, since it requires knowledge of the system's internal structure of the managed system. 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* [53], rebinding a component's interfaces, and so on. This paradigm is commonly referred to as *architecture-based adaptation* [33][52][66]. 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 indicators for making the adaptation decisions. For instance, Queuing Network models [25] and Hidden Markov models [56] 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 several 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* [96][97][98]) that were not accounted for during their formulation. In practice, the internal structure of managed software systems may not be completely known at design time. Even when partially known, runtime conditions can radically change the internal structure of the system in ways that were not accounted for during design.

Thus, unanticipated changes at runtime that violate the assumptions made about the internal structure of the system could make the analysis and hence the adaptation decisions inaccurate.

2. **Dependencies.** White-box approach assumes adaptation can be localized in atomic structural changes that can be carried out independently, while often in practice changes in different parts of the architecture need to occur 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 may arise at runtime. However, 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 [10][43][48]. 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.

1.2 Research Statement and Hypotheses

This thesis aims to develop a black-box approach for reasoning about software adaptation. The approach combines feature modeling, as the technique for explicitly representing the engineers' knowledge of practical software adaptation choices, with online machine learning, as the technique for automatically deriving analytical models predicting the impact of features on quality of service objectives. Engineers' knowledge, represented in feature-models, adds structure to online learning and limits the configuration space to the space of valid feature selections, which in turn makes online learning feasible. Black-box –yet fine-grained– analytical models derived through continuous online learning make adaptation efficient and accurate.

The research statement entails investigating the following three hypotheses:

Hypothesis #1: *Online learning could be used to generate a predictive model that quantifies the impact of features on the system's goals based on continuous observation and induction.*

Hypothesis #2: *Features' inter-relationships could be used to devise adaptation plans that 1) enforce dependencies among the system's constituents during and after adaptation (e.g., components, protocols), and 2) ensure the system maintains QoS goals during adaptation.*

Hypothesis #3: *Feature models together with online learning could be used to prune the configuration space, and thereby achieve efficient analysis and adaptation.*

1.3 Research Thrust Areas and Insights

The first thrust of this research is feature-orientation, which has played a central role in achieving systematic evolution and large-scale reuse in traditional software product lines [11][70], thus it is natural to believe its importance will only grow in the more complex domain of self-adaptive systems. A *feature* is an abstraction of a capability provided by the system that affects one or more of its properties [7]. A feature is traditionally used during the requirements engineering phase to model a variation point 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.

In this research, we propose an additional role for features that manifests itself at runtime. We use features as black-box 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 underlying software architecture, it could be used to address global architectural dependencies 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.

The second thrust of this research is to empower the system to "automatically learn how to adapt". In other words, instead of requiring a white-box knowledge about the software system, we propose a learning-based approach in which a predictive model of the system is automatically induced from the monitored data. The approach not only allows for automatic online fine-tuning of the adaptation logic to unanticipated conditions, but also reduces the upfront effort required for building such systems. Indeed, the goal as stated above is too ambitious and unrealistic for architecture-based adaptation, since learning is known to be computationally expensive and the number of alternative architectures is prohibitively large. For this reason, we constrain our approach to the aforementioned feature-oriented adaptation paradigm, where the space of configuration choices is significantly smaller.

1.4 Contributions

In this dissertation, we present 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 [11], and (2) a new method of assessing and reasoning about adaptation decisions through online learning [3]. The result of this research is a framework, entitled FeatUre-oriented Self-adaptatION (FUSION), which combines feature-models (i.e., as an explicit representation of domain experts' knowledge) with online machine learning. Domain knowledge, represented in feature-models, adds structure to online learning, which in turn improves the quality of adaptation decisions. Below lists the key contributions of the FUSION framework as an approach for building self-adaptive software systems:

1. FUSION reduces the complexity of building adaptation logic for software systems, since it does not require an explicit representation of the managed system's internal structure. It only requires a black-box specification of the system's adaptation choices (i.e., features) as input.
2. FUSION copes with the changing dynamics of the system, even those that were unforeseen at design time, through incremental observation and induction (i.e., online learning).

3. 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.
4. FUSION uses features and inter-feature relationships to significantly reduce the configuration space of a sizable system, making runtime analysis and learning feasible.

1.5 Organization of the Dissertation

The thesis is organized as follows: Chapter 2 provides background and representative related work found in the literature relative to software adaptation, analytical modeling approaches, online reasoning for adaptation, and tool support for self-adaptive software systems. Chapter 3 presents a motivating case study that is used to describe the research challenges and concepts presented throughout the thesis. Chapter 4 presents an overview of the FUSION framework. Chapter 5, 6, and 7 provide detailed description of the framework. Chapter 8 describes FUSION MDA-based tool support framework. 0Chapter 9 provides an empirical evaluation of FUSION using TRS to test and compare the learning and decision-making cycles in terms of accuracy and efficiency. Chapter 10 provides concluding remarks, limitations of the work, and directions for future research.

Chapter 2: **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 [5][6][10][33]. We provide an overview of the most notable research in this area and examine them in light of FUSION.

We begin the related work with an overview of key approaches for modeling dynamism and variability in software systems in Section 2.1. Then, Section 2.2 covers a number of approaches for analytical modeling that are commonly used in autonomic and adaptive systems. Section 2.3 lists key literature related to online reasoning for adaptation. Finally, Section 2.4 covers literature related to model driven support for adaptation.

2.1 Modeling Software Adaptation Space

Several methods of modeling adaptation space of dynamic software systems have been proposed. Each approach advocates a different abstraction for modeling the unit of adaptation in the system. We sort the approaches based on the scale of adaptations possible (i.e., local to global).

2.1.1 Parametric Adaptation

Ryutov et al. [59], for instance, provide a security framework that supports adaptive access control and trust negotiation through parameterization. Two parameters change the behavior of the system at runtime: system threat-level, which is collected from an intrusion detection subsystem, and user suspicion-level, which is collected from security logs. Parameterization as a paradigm of adaptation has the advantage of simplicity. However, it lacks support for large-scale adaptation at the system wide level.

Similarly, Menasce et al. [47][94][95] propose a parameterizable 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 is an effective method of building self-adaptive software, it is not applicable for structural adaptation with large-scale reconfigurations.

2.1.2 Component-Based Adaptation

In component-based adaptation [33][48][52][66], adaptation is 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* [53], rebinding a component's interfaces, and so on.

Usually, changes at the architecture level involve prerequisite steps to make sure the system is in a safe state for adaptation. Kramer et al. [33] describe the problem of change management in architecture-based adaptation formally in [107] and introduce the notion of quiescence. Gomaa et al. [23], uses a component-based approach to model adaptation patterns.

Hence, this paradigm enables large-scale adaptation at a system-wide level by swapping distributed components at runtime. Since this work serves as a foundation for this work, we further discuss this paradigm and how it relates to architecture-based reasoning in Section 2.3.1.

2.1.3 Aspect-Oriented Adaptation

The aspect based paradigm was invented to address crosscutting concerns that are related to non-functional requirements. In past work [83], we presented MATA (Modeling Aspects using a Transformation Approach), a UML aspect-oriented modeling technique that uses graph transformations to specify and compose aspects into complete software architectures. The approach uses unified aspect weaving operators for UML class diagrams, sequence diagrams and state diagrams. MATA maintains separation of aspects throughout the lifecycle by generating AspectWerkz code, which uses dynamic weaving facility (i.e., byte-code manipulation) to compose aspects.

Morin et al. [49][50] use aspects as course-granular units of adaptation. Aspects reduce the configuration space significantly and make runtime analysis feasible for large systems. However, since aspects are casually connected, there is no support for specifying user-defined inter-aspect relationships.

A key advantage of this paradigm is its ability to reduce the configuration space significantly when crosscutting concerns are taken in consideration in the system.

However, aspects may have significant interactions for which no accepted modeling formalism can be used to expresses inter-aspect relationships and constraints.

2.1.4 Feature-Oriented Adaptation

FODA is an engineering method that enables massive and large-scale reuse by identifying the common and variant features of a system family [88]. FODA development cycle is based on three phases:

- Context Analysis: analyzes the scope of the domain and defines its boundary by examining external environment and capturing that in a context model.
- Domain Modeling: elicits common and variants functional requirements capabilities within the domain using features. The relations among the features of the domain are represented using a tree structure in which features are hierarchically decomposed.
- Architecture modeling: develops task structures and internal modules of the tasks as well as their relationships using a high-level architectural representation.

PLUS [22] is an engineering method for software product lines that is based on the Unified Modeling Language (UML). PLUS uses features as well to encapsulate commonality and variability in the product line family. throughout the development lifecycle. PLUS enables two lifecycles to run in parallel:

- Product Line Engineering Cycle: enables analysis of the commonality and variability within the system family as a whole using features, which are categorized into kernel, optional, and alternative. Features are then mapped to the software architecture that is modeled using multiple UML views.

- Application Engineering Cycle: enables instantiation of member applications by tailoring a member application specific architecture that realizes user selected features only.

Finally, previous works [26][36][69] have used feature-orientation as a method of modeling the requirements of a dynamic product line. FUSION adopts a similar modeling methodology. However, in FUSION features are used at run-time as the units of adaptation and reasoning, and learning is employed in finding optimal feature selections. In Section 5.1, we discuss how features in FUSION differ from traditional dynamic product line features semantically.

2.2 Approaches for Analytical Modeling

A key feature of self-adaptive systems is the ability to use analytical models of the system as runtime artifacts to reflect on the level of satisfaction of goals and constraints. Such reflection depends on the nature and amount of knowledge represented in these models. We describe in this section a number of analytical models emphasizing on the ones that can or have been used at runtime (i.e., online models).

Analytical models differ in terms of the way knowledge of the behavior of the system is represented. From that perspective, analytical models fall under two broad categories: white-box and black box. We compare the two approaches in the follows sub-sections.

2.2.1 White-Box Approaches

White-box approaches for analytical modeling require an explicit model of the internal structure of the software system (i.e., typically an architectural model). We describe a number of examples in this section. Queuing Networks (QN) [25] are a mathematical models used for performance analysis of a software systems that is represented as a collection of *Queues* (i.e., system resources) and *Customers* (i.e., user requests). Queuing models can be either open, where the number of customers can be infinite, or closed, where the number of customers is known and finite. Layered queuing networks (LQN) models were developed as an extension of QN to model software entities and hardware devices. Software Queuing Networks-Hardware Queuing Networks (SQN-HQN) [103] are 2-layered QNs in which the SQN can be open and the underlying calculations are simple.

Markov models [56] in their simplist form are stochastic models that capture the state of the system using a random variable that changes overtime according to the probability distribution of the previous state. They are heavily used in assessing the reliability of software applications. An important extenstion to simple Markov models is Markov Decision Processes (MDP), which allow multiple probabilistic behaviors to be specified as output from a state.

Petri Nets are mathematical models for the description of distributed systems. In Petri Nets, systems are represented as nodes (i.e., events that may occur), places (i.e., conditions), and arcs (i.e., connect places to arcs). One application of Petri Net to adaptive systems can be found [109][110][111] where a visual model is used to describe the adaptation process of variable organizations as morphing sequence. In each node, one member configuration of the variable organization is depicted. A member configuration is obtained through unfolding the Petri Net that represents the variable organizations.

Hence, the approaches mentioned above require an explicit model of the internal structure of the software system. Such models are typically used at design time to analyze the trade-offs of different architectural decisions before implementation [101]. However, they are being used increasingly at runtime to conduct dynamic analysis using monitoring data. For instance, Menasce et al. [47][48] provide an approach for service discovery in SOA-based systems by using online analytical modeling. The scope of variability in such model is limited to simple parametric adaptation in [47] and is extended to be component-based in [48]. The structure of these models cannot be easily changed at runtime in ways that were not accounted for during their formulation (e.g., appearance of new queues in a QN model due to emerging software contentions).

2.2.2 Black-Box Approaches

Black-Box models (often called surrogate models) do not require knowledge of the internal structure of the system. Black-box approaches are also called data-driven modeling since machine learning on data is the basis for these approaches. A number of these approaches is explained.

Artificial Neural Networks (ANN) is an effective way of solving a large number of nonlinear estimation problems. Kriging [108] is another model that is based on treating predicted response as a realization of a random function with respect to the actual system response. Other types of models include Radial Basis Functions (RBF), Least Interpolating Polynomials, and inductive learning are also widely used.

Model trees are an extension of regression trees that associate leaves with multiple regression models (i.e., M5 Model Trees [10] and Multivariate Adaptive Regression Splines (MARS) [17]). M5 model trees are more understandable than ANNs. In addition, they are easy to configure and to train and capable of handling large number of attributes [105]. In addition, they provide the opportunity for letting users participate interactively in building the classifier at design-time due to the simplicity of the produced model trees [106].

Being a member of the black-box analytical modeling family, the above approaches do not require knowledge of the internal structure of the system as mentioned earlier. However, 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 overtime at runtime). FUSION follows the black-box approach in representing the behavior of a software application.

2.3 Online Reasoning for Self-Adaptive Software Systems

IBM's Autonomic Computing initiative advocates a reference model known as MAPE [30], which is structured as hierarchies of feedback-control loop consisting of the following 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 [5][10].

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.

Oreizy et al. pioneered the architecture-based approach to runtime adaptation and evolution management in their seminal work [52]. The work introduces a framework for runtime component evolution that is based on type theoretic language level support. The framework uses an Architecture based representation that separates computation concerns (i.e., Components) from communication concerns (i.e., Connectors). Runtime time adaptation is facilitated using a C2 architectural style that uses C2 Connectors to route messages among components through implicit invocation thus minimizing component interdependencies.

Kramer and Magee [33] introduced a three-layer reference model for self-adaptation (The terms self-adaptation and self-management are used interchangeably in this document). Designing and developing self-managed systems have shown to be significantly more challenging than traditional systems. The following briefly demonstrates the different levels of abstraction on which reasoning takes place in self-adaptive software systems:

1. **Goal management** is an ongoing runtime process that aims to maintain satisfaction of system goals. Adverse changes in the system or its environment may result in violation of goals. For instance, a sudden traffic spike may cause significant increase in response time and break system objectives. In such conditions, the role of Goal Management is to reflect on system requirements and devise a plan that transitions the system to a configuration in which goals can be satisfied. Goal management is the focus of this research.
2. **Change management** is the process that is responsible for making consistent changes to the software architecture to reach a desired configuration. This may involve effecting changes in arbitrary deployment architectures in which geographically distributed components interact and depend on one another.
3. **Component control** executes individual change steps (i.e., addition, removal, linking, and unlinking) and report success at the end of each step. It needs to

detect and report component’s state (i.e., active, passive, quiescent, failed, etc) accurately before any changes take place to it.

These works served as the foundation of this research. The following subsections discuss different approaches of online reasoning and decision making for runtime adaptation.

2.3.1 Architecture-Based Reasoning

Garlan et al. present Rainbow framework [18], 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. In particular, instead of generating an optimal adaptation plan, an explicit language of representation is provided to capture routine human expertise about system adaptation. The self-adaptation language describes rule-like constructs (condition-action), that is, mapping between invariants and adaptation strategies in the architectural model. Invariants define conditions that the system should maintain and an adaptation strategy is a script that modifies the architecture in response to changes in the executing system properties. Thus, when the running system violates the invariant imposed by the architectural model, the appropriate adaptation strategy is executed to adapt the system.

Malek et al. [73][74], provide a generic framework for improving QoS satisfaction by finding an improved deployment architecture. The framework provides a set of estimator algorithms that can scale for large number of components and hosts. Also, domain experts can express QoS properties using a generic architecture-based representation of the system.

Oreizy et al, [75][76] describe an architectural approach for runtime adaptation and self-healing. Their work uses architectural differences between a previous architectural model and a new one in order to produce an adaptation plan. Then, the adaptation plan is analyzed to validate the change to the architectural model. This technique relies on the use of a predefined set of rules.

Georgiadis et al. [21] propose a decentralized adaptation approach, where each self-organizing component manages its own adaptation with respect to the overall system goal. The goal of each component manager is to find a binding, for each of its required ports, that satisfies the global architectural constraints. If the architecture as a whole needs to evolve, it changes the global constraints.

All of the above approaches, including many others (e.g., see [10]), 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, unlike these approaches, 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 them, FUSION is capable of coping with unanticipated changes through online learning.

2.3.2 Rule-based Reasoning

Related to our research are adaptation frameworks that employ logic and rule based methods of induction. Sykes et al. [65] present an online planning approach to architecture-based self-managed systems. Based on the three-layer model for self-management [33], their work describes plan (i.e., a set of condition-action rules) generation with respect to a change in the environment or a system failure.

Georgas et al. [20] suggest a knowledge-based approach, such that the adaptation polices are specified as logic rules, which are in turn leveraged to derive new policies. The policies connect a contectual condition (i.e., trigger for adaptation) with adaptation actions (i.e., architecture-level changes).

Fleurey and Solberg [77] use two feature oriented representations: one for the system, and the other for the context of the system. At design-time, domain experts specify the rules that relate these two feature models and how they impact system properties.

The above approaches bear resemblance to the proposed work in their use of induction. While rule-based approaches have been shown useful in some settings (e.g., ensuring certain properties hold in the system), they cannot be used for making quantitative analysis of trade-offs between goals. Moreover, these approaches are known to suffer from conflicts in the knowledge base.

2.3.3 Learning-Based Reasoning

Related to our work are autonomic approaches that have employed online learning for reasoning about adaptation. Gambi et al. [104] proposed the use of online machine learning using surrogate models (i.e., a family of Black-Box models) to limit violations of SLA of software applications within Virtualized Data Centers (VDCs). The approach has very close proximity to FUSION. However, the study was focused on relocation of Virtual Machines and the associated resources in a data center rather than adaptation of software application logic itself. In addition, unlike FUSION, the machine learning approach does not apply any state space pruning heuristics to reduce the learning space and improve runtime convergence.

Tesauro et al. [67][68] have proposed a hybrid approach that combines white-box analytical modeling with (i.e., QN models) with Reinforcement Learning (RL) to make resource allocation decisions in data centers. Online learning uses a simplified black-box representation of the running system thus making learning feasible. The white-box QN model is used as a training facility to avoid direct exploration of adaptation decisions on the actual running system. 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. [31] 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. [115][114] uses a hybrid Supervised Reinforcement Learning (SRL) approach [114] that combines the merits of Supervised Learning (SL) and Reinforcement Learning (RL) to develop an adaptive real-time cruise control system. SRL is particularly used for learning problems in which 1) learning space is very large, 2) decision-making time is limited, and 3) there is plenty of domain knowledge that can be used as heuristics to reduce the learning space. For example, SRL was used for real-time adaptive cruise control, robot steering tasks, manipulator control, peg insertion task.

The works in Rieck et al. [57] and Sabhani et al. [60] demonstrate the use of several machine learning algorithms on four types of cyber attacks. The algorithms were used to detect the drifting 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 itself, 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, heuristics search, etc) to make online reasoning feasible.

2.4 Tool Support for Self-Adaptive Software Systems

There have been efforts to hire the best practices in Model-Driven Development (MDD) to generate adaptive software. Matevska et al. [78] discusses the need for dynamic models to enable reconfiguration. In [79] feedback (i.e., monitoring) implemented using models where adaptation is based on simple parameter changes. Languages have been developed for adaptive systems' requirements modeling [80], architecture design [81] and runtime platforms [40]. However, there has been limited tool support for self-adaptation throughout the development lifecycle. Such tool support would mainly rely on MDD and more specifically MDA.

Model Driven Architectures (MDA) is a framework for software development that takes MDD a step further by stressing on 1) precise modeling and 2) automated transformations. In MDA, a model is a description of a system or part of it using a language that has well-formed syntax and semantics such that a computer can interpret it automatically. Models in MDA can be of three types:

1. **Computation Independent Models (CIM)**: enable a level of domain modeling that is independent of the underlying computation architecture. CIMs are typically used to model software requirements and user-centric concerns.
2. **Platform Independent Models (PIM)**: enable a level of architecture level modeling (i.e., using more views) to capture a specification of the system while abstracting away programming language/platform specific details.

3. **Platform Specific Models (PSM)**: captures precise details of the software execution platform (e.g. Java or .NET) for which the running system is implemented.

A transformation is specification of a relationship between two or more models (e.g. higher-level model to lower level model or vice versa) that is defined using a well-form transformation language such that it can be executed by a transformation tool. A key transformation language is that Query-View-Transformations (QVT) language, which is a standard transformation definition language created by OMG. QVT defines three sub-languages (QVT-Operational, QVT-Relations, and QVT-Core) all of which operates on models. A transformation in itself can be regarded as a model. QVT-Relations received high attention due to its intuitive declarative style that permits bi-directional transformation naturally. The QVT-Relations has both textual and visual syntax in order to facilitate definition of consistency rules.

MDA provides a comprehensive lifecycle that takes the system from requirements all the way to code. In the context of self-adaptive systems, however, there has been limited work in integrating MDA throughout the lifecycle of a self-adaptive system. Notable exceptions include code generation from architectural diagrams [82], work in aspect-oriented model-driven engineering [50][83], and Giese et al.'s work on runtime model synchronization [84][85][86]. Lack of such lifecycle-wide support for self-adaptive systems makes the process of building self-adaptive systems unwieldy. Moreover, the complexity involved in constructing such systems grows exponentially with the level of sophistication of self-adaptation logic.

Giese et al. work on runtime model synchronization [85] lays the foundation for enabling online fine-tuning of models for self-adaptive systems by introducing incremental model transformation. It shows how this facility can be used to efficiently implement monitoring [87] which is the first step towards having self-* system. The model synchronization facility provided in the work is bidirectional i.e., it can both transfer change from a source model to a target model and backwards at runtime.

In [86], the approach augments a MAPE model with model synchronization. MAPE provides the control-loop while model synchronization provides the flexibility to change the architecture and therefore achieve compositional adaptation. The approach does not address specific self-management concerns (i.e., goal management, change management, and component control). In Becker et al. the work addresses self-management concerns specifically by attempting to provide model-driven support for the 3 layer reference model using model transformation and synchronization [84]. However, the work uses architecture based representations for both Goal and Change Management layers, where both layers use PIM. Goal Management layer is based on relatively low level representations, which reduces the potential for maintaining separation of goal and change management concerns. This problem also holds for some other related works [50] where adaptation logic is expressed in terms of platform specific artifacts and as such separation of concerns is not achievable. In addition, Component Control concerns are not addressed as they can be best handled using PSM.

Chapter 3: Motivating Case Study

For illustrating the concepts in this thesis, we use an online Travel Reservation System (TRS) that provides a web portal for making travel reservations remotely. Figure 1 shows a subset of this system's software architecture using the traditional component-and-connector view [66]. 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 user, 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, increase authentication to thwart a security attack, and enable logging 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 of multiple layers of abstraction (i.e., network, software, user, etc.) to characterize software behavior

As mentioned earlier, the construction of adaptation logic for such a system is challenging.

3.1 Concept Drifts

Consider a QN model that quantifies the impact of an adaptation decision on the

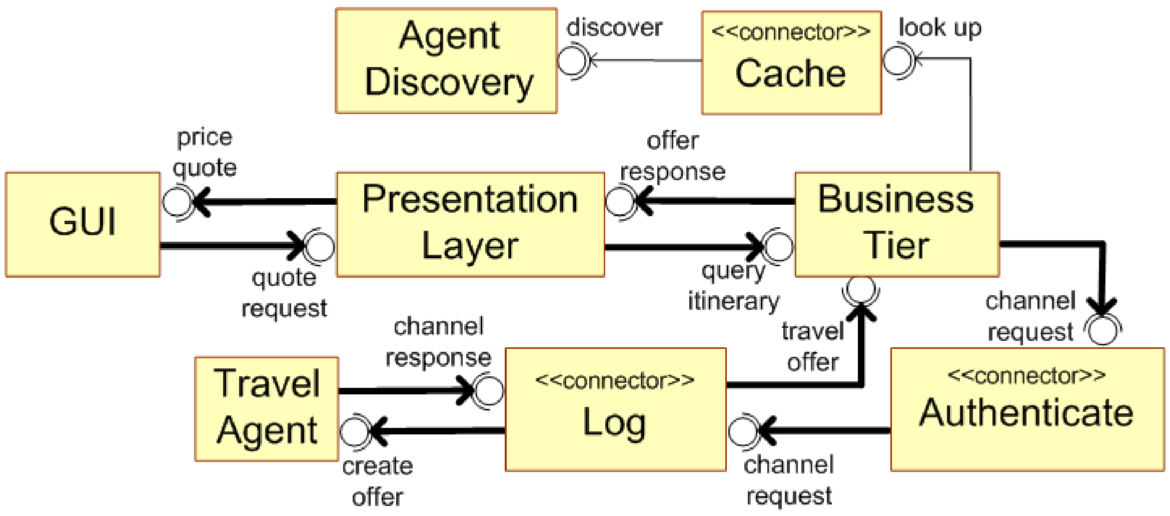


Figure 1. Software architecture of the Travel Reservation System (TRS); thick lines represent an execution scenario associated with receiving a price quote.

response time of receiving price quotes from travel agents (thick lines in Figure 1). 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 dynamically); 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.

3.2 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 [46]. 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 1). 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.

3.3 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 possible. 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 $(M^{P}\ \textit{possible deployments})^{N\ \textit{possible ways of authentication}} = M^{NP}\ \textit{possible configurations}$. 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.

These difficulties form the prime motivation behind our work, which instead of using 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.

Chapter 4: **FUSION Overview**

Figure 2 depicts the envisaged 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 will be discussed in Chapter 5.

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

- Based on the metrics collected from the running system, *Detect* calculates the achieved utility (i.e., measure of user’s satisfaction) to determine if a goal violation has occurred.
- When a goal is violated, *Plan* searches for an optimal configuration (feature selection). The optimal feature selection minimizes the negative impact of goal violation on the system’s overall utility.

- Given a new feature selection, *Effect* determines a set of adaptation steps (i.e., enable/disable features) to place the system in the new configuration. The steps have to abide by the feature model constraints (i.e., dependencies and mutual exclusion relations) to ensure consistency during the adaptation.

FUSION uses *learning cycle* (depicted in Figure 2) to induce 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

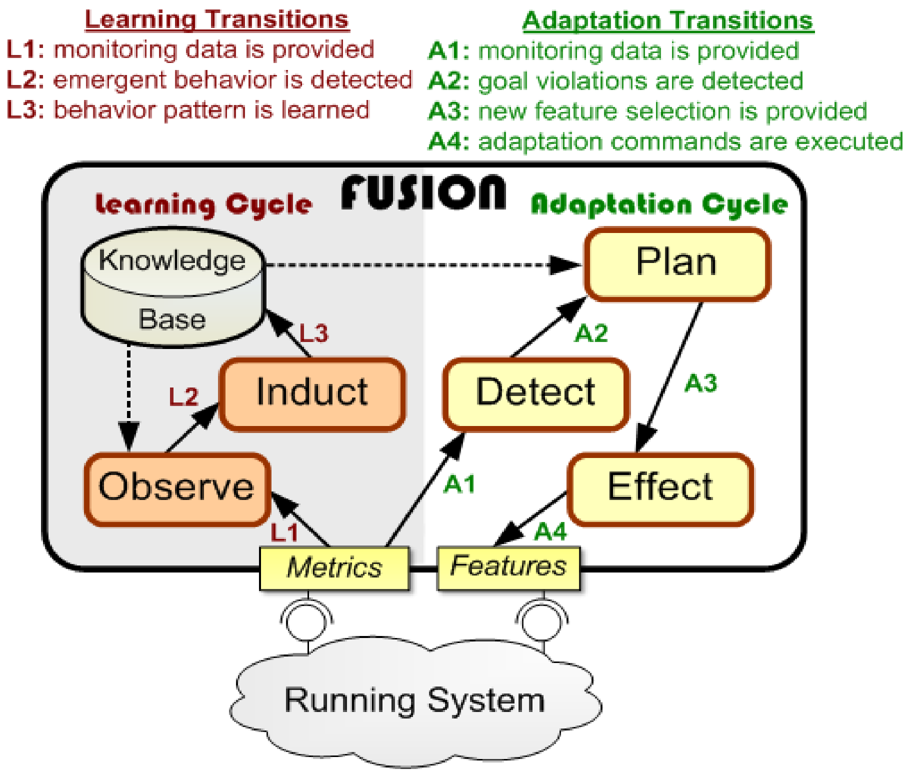

Learning Transitions
L1: monitoring data is provided
L2: emergent behavior is detected
L3: behavior pattern is learned
Adaptation Transitions
A1: monitoring data is provided
A2: goal violations are detected
A3: new feature selection is provided
A4: adaptation commands are executed
Learning Cycle
FUSION
Adaptation Cycle
Knowledge Base
Plan
L3
A2
Induct
Detect
L2
A3
Observe
Effect
A1
L1
A4
Metrics
Features
Running System

Figure 2. Overview of the FUSION framework.

is collected. This data is then used to train FUSION, as it induces 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 two activities in the following sequence:

- Based on the measurements collected from the running system, *Observe* normalizes the data in preparation for learning and detects any emerging patterns of behavior. A potential emergent pattern is detected when the system sets wrong expectations (e.g., inaccurate prediction of utility for an adaptation step).
- *Induce* learns the new behavior and stores the refined model in the *knowledge base* so that informed adaptation decisions can be made in the adaptation cycle.

In the following section, we demonstrate the key ideas in FUSION that make online learning feasible for self-adaptive software systems.

Chapter 5: FUSION Model

We first describe FUSION's modeling methodology, which forms the centerpiece of our approach. As will be detailed in Sections 5.1 and 5.2, FUSION's feature-oriented models enable effective learning and analysis by allowing domain experts to specify key factors in the software system (i.e., whether at the domain, architecture, or platforms level) that affect system goals.

5.1 Feature-Oriented Adaptation

In FUSION the unit of adaptation will be a *feature*. A feature is an abstraction of a capability provided by the system [7][11]. A feature may affect either the system's functional (e.g., ability to select seats in a reservation) or non-functional (e.g., authentication feature for security) properties. In traditional software product lines, a feature is used during the requirements engineering phase to model a variation point 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. We assume an additional role for features that manifests itself 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 underlying software architecture, 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.,

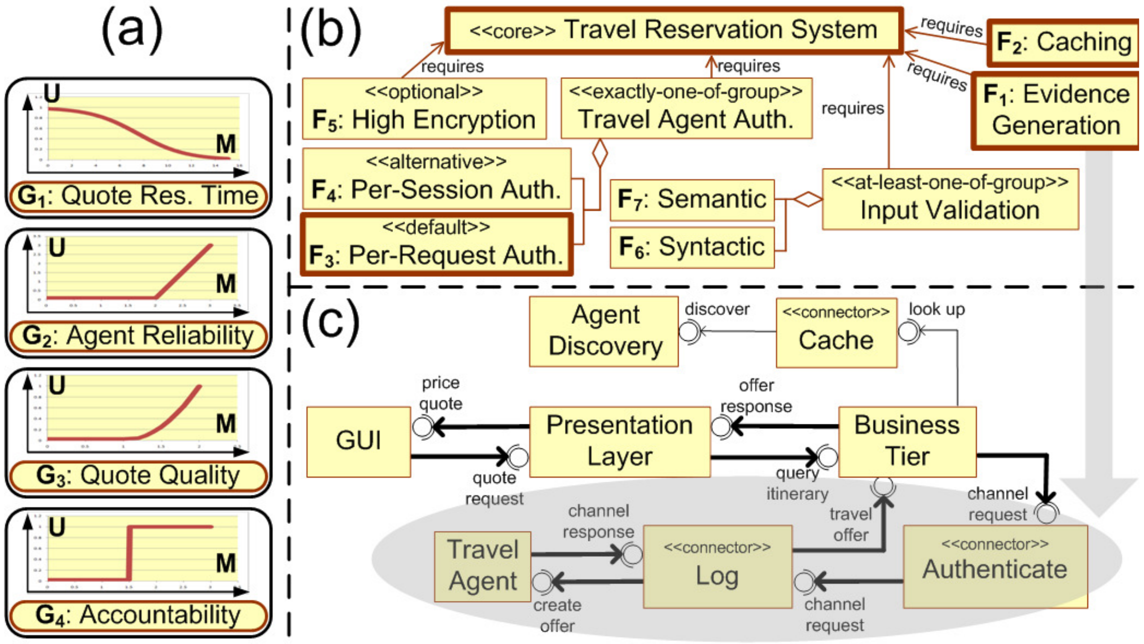


Figure 3. Travel Reservation System: (a) goals are quantified in terms of utility obtained for a given level of metric; (b) subset of available features, where features with thick borders are selected; (c) software architecture corresponding to the selected features, where the thick lines represent an execution scenario associated with goal G1.

software architecture) rather than the problem domain (i.e., requirements) as in traditional software product lines.

The use of features as an abstraction makes the FUSION framework independent of a particular implementation platform or application domain. For example, features may correspond to configuration parameters that are expressed in configuration files as in Figure 4a. Features may be realized using aspects that are weaved to the running system dynamically when the corresponding feature is enabled [27][83] as in Figure 4b. In this research, we adopt a particular realization of a feature: a feature represents an extension of the architecture at well-defined *variation points* as in Figure 4c. A feature maps to a subset of the system's software architecture (refer to 8.2.1). For example, Figure 3b 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 3b shows a simple feature model for TRS. There are four features in the system and one common *core*. The features in the example use two kinds of relationships: *dependency* and *mutual exclusion*. The dependency relationship indicates that a feature requires the presence of another feature. For example, enabling the *Evidence Generation* feature requires having the *core* feature enabled as well. Mutual exclusion is another relationship, which implies that if one of the features in a mutual *group* is enabled, the others must be disabled. For example, *Per-Request Authentication* and *Per-Session Authentication* cannot be enabled at the same time as they belong to the same mutual group. Feature modeling supports several other types of feature relationships (see [89][90]) that for brevity are not discussed here.

We adopt the feature modeling notation used in [22], which supports several other types of inter-feature relationships such as all-or-zero-of groups and at-least-one-of groups described in further details in section 8.1.1.

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"; disabled features are set to "0". For example, one possible configuration of TRS would be "1101111", which means that all features from Figure 3b would be enabled except *Per-Request Authentication*. The adaptation of a system is modeled as a transition from one feature selection string to another, which we describe in Section 7.3.

We incorporate additional feature relationships into the FUSION framework. For instance, some other feature relationships not depicted in Figure 3b include zero-or-one-of, one-or-more-of, mutually-include, etc [22][28]. 0Chapter 8 demonstrates how features

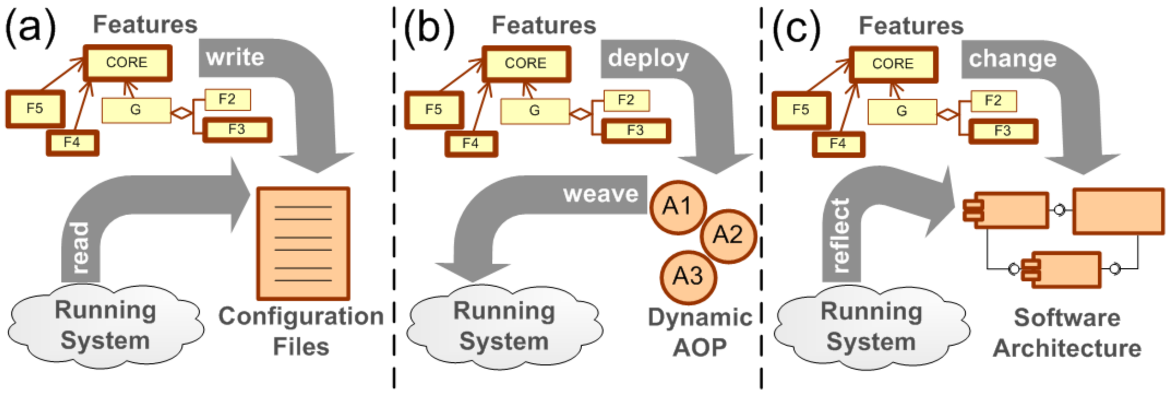


Figure 4. Features can be realized in many ways: (a) configuration parameters modified by the self-management layer at runtime and read by running system; (b) aspects that are weaved into the running system when the corresponding feature is enabled; and (c) architecture mapping, which then modifies the running system by adding/removing components and connectors.

are specified and mapped to the underlying software architecture in a tool support chain.

5.2 Goals

In FUSION, a *goal* construct 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 Chapter 8.

A utility function is used to express the user's preferences (satisfaction) for achieving a particular metric. For instance, G_1 in Figure 3a specifies the user's degree of satisfaction (*U*) with achieving a specific value of *Quote Response Time* (*M*). A utility function could be used to simply express hard constraints. In that case the utility function would be a step-function such as the utility of G_4 depicted in Figure 3a. A utility function may take on more advanced forms (e.g., sigmoid curve), and express more complex preferences, such as G_1, G_2, and G_3.

FUSION places one constraint on the specification of utility functions: they need to return a value less than zero for the range of metric values that are not acceptable to the user. As will be discussed in Section 7.1, when a utility associated with a goal is less than or equal to zero, FUSION considers that goal to be violated and initiates adaptation.

Several previous works [54][63][64] have demonstrated the feasibility of accurately capturing the user's QoS preferences in terms of utility. 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).

5.3 Context

In FUSION, a *context* is an input from the environment of the software system that affects system goals. A typical example of a context variable is workload. Workload affects performance related QoS goals such as response time. Security goals may also be sensitive to context variables such threat level, which is used by intrusion detection systems to raise/lower the level of authentication and access control to system resources. We describe in Section 6.2 how FUSION quantifies the impact of such context variables.

5.4 Incorporating Engineers' Knowledge

Figure 5 demonstrates the inherent advantage of FUSION's approach for reducing software adaptation space. In architecture based approaches, runtime reasoning operates on the full architectural configuration space in which a vast majority of the configurations can be invalid. Recall from 3.3, analyzing the impact of *N* authentication components on *M* software components, which may be deployed on *P* different hardware platforms requires exploring a space of $(M^P \textit{ possible deployments})^{N \textit{ possible ways of authentication}} = M^{NP}$ *possible configurations*. As a result, not only does the process become inefficient, but also engineers end up defining a significant number of complex constraints most of which are for validation purpose (i.e., to detect invalid configurations and exclude them from the space).

In FUSION, instead of operating on the full architectural configuration space, the feature model eliminates a significant portion of the configurations by design and provides only the universe of valid configurations. Engineers of the self-adaptive logic expose sensible adaptation choices to the feature level (i.e., as features). As mentioned in Section 5.1, features produce a Boolean action space (i.e., a feature can be either "1" = enabled or "0" = disabled). Thus, the full adaptation space is limited to 2^F, where F

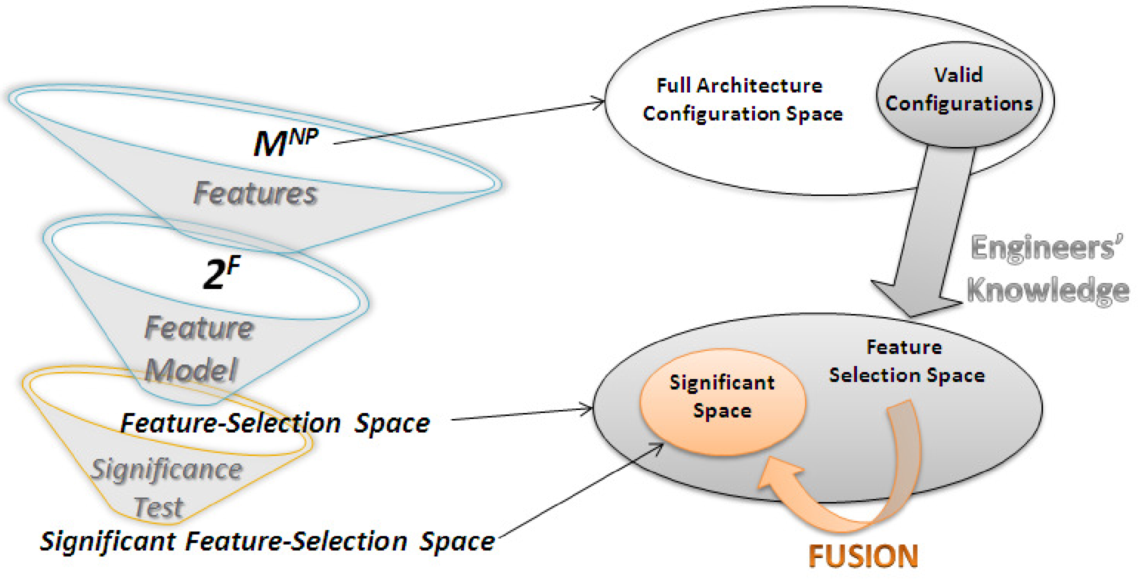


Figure 5: FUSION uses the feature model to incorporate engineers' knowledge of architectural dependencies and the valid configurations of the system.

represents the set of all the features in the system.

In addition, engineers create feature relationships as constraints that limit the space of valid combinations of features even further. This produces the Feature Selection Space as depicted in Figure 5. Note that applying feature model constraints at runtime is a feasible process due to their simplicity (i.e., linear expressions) compared to architecture-level ones as will be demonstrated in Section 7.2.

In addition to these reductions, which are determined at design time by engineers, FUSION can further narrow down the Feature Selection Space through machine learning. Often, machine learning algorithms use some mechanism to determine the subset of features (*principle components*) that have significant impact on a metric. This produces the Significant Space that is depicted in Figure 5.

5.5 Assumptions and Limitations of the FUSION Model

FUSION model makes the following key assumptions:

1. Metrics can be collected from the running system and be exposed at the FUSION model level. The FUSION model abstracts away the complexity involved in collecting such data accurately.
2. Utility functions accurately represent users' preference.
3. Utility function does not require any elements of the system's internal structure in its formulation. In other words, utility is merely a function of metrics.

4. FUSION does not have access to the architecture space. Therefore, adaptation choices are limited to the ones exposed at the feature level by the engineers.

Chapter 6: **FUSION Learning Cycle**

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

Learning is typically a very computationally intensive process. Learning at the architecture-level is infeasible for any sizable system, which is the reason why its application in this domain has been limited. As you may recall from Section 3.3, there are M^{NP} possible configurations for a software system with *M* software components, *N* authentication options, and *P* platforms. Clearly, learning in such an enormous solution space is infeasible. FUSION's feature-oriented model offers three opportunities for tackling the complexity of learning:

1. Learning operates on *feature selection space*, which is significantly smaller than the traditional architectural-level configuration space. The features in FUSION encode the engineer's domain knowledge of the adaptation choices that are practical in a given application. For instance, in the above authentication example, the engineer may expose only the authentication strategies that are meaningful. Figure 3b shows

two authentication strategies modeled as features in TRS: F_3 and F_4. These two features represent what the TRS engineer envisioned to be the reasonable applications of authentication in the system.

2. By using the feature relationships (e.g., mutual exclusions, dependencies), one can significantly reduce the feature selection space. For instance, Figure 3b shows a mutual exclusive relationship between F_3 and F_4. This relationship is a manifestation of the domain knowledge that applying the two authentication protocols at the same time is not appropriate. Such relationships reduce the space of valid feature

```
SelectionCounter(Feature F) :int
int Count = 1;
switch (F.Type)
   case "LeafFeature":
      Count = 2;
   case "exactly-one-of-group":
      foreach ( C in F.Children)
            Count +=SelectionCounter(C) - 1;
   case "zero-or-one-of-group":
      foreach ( C in F.Children)
            Count +=SelectionCounter(C) - 1;
            Count ++;
   case "zero-or-all-of-group":
      foreach ( C in F.Children)
            Count +=SelectionCounter(C);
            Count -= F.Children.Count  + 1;
   case "at-least-one-of-group":
      foreach ( C in F.Children)
            Count *=SelectionCounter(C);
            Count --;
   default:
      for each ( C in F.Children)
         Count *=SelectionCounter(C);
         Count ++;
return Count;
```

Figure 6. Algorithm for sizing the feature selection space.

selections, further aiding FUSION to learn their trade-offs with respect to goals.

3. FUSION applies a significance test that further narrows the feature space for a given metric down to the features that affect its values. For instance, in TRS, it has been shown (see Section 9.1.2) that approximately 88% of TRS features can be eliminated from the learning space of a given metric on average.

Figure 6 shows an algorithm that determines the size of the valid feature selection space in our preliminary feature model recursively. Applying this algorithm to the feature model in Figure 3b yields a space of 48 valid feature selections, calculated as follows: $2 \text{ from } F_1 \times 2 \text{ from } F_2 \times (2 \text{ from } F_3 + 2 \text{ from } F_4 - 2) \times 2 \text{ from } F_5 \times (2 \text{ from } F_6 + 2 \text{ from } F_7 - 1)$. Without considering the inter-feature relationships to prune the invalid selections, the space of feature selections would have been $2^{\text{number of features}} = 2^7 = 128$.

The first cycle of learning takes place when engineers' populate FUSION's knowledge base with training data and an initial set of induced functions. Consequently, at runtime, the learning cycle fine-tunes the induced functions to accommodate emergent behaviors. In 8.3, we demonstrate some practical considerations for the training. The rest of this section describes the two activities that take place to populate and fine-tune the knowledge base.

Table 1. Normalized observation records

Independent Variables								Dependent Variables				
F_1	F_2	F_3	F_4	F_5	F_6	F_7	..	M_{G1}	M_{G2}	M_{G3}	M_{G4}	..
0	0	0	1	0	0	0	..	-0.842	-0.308	1.432	0	..
1	0	0	1	0	0	0	..	0.650	0.513	1.371	2	..
0	1	0	1	0	0	0	..	-1.470	-0.719	1.378	1	..
0	0	1	0	0	0	0	..	-0.132	-0.103	0.740	0	..
0	0	0	1	0	0	0	..	-0.736	-1.335	1.103	1	..
..	..	..	..	..	..	..	..	..	..	..	..	..
1	0	1	0	0	0	0	..	1.574	1.951	0.550	2	..
1	1	0	1	0	0	0	..	0.153	0.513	1.090	2	..
1	1	1	0	0	0	0	..	0.804	-0.513	0.562	2	..
..	..	..	..	..	..	..	..	..	..	..	..	..

6.1 Observe

Observe is a continuous execution of two activities that: (1) normalize raw metric values so that they are suitable for learning, and (2) test the accuracy of learned functions to determine whether further learning is needed or not. We will describe each of these activities below.

Learning in terms of raw data hampers the accuracy. For instance, consider the fact that the actual impact of a feature on a metric may depend on time in the day, which is one factor that has been overlooked by engineers. Therefore, the actual metric data obtained from executing the same software system (i.e., same feature selection) under different times may result in starkly different metric readings, thus making it difficult to generalize in the form of a induced function.

To address this issue, *Observe* takes raw metric data through an automated normalization process prior to storing them as observation records. Many normalization techniques can be applied to transform the learning inputs into a representation that is less sensitive to the execution context. Table 1 demonstrates how observations are collected in FUSION. Observation records are normalized using *studentized residual* [8] 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. Other normalization methods, such as *min-max normalization* [8], *time-series detrending* [12], were not used in this research due to certain limitations. For instance, if the data contains outliers, min-max normalization compresses the data points down to a very small interval.

Once a preliminary set of functions are learned (details of the approach provided in the next section), *Observe* continuously tests the accuracy of induced functions against the latest collected observations. Accuracy is defined as the difference between predicted value of a metric using the induced functions and actually observed value. For that purpose, one could 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 it is not desirable to run the learning algorithm frequently. If the accuracy test fails, *Observe* takes this as an indicator that either learning is incomplete or new patterns of behavior are emerging in the system and, thus, notifies the *Induce* activity to fine-tune the induced functions using the latest set of observations.

6.2 Induce

Based on the collected observations, the *Induce* activity constructs a function that estimates the impact of making a feature selection on the metrics. Induce executes two steps in order to obtain these functions as described below. The first step is a *significance test* that determines the features with the most significant impact on each metric (also known as feature extraction). This allows us to reduce the number of independent variables (recall Table 1) that learning needs to consider for each metric. After the significance test, we apply the learning algorithm, which for each goal, given the normalized observations and the features with significance, derives the corresponding relationships.

While FUSION is not tied to a particular learning algorithm, in our implementation we have used the M5 model tree (MT) algorithm [10], which is a machine learning technique with three important properties: (1) ability to eliminate insignificant features automatically, (2) fast training and convergence, (3) robustness to noise, and (4) simplifies significance testing.

Table 2 shows an example of how the induced relationships among features and metrics look like. We eliminated contextual variables (e.g., workload) from the table to the simplify the demonstration of the approach. The empty cells correspond to

Table 2. Induced functions for metrics M_{G1}-M_{G4}. An empty cell means that the corresponding feature has no significant impact.

Significant Variables	Induced Functions				
	M_{G1}	M_{G2}	M_{G3}	M_{G4}	..
Core	-0.843	-0.161	1.332	0	..
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				..
..	..	..	..	..	..

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

$$M_{G1} = \mathbf{1.553}\, F_1 - \mathbf{0.673}\, F_2 + \mathbf{0.709}\, F_3 + \mathbf{0.163}\, F_1F_3 - \mathbf{0.843} \qquad \text{(Eq. 1)}$$

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 (“1010000”) can be calculated as follows:

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

When making adaptation decisions, values obtained from the induced functions (e.g., 1.482 from Eq. 2 above) would need to be denormalized by applying the inverse of normalization equation used in *Observe*. 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*) at the same time increases M_{G1}. This is because according to Table 2, F_1F_3 increases the response time by a magnitude of 0.163, which decreases the utility of G_1 (utility of G_1 is shown in Figure 3a). This feature interaction is depicted in Figure 3c and can be explained as follows. F_1 introduces a delay by adding a mediator connector, called *Log*, that records the transactions with remote travel agents. At the same time, F_3 changes the behavior of the *Log* ,as it causes an additional delay in mediating the exchange of per session authentication credentials .Enabling the two features at the same time has a negative ramification that is beyond the individual impact of each.

In reality, learning needs to incorporate some contextual factors as independent variables, due to their impact on metrics. Consider the impact of system workload at different times of day on response time. 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):

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

Where *w* is the average inter-arrival time between requests in milliseconds; lower inter-arrival time implies higher workload. Here, the generated functions indicate that TRS reaches saturation when *w* is in the range of 1.21–1.29 milliseconds. Since the impact of features on M_{G1} changes dramatically in that range, the learning algorithm produces a separate equation targeted at that. Although these equations may be of any type (e.g., linear, sigmoid, or exponential), for clarity we have limited the discussion to multi-linear equations only that can be generated from the M5 algorithm.

Note that other methods of representing feature-metric relationships (i.e., induced functions) are also possible. For instance, in the case of discrete metrics, classification-based techniques [29] are more suitable, as they can efficiently represent such relationships in the form of decision trees [72].

M_{G4} for example is a discrete variable that has five levels: *Very-Low=0, Low=1*, *Medium=2*, *High=3* and *Very-High=4*. In this case, learning needs to use a classification based learning algorithm, such as Decision Trees [72], which typically produces 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 form as follows (Hence, each level is mapped to distinct integer value (i.e., 1-5)):

$$M_{G4} = \begin{cases} \mathbf{4}, & F5 = 1 \\ \mathbf{2}, & F1 = 1 \\ \mathbf{1}, & F4 = 1 \\ \mathbf{0}, & else \end{cases}$$

Induce can leverage sophisticated learning techniques, such as CART [19] and MARSplines [17], which provide hybrid representations that combine classification and regression in one model to find an optimal configuration of the system.

Chapter 7: **FUSION Adaptation Cycle**

In this chapter, 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 that "*if the system works, do not change it; when it breaks, find the best fix for the broken part*". While intuitive, this approach sets FUSION apart from the existing research that either attempts to optimize the entire software system (e.g., [45]), or solely solves the constraints (i.e., violated goals) in the system (e.g., [18]). FUSION will adopt a middle ground that performs only partial optimization of the architecture. We argue this is the most sensible approach, and achieves the following objectives:

1. *Reduce Interruption*: Adaptation typically interrupts the system's normal operation (e.g., transient unavailability of certain functionality and higher response time during adaptation). In turn, even if at runtime a solution with a higher overall utility is found, the engineer 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*: Since the adaptation occurs at runtime, the computing overhead of executing the analysis and performing adaptation is crucial. As detailed below, FUSION uses the learned knowledge to scope the analysis to only the parts that are affected by adaptation, hence making it significantly more efficient than assessing the entire software system.

3. *Stable Fix*: Given the overhead and interruption associated with adaptation, effecting solutions that provide a temporary fix are not desirable. We would like FUSION to minimize frequent adaptation of the system for the same problem. To that end, instead of simply satisfying the violated goals, FUSION finds a solution that is optimal and hence less likely to be broken due to fluctuations in the system.

7.1 Detect

The adaptation cycle is initiated as soon as *Detect* determines a goal violation. This is achieved by monitoring the utility functions (recall Section 5.2). A utility function serves two purposes in the adaptation cycle: (1) when the metric values are unacceptable, returns zero or a value less than zero but greater than "-1" to indicate a violated goal (i.e., hard constraint), and (2) when the metric values satisfy the minimum, returns a positive value less than "1" to indicate the engineer's preferences for improvement. Therefore, utility is not only used to initiate adaptation, but to also perform a trade-off analysis between competing adaptation solutions (i.e., feature selections), such that an optimization of the solution can be achieved.

7.2 Plan

To achieve the adaptation objectives, FUSION relies on the knowledge base to generate an optimization problem tailored to the running software:

- Given a violated goal, it uses the knowledge base to eliminate all of the features with no significant impact on the goal from the set *F*, which represent the full space of adaptation units (i.e. features). We call the list of features that may affect a given goal *Shared Features* $\subseteq F$. Consider a situation in the TRS where G_2 is violated. By referring to column M_{G2} in Table 2, we can eliminate feature F_3, since it has no impact on G_2's metric. In this example *Shared Features* = $\{F_1,F_2,F_4\}$.

- *Shared Features* represent our adaptation parameters. These features may also affect other goals, the set of which we call the *Conflicting Goals*. To detect the conflicts, again we use the knowledge base, except this time we backtrack the learned relationships. For each feature in the *Shared Features,* we find the corresponding row in Table 2, and find the other metrics that the feature affects. In the above example, we can see that features F_1, F_2, and F_4 also affect metrics M_{G1} and M_{G4}, and hence the corresponding goals, G_1 and G_4.

By using the knowledge base, FUSION generates an optimization problem customized to the current situation. The objective is to find a selection of *Shared Features*, F^*, that maximizes the system's overall utility for the *Conflicting Goals* given a system context C (e.g., workload) as follows:

Table 3.Constraints to the optimization problem generated by Plan.

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

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

where U_g represents the utility function associated with the metric M_g of goal g (recall Figure 3a). Since we do not want the solution to violate any of the conflicting goals, the optimization problem is subject to:

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

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

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

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

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

This inequality does not hold if a child (dependent) feature is enabled without its parent being enabled. Applying this formulation to the TRS scenario in which G2 is violated generates the following optimization problem:

$$\begin{aligned}
&Shared\ features = \{F_1, F_2, F_4\} \\
&argmax_{(F)} U_{G1}(M_{G1}(F)) + U_{G2}(M_{G2}(F)) + U_{G4}(M_{G4}(F)) \\
&Subject\ to:\ U_{G1}(M_{G1}(F)) > 0 \\
&\qquad U_{G2}(M_{G2}(F)) > 0 \\
&\qquad U_{G4}(M_{G4}(F)) > 0 \\
&\qquad F_3 + F_4 \leq 1 \\
&Where:\ M_{G1} = 1.553\ F_1 - 0.673\ F_2 + 0.709\ F_3 + 0.163\ F_1F_3 - 0.843 \\
&\qquad M_{G2} = 1.137\ F_1 - 0.938\ F_2 - 0.174\ F_4 - 0.161 \\
&\qquad M_{G4} = \begin{cases} 4, & F5 = 1 \\ 2, & F1 = 1 \\ 1, & F4 = 1 \\ 0, & else \end{cases}
\end{aligned}$$

Note that by eliminating F_3 , F_5 , F_6 , and F_7 (as well as U_{G3}) from the optimization problem, we obtain an optimization problem tailored to the violated goals. The customized problem has less number of features and goals than the original problem. It is reasonable to expect that in large software systems pruning the features and goals from the optimization problem results in significant performance gains.

We can then use both exact and approximation methods of solving the above optimization problem. By representing each feature with a binary decision variable, we solve the optimization problem using well-known *Integer Programming Solvers* [71], which provision the optimal solution. Stochastic algorithms, such as *greedy* and *genetic* [58], that rely on FUSION specific heuristics can also be used for providing near-optimal solutions very fast.

7.3 Effect

Once the optimal feature selection is determined in *Plan*, *Effect* is invoked and provided with the optimal feature selection as a target configuration (Recall *A3* from Figure 2). In our previous work [119], the managed system is transitioned to the target configuration directly in one step. At the software architecture level, we compute the Architecture-Diff between current and target feature selections and devise a sequence of adaptation steps accordingly. In the general case, however, architecture-diffing may not be feasible in some cases, especially when features map to different layers of the system stack (e.g., software architecture, network, operating system, etc). For instance, consider a situation where an authentication feature in the software architecture has a dependency on a feature in the network architecture such as IPsec. To address such situation, Effect provides the option of devising a sequence of adaptation steps that enforces such dependencies at the feature level.

Figure 8 depicts the *Effect* heuristics based search algorithm for devising an adaptation path. In the remainder of this section, we describe the algorithm using an example. Since there are many possible paths to reach a target feature selection, the *Effect* algorithm is responsible for picking a path that does not result in a transition to an invalid feature selection. 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* activity enforces feature model constraints in every step.

More formally, Effect achieves this by devising a plan with several consecutive *adaptation steps*. 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 7.2, 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 consecutive adaptation steps an *adaptation path* π which transitions the current feature selection towards another feature selection: $\pi = \sigma_1 \sigma_2 \sigma_3 \ldots \sigma_{|\pi|} : (\sigma_i \in S)$.

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 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). Figure 7a shows that the current feature selection of the TRS system before adaptation is “1110111”, which means all the features expect F_4 (*Per-Session Auth.*) are currently enabled. In turn, we define each edge $e \in E$ to be a valid adaptation step that can transition one feature selection to the other ($e = v_{src} \xrightarrow{\sigma} v_{dst} : \sigma \in S$).Therefore, an adaptation path will correspond to a path in this graph: $v_{src} \xrightarrow{\pi} v_{dst} = v_{src} \xrightarrow{\sigma_1} v_1 \xrightarrow{\sigma_2} v_2 \xrightarrow{\sigma_3} v_3 \dots \xrightarrow{\sigma_{|\pi|}} v_{dst}$.

To search through the graph we developed the *Effect* algorithm following the A* search technique [58] which is a heuristic-based graph search approach. A* based search requires an admissible heuristic for the estimation of the distance from each vertex v_i to the target. We discovered that the binary representation of the current and target feature selection can be used to compute a feature-based admissible heuristic, which we refer to as *Feature Diff*. Feature Diff can be obtained by applying bitwise XOR operator on the binary representation of v_i and target and simply counting the number of bits that have the value of 1 as follows: $\boldsymbol{count\_ones(v_i \oplus v_{target})}$. For example, Feature Diff between a configuration “1110111” and a target configuration “0010111” would be $count\_ones(\text{"1110111"} \oplus \text{"0010111"}) = count\_ones(\text{"1100000"}) = 2$.

Effect uses this heuristic combined with a minimum step cost function to determine the order of exploration of neighbor vertexes in the graph. Calculating the minimum step cost is as simple as finding the minimum edge weight ($minw$) in the graph and is calculated once for the feature space graph. So, the heuristic function can be defined as:

$$h(v_i) = minw * count\_ones(v_i \oplus v_{target}).$$

Note that in *Effect* we are looking for a path (π) that goes from the vertex corresponding to the current configuration of the system to the vertex corresponding to the target configuration while minimizing a total cost function. We consider two types of costs in our formulation:

1. *Adaptation Overhead*: corresponds to the level of interruption as a result of, for example, disruption in the system's components, temporal delay in moving data between the servers, and so on. We model this cost by assigning weights to edges in the feature space graph. The weight of an edge depends on the source node and the adaptation step taken at the edge, since adaptation steps can have different costs when they are applied in different states. We formalize the weights as $w: (V \otimes S) \longrightarrow \mathbb{R}_{+}$. In turn, we define the cost of a path to be the accumulative cost of its steps:

   $$g(\pi) = \sum_{i=1}^{|\pi|} w(v_{i-1}, \sigma_i) \text{ where } v_0 = v_{src}.$$

2. *Utility Loss*: corresponds to the level of violation of system goals incurred. Based on the knowledge we obtained in the learning cycle (recall Table 2), we can calculate utility loss as: $y(v_i) = |\sum_{\forall g \in Violated\ Goals} U(v_i) \quad |$, where *Violated Goals* is the set of goals that will have 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. In turn, we define the utility loss of a path to be the utility loss of the worst adaptation step:

$$Y(\pi) = \max_{0 \le i \le |\pi|} y(v_i) \qquad \text{where } v_0 = v_{src}.$$

Figure 7 depicts the process of finding the adaptation path for disabling the first two features in TRS system where the target feature selection is “0010111”. Figure 7a depicts the current feature selection of the system, we used underline to indicate the bits that are different from the target feature selection (i.e., the first two bits from the left). *Effect* keeps track of search scope by a priority queue, in Figure 7 we indicate the members of this queue with an oval with a bolded perimeter.

In each iteration, a vertex, which is indicated by black oval in Figure 7, is extracted from the queue and the vertexes neighbor to it (i.e., reachable with one valid adaptation step) are added to the queue (and if already in the queue will be updated). If any of the recently added vertexes to the queue is the target the *Effect* algorithm stops as the target is reached. Finally, the extracted vertex is added to the visited list. We keep the record of the visited vertexes so we can build the path starting from the initial vertex (i.e., the current feature selection) to each vertex v in the queue; we call it partial path for that vertex and depict it as $\pi_p(v)$.

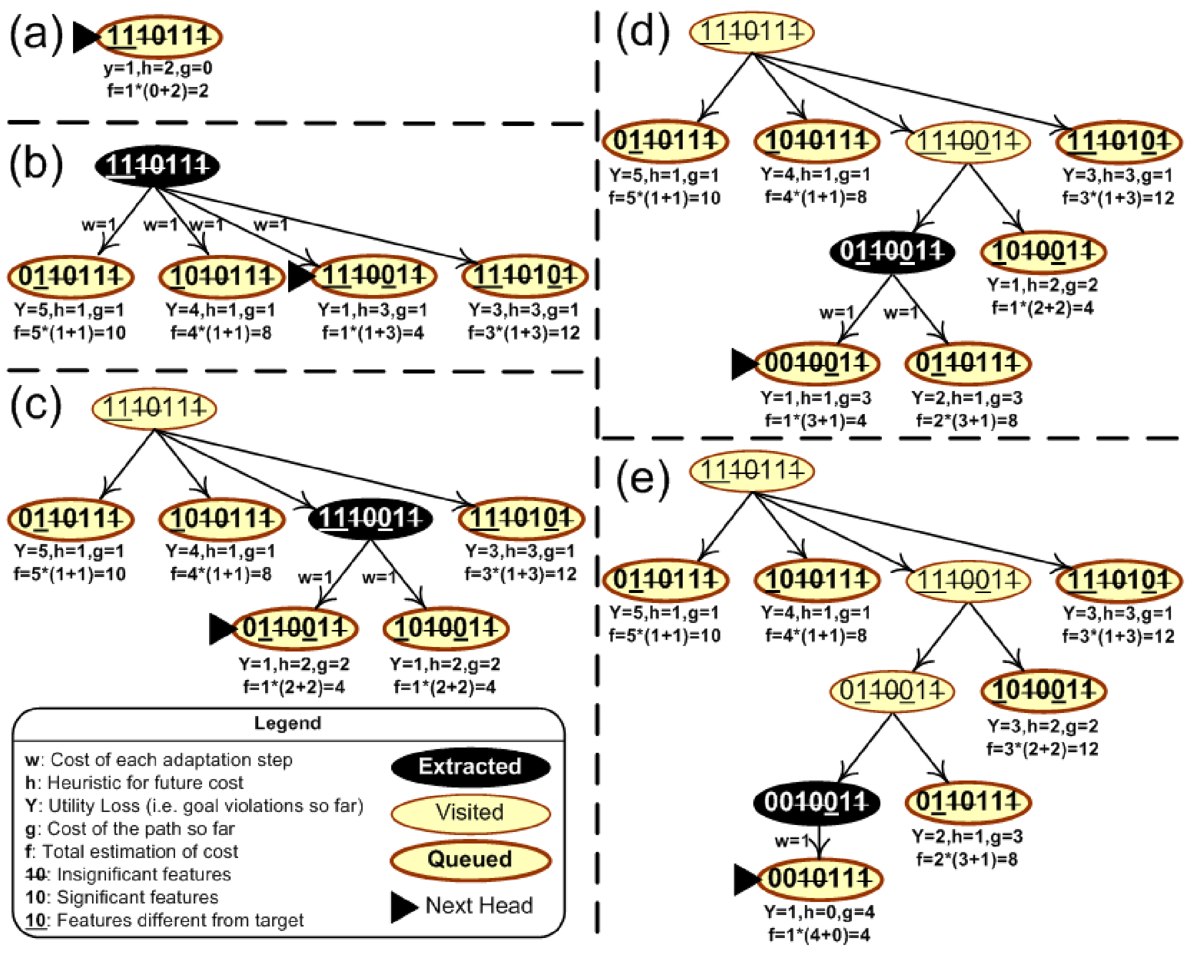


Figure 7: Effect finds shortest path within the significant feature space.

As indicated in Figure 7a the process starts with the current feature selection of the system as the only item in the queue. Therefore, in the next step (Figure 7b) that only vertex is extracted from the queue and its neighbors are added to the queue. *Effect* algorithm creates the graph gradually (directed by the heuristic) as it progresses and does not assume a complete graph to be present. On top of this, given the *Conflicting Goals*, we use Table 2 to eliminate insignificant features; (i.e., pruning them, since they are not going to have a significant impact on *Conflicting Goals*). This drastically decreases the neighborhood size of each vertex as the insignificant features are generally more than the significant ones. For example, assuming the situation shown in Figure 7 we are concerned about G_2 and G_3 (i.e., *Agent Reliability* and *Quote Quality*), then we can prune F_3, F_4 and F_7 (i.e., *Per-Request Auth.*, *Per-Session Authentication*, and *Semantic*) as they do not have significant effect on the goal. Using the heuristic in exploring the graph and avoiding the insignificant features helps us to reduce the search space significantly and makes *Effect* applicable and efficient for finding adaptation paths in systems with large feature spaces. Insignificant features in Figure 7 are indicated using a strikethrough line.

A priority queue is used in FUSION's Effect algorithm to sort its items and extracts them in order. The priority queue in *Effect* extracts vertex which has the least estimated total cost, defined as $f(v) = Y(\pi) * [g(\pi) + h(v_i)]$. The first item in this order is always the head of the priority queue. In other words, *Effect* gives priority to finding a path that does not degrade the performance of the system For instance, Figure 7b shows two vertexes that can potentially produce the lowest path length. However, since they violate one goal and since there are other vertexes that satisfy all goals, they are not explored first (Figure 7c). In Figure 7 we have provided all the required values (i.e., Y, h, g, and f) for each queued vertex. These values are calculated when the vertex is added (updated) in the queue.

When the number of goal violations and estimated cost for two vertexes in the queue are equal, we break the tie by choosing the one that has smaller $h(v)$ assuming it is closer to the target. For example in Figure 7d between the two items that have total estimated cost of 4 the one with less heuristic value (i.e., $h = 1$) is selected to be the next head. If there are two items that have the same $h(v)$ value, we select between them non-deterministically. For instance, in Figure 7c, the algorithm selects the vertex with less heuristic value until the target is added to the priority queue which by our definition will become the head of the queue.

Note that, in some cases, adding utility loss to the formulation increases the number of steps in the path. These extra steps represent exploration (*information collection*) that helps with keeping the knowledge base up-to-date and potentially improving the learning cycle.

```
function Effect(node src, node tgt)

        g_score[src] := 0
        h_score[src] := h(src, tgt)
        y_score[src] := Y(src)
        f_score[src] := y_score[src] * ( g_score[src] + h_score[src] )

        closed_set := the empty set
        open_set := matrix containing start node
        came_from := the empty map
        significant_features := significant_feature_space()

        while openset is not empty

                x := the node in openset having lowest f_score[] value

                if x = tgt
                        return reconstruct_path(came_from, came_from[tgt])

                remove x from open_set
                add x to closed_set
                neighbor_nodes = expand(x, significant_features);

                foreach v in neighbor_nodes
                        if v in closed_set
                                continue;

                        if v not in open_set
                                add v to open_set

                        came_from[v] := x
                        g_score[v] := g_score[x] + w(x,v)
                        h_score[v] := h(v, tgt)
                        y_score[v] := y(v)
                        f_score[v] := y_score[v] * ( g_score[v] + h_score[v] )

        return failure

function reconstruct_path(came_from, cur_node)
        if came_from[cur_node] is not null
                p = reconstruct_path(came_from, came_from[cur_node])
                return(p + current_node)
        else
                return current_node
```

Figure 8. *Effect* heuristics-based path search algorithm.

```
function h(node v, node tgt) //computes heuristic for a given node v

        diff := to_binary(v) ⊕ to_binary(tgt)

        return minw * count_ones(diff)
```

```
function Y(node v)     //computes utility loss for a given node v

        conflicting_goals := set of <metric, utility> tuples of the goals from Plan
        tentative_metric := 0;
        tentative_utility := 0;
        utility_loss := 0;

        for each metric, utility in conflicting_goals
                tentative_metric := metric(context, v);      //learned function
                tentative_utility := utility(tentative_metric);
                if tentative_utility < 0
                        utility_loss += (-1 * tentative_utility);

        return utility_loss
```

```
function significant_feature_space()

        conflicting_goals := set of goals from Plan as <metric, utility> tuples
        induced_functions := set of all <metric, feature, coef> tuples in KB
        relations := set of all <child, parent> feature relationship tuples in KB
        neighbor_nodes_set := the empty set

        //select features to flip from the set of (significant features)
        features_to_flip := select feature from induced_functions
                            where coef is not null and
                            metric in (select mwetric from oonflicting_goals)
        features_to_flip += select parent from relations
                            where child in features_to_flip

        return features_to_flip
```

Figure 9. *Effect* helper methods (heuristics, utility loss, and significant feature space functions).

Chapter 8: **FUSION Implementation**

Figure 10 shows snapshots of a prototype implementation of FUSION. This figure closely matches the structure of Figure 3, and illustrates the realization of the modeling concepts in FUSION using the 3-layer-model reference architecture in [33], which distinguishes between three kinds of concerns in self-adaptation logic: *Goal Management*, *Change Management*, and *Component Control* (Recall from Section 2.3).

Each part in Figure 10 is developed using a dedicated editor as will be detailed in the following subsections. As mentioned in section 5.1, 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 10b is realized using the architectural fragment *AddHoc Reports* in Figure 10c, which extends TRS's base architecture.

Note that self-adaptation logic crosscuts FUSION (i.e., in Figure 10a and Figure 10b) to reach the architecture in Figure 10c and eventually to the running system. In other words, self-adaptation logic resides at different levels of abstraction, and thus make it impractical to use a single modeling language.

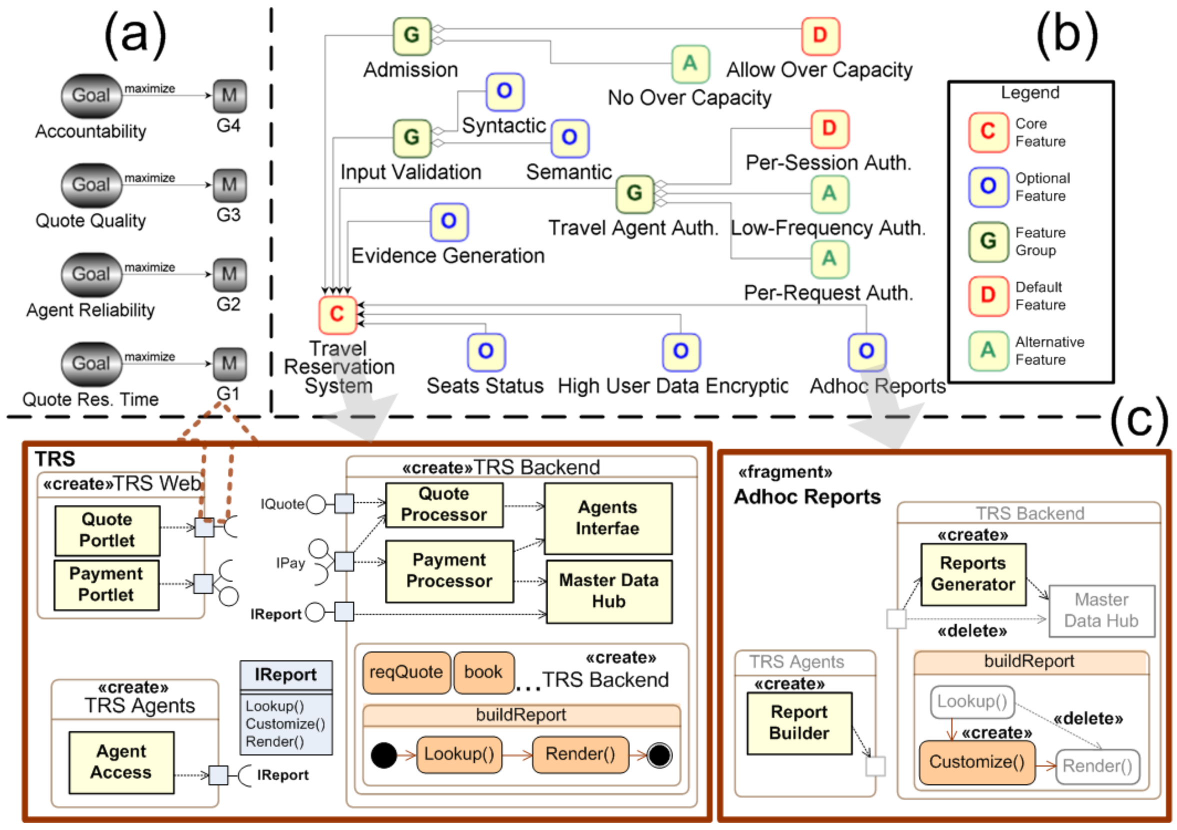


Figure 10. Subset of TRS in our prototype implementation of FUSION: (a) goals and metrics, (b) feature model, (c) implementations of Core and *Caching* features.

In this chapter, we present a tool support framework for self-adaptation that aligns self-adaptation logic as elicited in the 3-layer-architecture [33] to an Online Model Driven Architecture (Online MDA). The framework, depicted in Figure 11, assigns an additional role to MDA that manifests itself at runtime. In particular, models and transformations are used at runtime to realize a hierarchical control loop. For example, enabling the *AdHoc Reports* feature in Figure 10b triggers a transformation rule that

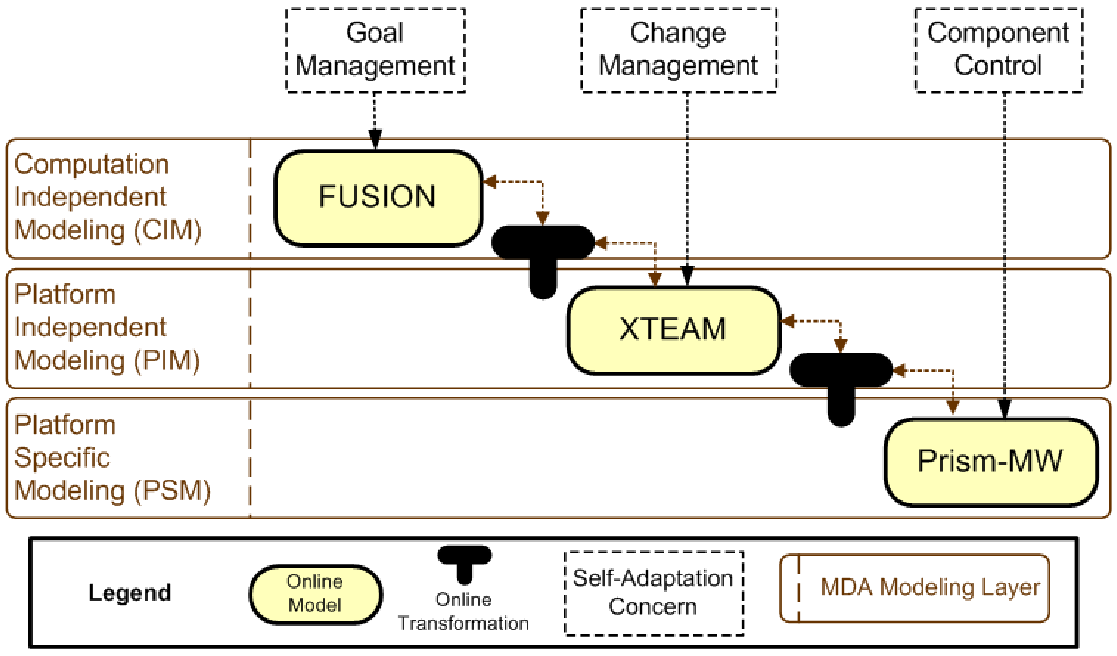


Figure 11. Self-Adaptation tool support using Online MDA.

effects the corresponding *AdHoc Reports* architectural fragment in Figure 10c, which then deploys the corresponding components and connectors into the running system.

The following subsections describe the implementation framework in details. Section 8.1 describes self-adaptation models depicted in Figure 10 along with their meta-models (i.e., FUSION, XTEAM, and Prism-MW in Figure 11). It also describes how changes in each model are realized by an underlying implementation platform that is kept in sync with the model. Section 8.2 describes how the models connect and propagate changes among one another through transformation rules that are defined at the meta-model level (i.e., Online Transformations in Figure 11). Section 8.3 describes a component of the framework that aims to facilitate incremental training of FUSION's knowledge base since it adds overhead to the process of implementing self-adaptive software systems.

8.1 Modeling Self-Adaptation Logic

The following subsections provide an overview of the individual components involved in implementing self-adaptation logic for each concern in conformance with the 3-layered-model.

8.1.1 FUSION-Based Goal Management

We utilize FUSION as a Goal Management platform. Figure 12 depicts a portion of the meta-model of FUSION. The corresponding concrete syntax is depicted in Figure 10a for the TRS system. The FUSION modeling tool provides three editors for: 1) feature modeling, 2) goal modeling, and 3) contextual variables. The feature model editor (depicted in Figure 10b) produces MOF compliant feature models following the approach in [22] that supports creation of *Features* and *Feature Group* model elements. Each Feature is associated with an attribute, which is used to capture runtime state (i.e., enabled=”1”, disabled=”0”). For example, when FUSION decides to disable feature *F1*, it updates the value of the *selected* attribute to “0”. Features can be of any of the following types:

- *Core*: must be enabled all the time as long as the system is running,
- *Optional*: can be enabled/disabled at runtime, and

- *Alternative*: can be enabled only when all features under mutual exclusive relationship are disabled.
- *Default*: is a special kind of Alternative feature that is enabled by default at system starts up time.

Feature Group model elements are used to capture the following feature interrelationships:

- *Exactly-one-of-group*: captures a mutual exclusion relationship among all features associated with it where one-and-only-one feature must be enabled.
- *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.
- *At-least-one-of-group*: captures a mandatory variation at which multiple features may be enabled.
- *Zero-or-all-of-group*: captures a mutual inclusion relationship, that is either all features are enabled or none. The features also apply to an optional variation point.

In addition, a number of consistency rules apply to the feature model both at design time and runtime. Below are some examples of design time consistency rules that apply to the feature model:

- Core feature cannot be dependent 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.

At runtime, a number of rules ensure consistency of feature selections and enforcement of Feature Group semantics (i.e., Feature relationships) such as the constraints listed in Table 3. The Goal Model editor (depicted in Figure 10a) supports two types of model elements. The first type of model elements is the *Goals*, which can be of a number of subtypes based on a pre-defined utility function template (i.e., Linear, Step-Function, Exponential, and Sigmoid). The second type of model elements is *Metrics*, which can be of the following two sub-types: *Continuous*, which allows for any

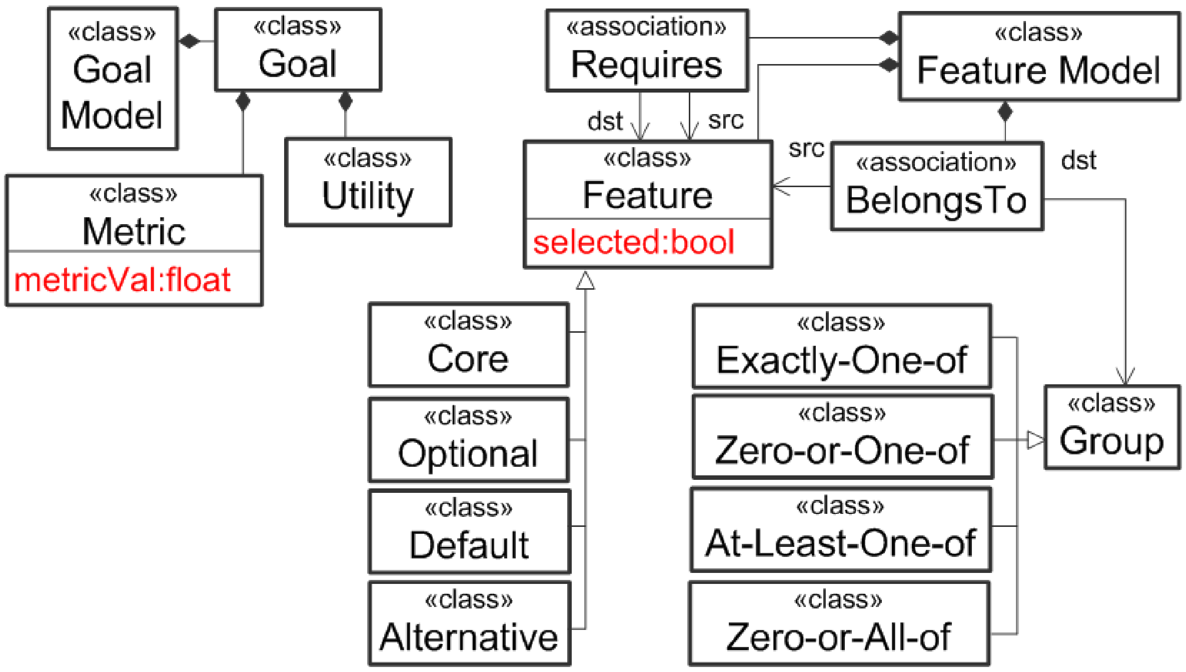


Figure 12. FUSION-Based Goal Management meta-model.

rational number to populate its value, and *Discrete*, which requires a predefined array of valid values.

FUSION underlying platform is kept synchronized with an online version of the FUSION Model, which includes the Feature Model, Goal Model, and Contextual Variables, in memory and updates it when runtime changes occur such as it makes new feature selections or receiving a metric reading from the running system in order to keep track of runtime state changes. For instance, the *metricVal* attribute on the Metric class is updated at runtime with the most recent read value. Section 8.2.1 describes how metric readings are kept up-to-date.

The runtime platform also interacts with a number of tools. For instance, we use WEKA API [91], which provides an open source implementation of a number of learning algorithms [92] leveraged in our work, to realize the Induce activity. The outputs of Induce activity (i.e., FUSION's Knowledge Base) as well as all observations provided as input are stored in a relational database.

8.1.2 Architecture-Based Change Management

We have provided support for Change Management by extending XTEAM, an architectural modeling and analysis environment developed in [14] for the representation of system's architecture. XTEAM supports modeling of a system's software architecture using heterogeneous Architectural Description Languages (ADLs) [46], where each ADL is suitable for representing a particular concern. For instance, XTEAM supports Finite State Processes (FSP) [37] and eXtensible Architectural Description Language

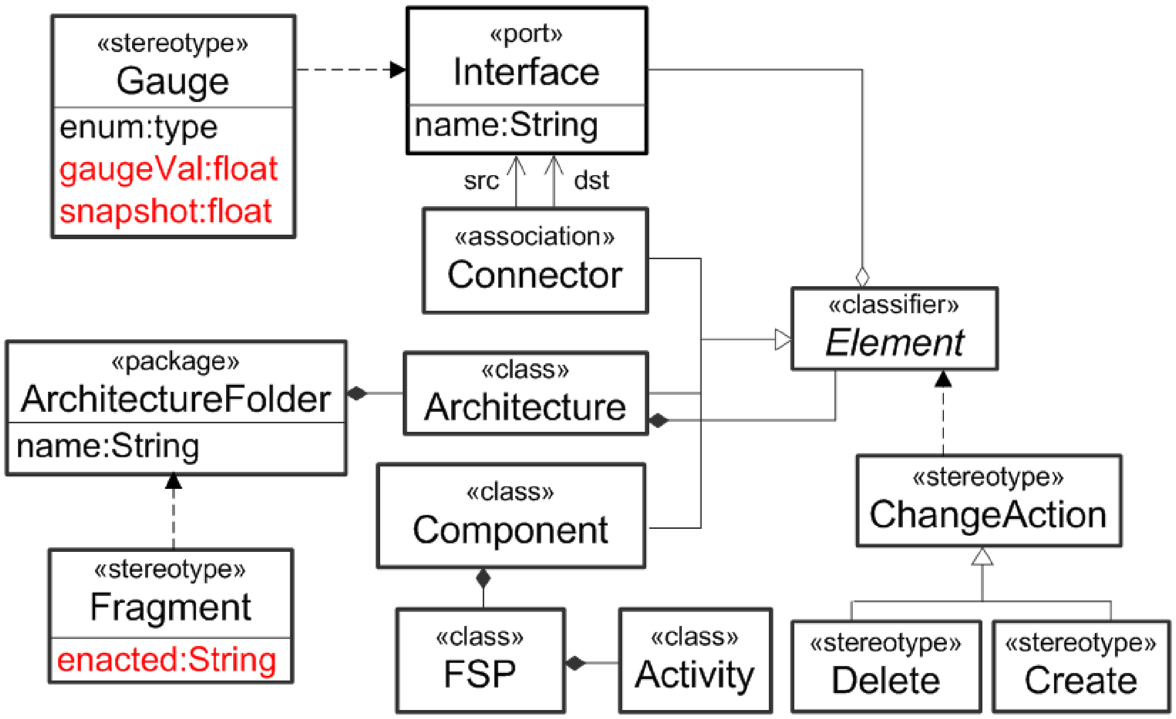


Figure 13. XTEAM-Based Change Management meta-model.

(xADL) [13] for modeling the behavioral and structural properties of a software system, respectively. (see Figure 13). We have modified the notation of XTEAM in Figure 10c and the underlying meta-model in Figure 13 to simplify the framework. However, we reused the underlying dynamic QoS analysis facilities.

A snapshot of xADL model for a subset of TRS is shown in Figure 10c. The models are organized in *ArchitectureFolders*. Each folder may contain one or more *Architecture*s. The internal structure of the architecture is defined interms of *Components* (i.e., distributed software units) which interact using *Connectors* (i.e., communication lines). For example, Figure 10c depicts the *TRS* ArchitectureFolder, which contains three Architectures: *TRS Web*, *TRS Agents*, and *TRS Backend*. The TRS Backend architecture has three interfaces: *IQuote*, *IPay*, 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. Components also expose interfaces but we do not show them in Figure 10c to reduce cluster. In addition, the *TRS Backend* architecture contains an FSP that supports a number of activities where each activity processes a message incoming for an interface. For example, *Build Report* activity processes messages coming from the *IReport* interface.

We extended XTEAM to provide means for change management too. A *Fragment*, for instance, is a snippet of Components and Connectors 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 as variation points to specify the insertion points for the changes. For example, Figure 10c shows the impact of weaving the *Adhoc Reports* fragment on the core architectural model, i.e., it results in the addition of *Report Builder* and *Report Generator* components, some connector modifications, and the new *Customize()* activity in the FSP of *TRS Backend* architecture. Hence, the stereotypes <<create>> and <<delete>> represent *ChangeActions* depicted in Figure 13.

We also extended XTEAM to capture runtime state information. For instance, the *enacted* attribute of a Fragment (marked red in Figure 13) identifies if the fragment is weaved into the base architecture (i.e., 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 10c highlights the *Quote response time* interface, which acts as a *Gauge* that collects roundtrip time for a message to go throughout the architecture and return to that interface. The gauge captures roundtrip response time using the *gaugeVal* attribute depicted in Figure 13.

8.1.3 Prism-MW Based Component Control

We provide support for Component Control using Prism MW [40]. Typically, a software system operates on a number of platforms. For instance it may use a platform for implementing the internals of a software component. It can then use another platform for implementing the interfacing logic (i.e., using Web Services). We focused our experimentation on Prism-MW to simplify the framework.

Prism-MW is an *architectural middleware*, which supports architectural abstractions by providing implementation-level classes for representing each component and connector (i.e., *Bricks*), with operations for creating, manipulating, and destroying instances of the bricks. A subset of the meta-model of Prism-MW is depicted in Figure 14. These abstractions enable direct mapping between a system's software architectural model and its implementation. Prism-MW provides three key capabilities that we have relied on to implement the component control layer. It provides support for (1) basic architecture-level dynamism, (2) multiple architectural styles, and (3) architectural reflection.

On top of Prism, we created an online Platform Specific Model (PSM) that conforms to the meta-model depicted in Figure 14. The model also keeps track of two key dynamic state variables: *deployed* and *probVal*.

At design time, code generation from the PSM model takes place following the approach introduced in [83]. For instance, applying the <<create>> stereotype to *Report Builder* component in the *AdHoc Reports* fragments corresponds to generating a command in Prism-MW that adds the component to the *TRS Agent*. Figure 15 depicts a snapshot of the generated Prism-MW code for the *AdHoc Reports* architectural fragment depicted in Figure 10c. The code generation follows the rules listed in Table 4.

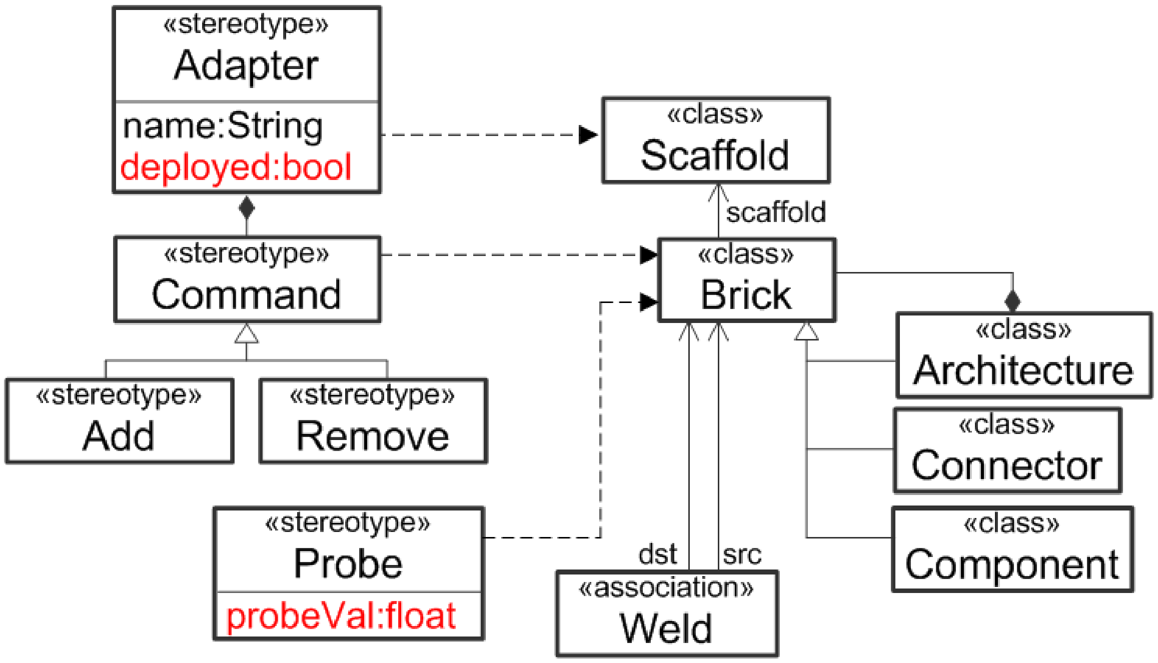


Figure 14. Component control meta-model extends Prism-MW.

```
   public class AdHocReports extends Adapter{

      public static boolean deploy(){

         Component ReportBuilder = new Component(ReportBuilderImpl);
         TRSAgents.add(ReportBuilder);
         TRSAgents.weld(ReportBuilder, IQuote);

         Component ReportGenerator = new Component(ReportGenerator);
         TRSBackend.add(ReportGenerator);
         TRSBackend.unweld(IQuote , MasterDataHub);
         TRSBackend.weld(IQuote , ReportGenerator);
      }

      public static boolean undeploy(){

         TRSAgents.unweld(ReportBuilder, IQuote);
         TRSAgents.remove(ReportBuilder);

         TRSBackend.unweld(IQuote , ReportGenerator);
         TRSBackend.remove(ReportGenerator);
      }
}
```

Figure 15. *AdHoc Report* adapter code in Prism-MW.

At runtime, the PSM model is synced with the underlying Prism-MW platform. A change in the PSM model results in the execution of code. For instance, setting the deployed attribute of the Adapter *AdHoc Reports* to "0" results in executing undeploy() method in Figure 15.

Prism-MW has been enhanced in [118] to support *adaptation patterns* [23]. When the adaptation steps are carried out, Prism-MW uses *adaptation patterns* to put dependent components (*passive set*) into *passive* state to be able to provide quiescence property for the component that is being changed. This ensures the consistency of the system after each adaptation step and in turn after the whole adaptation. To infer the dependencies and construct *passive set*, we use the rules and constraints of an architectural style. This inference is valid for any software system built according to a particular style. For each style, we used the dependency rules to determine a reusable sequence of changes that need to occur for placing a node in quiescence. Such a recurring sequence of changes is called an *adaptation pattern*. An adaptation pattern provides a template for making changes to a software system built according to a given style without jeopardizing its consistency.

Figure 16 depicts a state machine of a subset of such a pattern for C2 style describing C2 component that is being changed. The state machine suggests that for a C2 component to go from an *Active* to *Passive*, first it should go through *Passivating Dependents* (nodes below it) and then it should pass the *Passivating Itself* step. For instance, to add connector between *Reports Generator* and *Master Data Hub* in Figure 10c. As a result, dependent components and connectors (i.e., *Report Generator*) must be passivated first then the connector passivates itself.

8.2 Connecting Self-Adaptation Models Together

This section provides an overview of the transformations that are used to connect self-adaptation models together. Transformations are defined at the meta-model level using QVT-Relations. Interested reader may refer to [117] for full specification of the

QVT Relations language. We provide a brief description of the key constructs that will be used in the following two subsections.

QVT-Relations 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 pattern and a target pattern). 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.

8.2.1 Mapping FUSION to XTEAM

Our experimental environment uses FUSION as the goal management model and XTEAM as the change management model. As such, we defined a transformation bridge

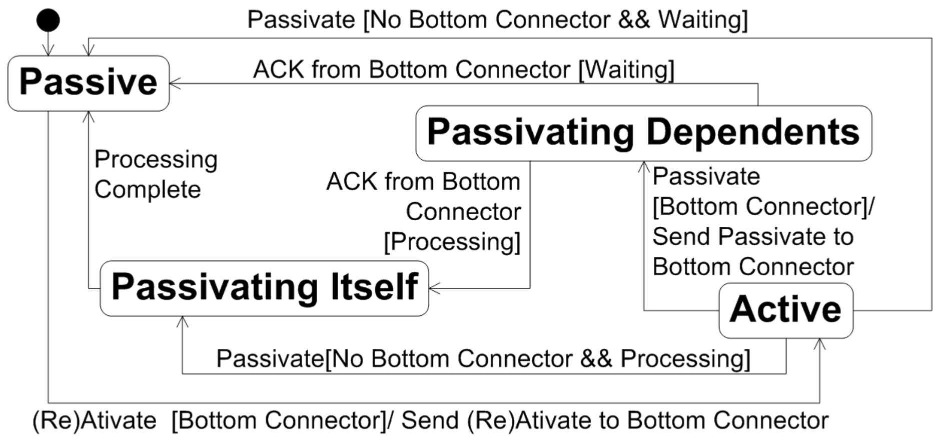

Passivate [No Bottom Connector && Waiting]
Passive
ACK from Bottom Connector [Waiting]
Passivating Dependents
Processing Complete
ACK from Bottom Connector [Processing]
Passivate [Bottom Connector]/ Send Passivate to Bottom Connector
Passivating Itself
Active
Passivate[No Bottom Connector && Processing]
(Re)Ativate [Bottom Connector]/ Send (Re)Ativate to Bottom Connector

Figure 16. The state charts of C2 adaptation pattern for adapting a C2Component.

(i.e., *FUSION-to-XTEAM*) that maps goal management concepts to change management concepts in the two meta-models. For example, in Figure 17, the *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 10b maps to the *AdHoc Reports* fragment in Figure 10c 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 10a) to a Gauge in the XTEAM models if the names match.

FUSION-to-XTEAM defines two additional relations (i.e., *adapt* and *monitor*) that are related to self-adaptation since they operate on model elements that represent runtime state. For example, the *adapt* relation in Figure 17 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 (i.e. the **when** operator) to verify that their owners also correspond to each other (i.e., the owning feature matches the owning fragment). Otherwise, the QVT-Relations rule engine will make all *selected* tagged values match the values of all *enacted* tagged values. QVT allows the use of chaining of relations so that sub-relations can be enforced before super relations. In the case of *adapt* relation, it achieves this by using *FeatureToFragment* relation as a super-relation.

Note that *adapt* is defined as an incremental transformation; that is, it is triggered to execute dynamically whenever the *selected* attribute of a feature is modified without affecting other parts of the models. As a result, the corresponding *enacted* attribute only in XTEAM will be updated to have the same value. For example, if FUSION decides to enable feature *AdHoc Reports* in Figure 10b, it modifies the attribute *selected* of that

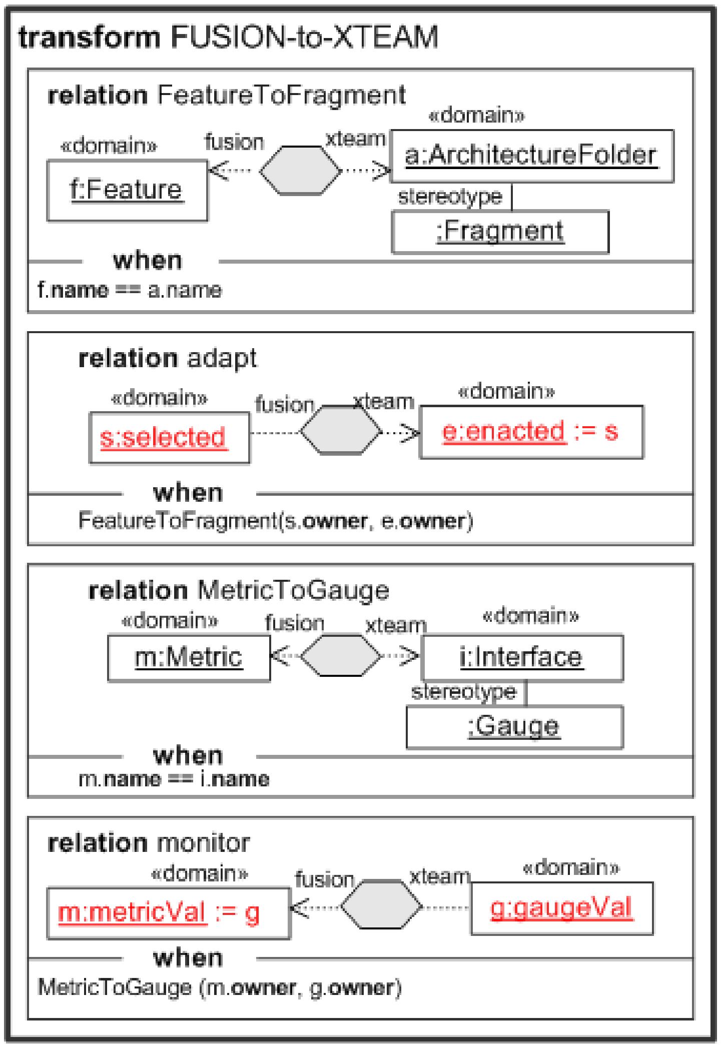


Figure 17. FUSION-to-XTEAM transformation definition.

feature to “1”. As a result, the corresponding enacted attribute of the *AdHoc Reports* fragment in Figure 10c will be modified to be “1”. Hence, a single execution occurrence of this rule means that FUSION is commanding XTEAM to effect the feature by changing the architecture (Notice the direction of the arrow in the definition of *adapt* relation indicating that this is a one-way transformation).

Similarly, the *monitor* relation in Figure 17 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. FUSION receives metric readings from XTEAM and not the other way around. The relation listens to modifications of *gaugeVal* in XTEAM so that it can propagate them to FUSION. The efficiency of such incremental transformations and their feasibility for use at runtime has be shown in [85].

8.2.2 Mapping XTEAM to PrismMW

The *XTEAM-to-PrismMW* transformation uses a number of relations to produce a platform specific model. Relations in this transformation (depicted in Figure 18 and Figure 19) are heavily chained together using the **when** construct to produce complex output. The first group of relations, which operate on design models created by the engineer in Figure 10c, capture relationships within the base architecture of the software system (Figure 18). They are chained as follows:

1. *FragmentToAdapter*: maps a *Fragment* in XTEAM to an *Adapter* in PrismMW. For example, the *AdHoc Reports* fragment maps to the *AdHoc Reports* adapter code depicted in Figure 15.
2. *XArchitectureToPArchitecture*: maps an XTEAM *Architecture* to a Prism-MW *Architecture*. For instance, *TRS Backend* architecture is mapped to *TRSBackend* architecture in Prism-MW. Note that Prism-MW treats the presence of an architecture that is not associated with the <<create>> operator as a reference to an existing architecture in the base system.
3. *XComponentToPComponent*: maps a *Component* in XTEAM to a *Component* in Prism-MW. For instance, the Report Generator is mapped to *ReportGenerator* Prism-MW component. Note that Report Generator is associated with the <<create>> operator, which will result in triggering the CreateToAdd relation described next. Once the CreateToAdd relation is executed, Prism-MW will treat this as an <<Add>> command and generate code for it as shown in the *deploy()* and *Undeploy()* methods of Figure 15.
4. *InterfaceToPort*: For a given architecture, this relation maps an *Interface* in XTEAM to a *Port* in Prism-MW. For instance, *TRSBackend* in *AdHoc Reports* has a reference to a port. The port reference is mapped to Prism-MW.
5. *XConnectorToPConnector*: maps a Connector in XTEAM to a Connector object in Prism MW. For instance, the link between Reports Generator and Master Data Hub components (Figure 10c) maps to a Prism-MW connector. Prism-MW generates code for it as in Figure 15.

The second group of relations maps elements that relate to adaptation logic (Figure 19). They are also chained with the first group of relations (i.e., the ones related to the base architecture) to ensure consistency of execution.

1. *CreateToAdd*: this relation passes a <<create>> stereotype down to Prism-MW as <<Add>> since Prim-MW code generator has been developed independently and recognizes it. The code generation is based on the rules depicted in Table 4. Since this particular stereotype applies to multiple kinds of model elements, the relation chains itself to a number of super-relations and connect them together with **or** to indicate that the execution context can be coming from any of these relations.
2. *DeleteToRemove*: similar to CreateToAdd, it passes the <<delete>> stereotype to Prism-MW, which then generates code internally according to Table 4.

The third group of relations (i.e., *adapt* and *monitor*) operate on runtime state information (marked red in Figure 13 and Figure 14) and therefore are related to runtime self-adaptation. The *adapt* relation in Figure 19 maps the *enacted* attribute in XTEAM to the *deployed* attribute in Prism-MW. To make sure that the attributes on both sides correspond to each other, a precondition is defined (i.e. the **when** operator) to verify that their owners also correspond to each other (i.e., the owning fragment matches the owning adapter). Similarly, the monitor relationship maps a probe with a snapshot in the XTEAM model. Whenever a new probe reading is available, this transformation kicks in to update the XTEAM model with the latest readings, which eventually propagates the readings upwards to FUSION.

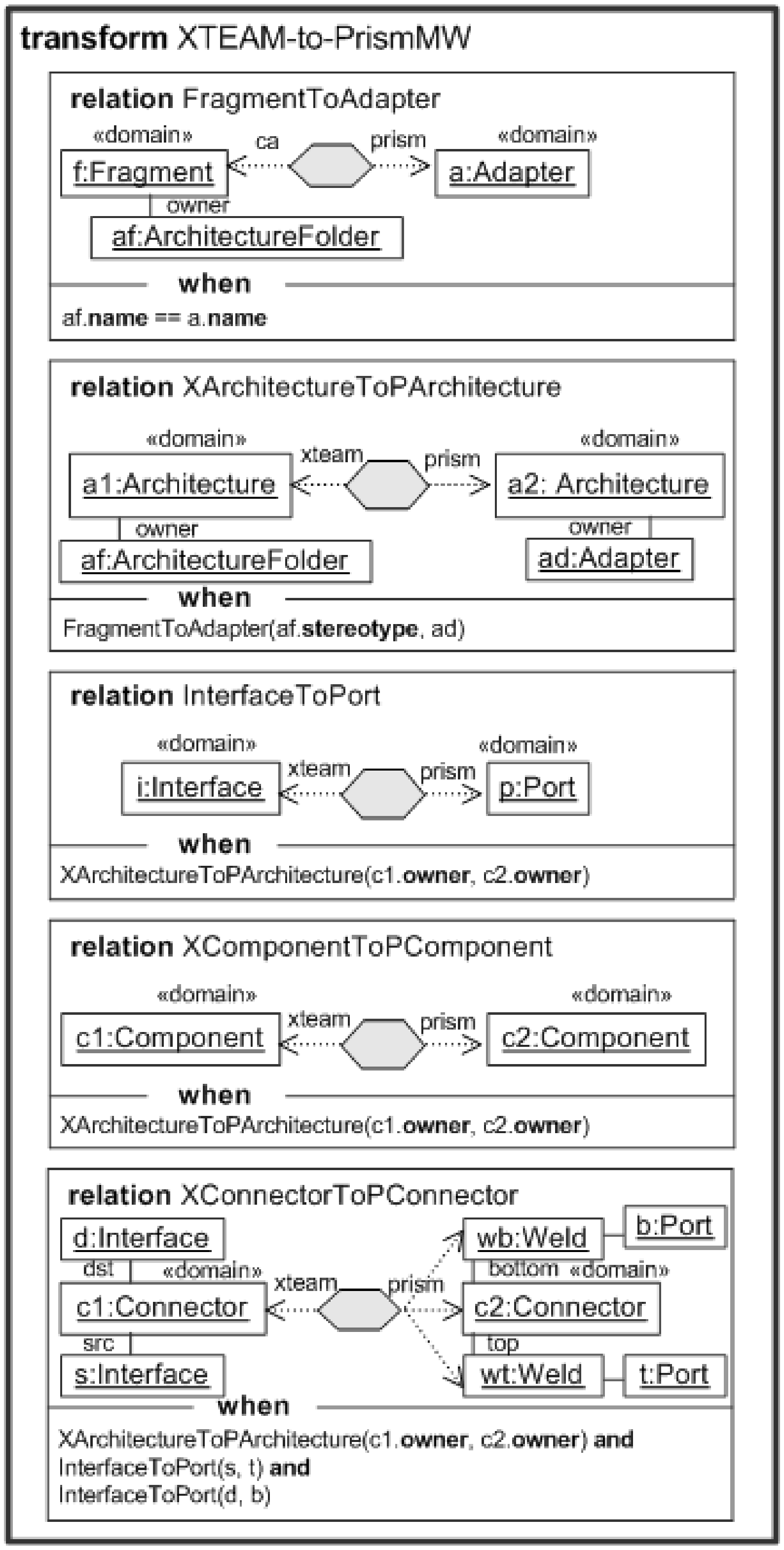


Figure 18. Mapping base architecture elements in XTEAM-to-PrismMW transformation.

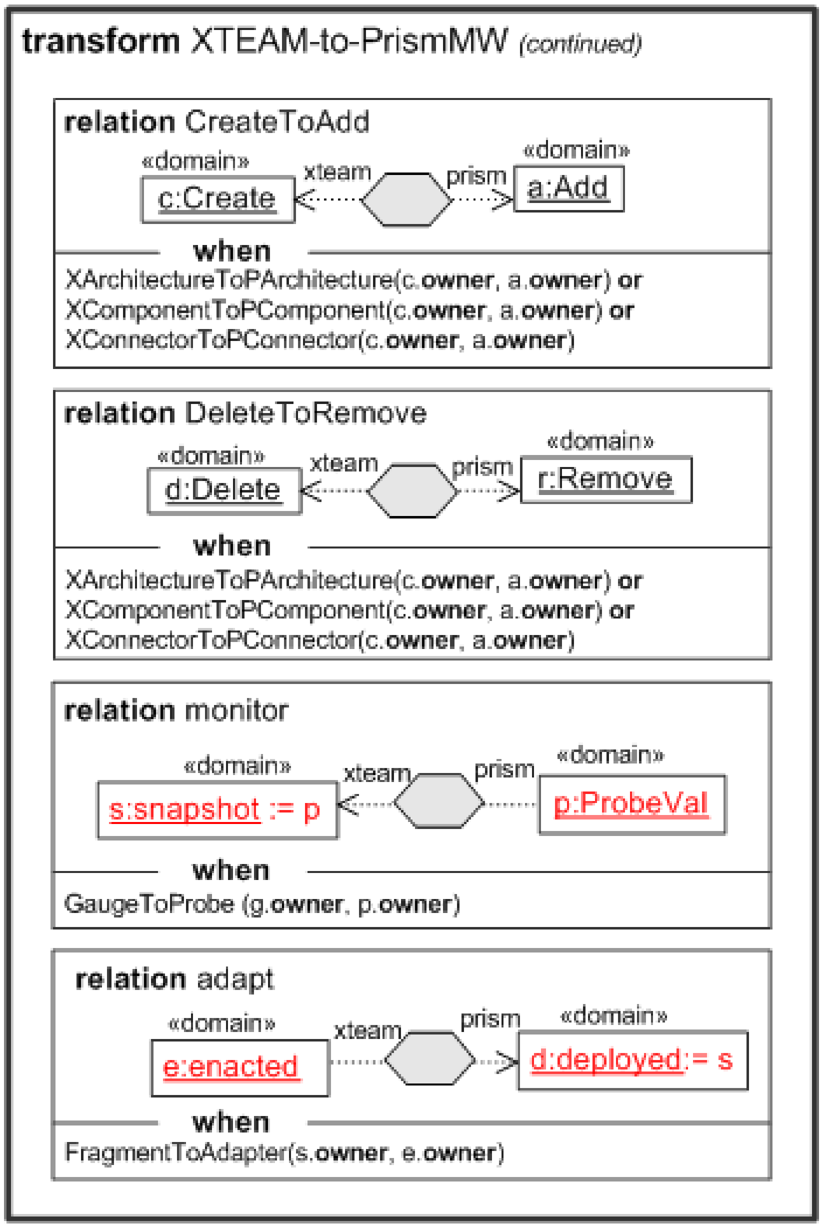

transform XTEAM-to-PrismMW (continued)
relation CreateToAdd
«domain»
c:Create
xteam
prism
«domain»
a:Add
when
XArchitectureToPArchitecture(c.owner, a.owner) or
XComponentToPComponent(c.owner, a.owner) or
XConnectorToPConnector(c.owner, a.owner)
relation DeleteToRemove
«domain»
d:Delete
xteam
prism
«domain»
r:Remove
when
XArchitectureToPArchitecture(c.owner, a.owner) or
XComponentToPComponent(c.owner, a.owner) or
XConnectorToPConnector(c.owner, a.owner)
relation monitor
«domain»
s:snapshot := p
xteam
prism
«domain»
p:ProbeVal
when
GaugeToProbe (g.owner, p.owner)
relation adapt
«domain»
e:enacted
xteam
prism
«domain»
d:deployed:= s
when
FragmentToAdapter(s.owner, e.owner)

Figure 19. Mapping adaptation related elements in XTEAM-to-PrismMW transformation.

8.3 Training the Knowledge Base

Tool support covered in the last two section (i.e., Sections 8.1 and 8.2) facilitates incremental design of complex self-adaptive systems by utilizing MDA best practices. However, when a black-box approach like FUSION is used, tool support for the extensive training process involved is essential. There are multiple sources of training data (e.g., design experimentation, existing runtime data of an older version of the system that is deployed at runtime, etc). Thus, a mechanism is needed to help identify how much of the knowledge available (i.e., whether from design experimentation or from older running versions) can be incorporated back into the design process.

In this section, we demonstrate how we used a key property of feature models to facilitate incremental training in our implementation. Figure 20 demonstrates the design

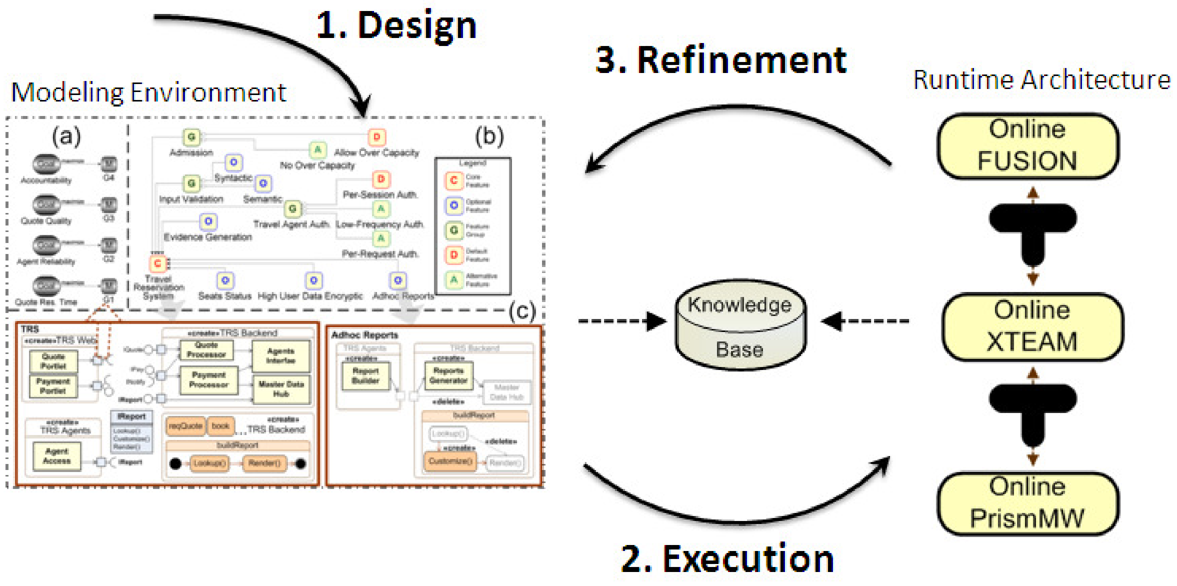


Figure 20. Development Lifecycle for TRS.

phase engineers go through:

1. **Design**: Engineers develop design models that define self-adaptation logic (i.e., using the modeling languages described in Section 8.18.2).
2. **Execution**: Models are executed (i.e., using the tools described in Section 8.2) to explore the configuration space and identify key variables to expose to FUSION model level. The final outcomes of the execution activities is stored in FUSION KB as training data.
3. **Refinement**: Engineers may change the feature model by adding, removing, or modifying features in the feature-model. This has implications on the data stored in the Knowledge Based of FUSION. Depending on the nature of refinements in the design, the system may need to be retained accordingly.

The cycle between Execution and Refinement is where training data is generated. As for the Execution phase, the focus of the training process is on addressing new feature selections that were not supported in past versions of the system.

In the Refinement phase however, we found it difficult to know if the available training data is usable without proper identification of the impact of refinements on existing the feature selection space. We used a key property of feature models that helps with identifying the impact of refinements on the knowledge base so that training can also take place incrementally. That is, as engineers change the feature model, FUSION applies refinement operators to identify obsolete data (Recall Table 1) and eliminate them automatically while retaining the rest of the observations in the KB. The refinement operators are listed in Table 5. Whenever a new version of the feature model is created,

Table 5. FUSION KB refinement guided by feature models.

Feature Model Refinement		KB Refinement Procedure
Type of Change	**Relationship**	
Backward compatible	$S_{old} - S_{new} = \phi$	No KB refinements required
Unsupported feature selections	$S_{old} - S_{new} = S_{unsupported}$	Remove $S_{unsupported}$ observations
Obsolete Feature Selections	$S_{old} - S_{modified} = S_{obsolete}$	Remove $S_{obsolete}$ observations

FUSION applies the following intuitive rule: "A new feature model is an increment of an old feature model if and only if $S_{old} \subseteq S_{new}$, where S is the feature selection space".

Depending on the relationship between the old and new feature models, the refinement process differs. The first possibility of the relation between old and new feature model is to be *backward compatible*. That is, the new feature model supports all of the feature selections that used to be supported in the older version. In this case, no refinement is needed.

The second possibility is to have some *unsupported feature selections*. This can potentially happen if the engineers decide to either eliminate features completely or add new dependencies among features. In these cases, some portions of the old feature selection space become unsupported in the new system. As a result, the corresponding observations must be removed from the KB according to the rules in Table 5.

The third possibility is for the engineers to modify the internal implementation of a feature without making changes to the feature model. In this case, modified features will be treated as if they were deleted and new ones were added since they do not anymore exhibit the same behavior they used to have and therefore FUSION's knowledge of their behavior is obsolete. As shown in Table 5, FUSION simply removes observations that correspond to the *obsolete feature selections* and reruns Induce.

Note that the refinement operators mentioned do not guarantee accuracy of the knowledge obtained. For example, infrastructure upgrades can change the magnitude of impact of a feature without any design changes to it. However, such data can be used to produce first cut knowledge of the system. Once these refinement operators are applied, engineers train the system on the new feature selections that are supported by the new feature model.

Chapter 9: Evaluation

In this chapter, we provide an evaluation of FUSION using a travel reservation system to test and compare the learning and adaptation cycles in terms of accuracy and efficiency empirically. The research hypotheses were used to guide the evaluation

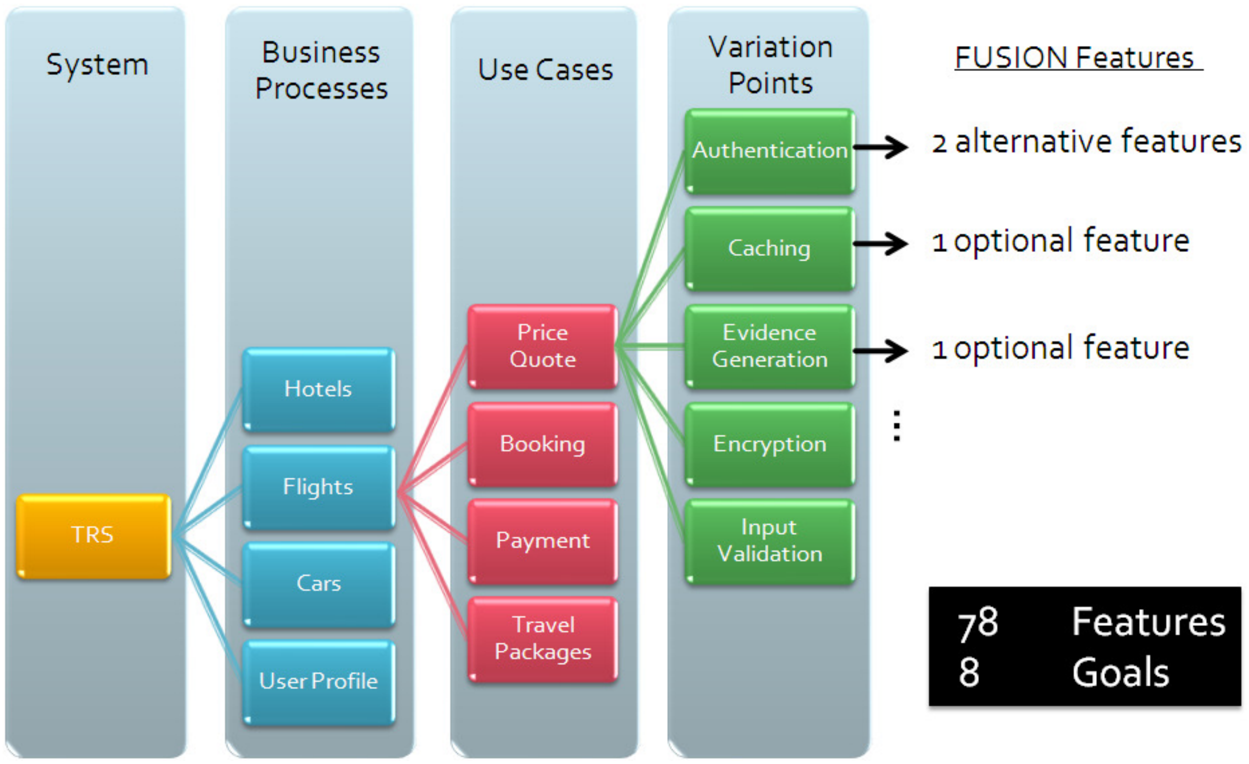


Figure 21. Elicitation of adaptation choices (i.e., Features) in TRS.

process and determine needed validation exercises.

We demonstrate in the following subsections empirical results obtained from an extended version of TRS, which consisted of 78 features and 8 goals. Figure 21 depicts a portion of the full 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 price quotations to the customer.
2. Booking: allows the user to select one of the quotes that are obtained from the previous use case and then proceed with selecting detailed preferences. In some cases, the preferences 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 a user profile that is maintained for each customer to determine future discounts and travel packages.
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.

Engineers of each QoS dimension (e.g., security, performance, forensics, reliability, etc) analyze each use case and identify key adaptation choices that have significant impact on the system's goals based on stakeholder priorities.

To evaluate FUSION's ability to learn and adapt under a variety of conditions, we set up a controlled environment. We developed a prototype of the implementation environment described in Chapter 8. We used XTEAM to simulate the execution context of the software (e.g., workload) as well as the occurrence of unexpected events (e.g., database indexing failure). However, note that neither the TRS software nor FUSION was controlled, which allowed them to behave as they would in practice. FUSION was 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 price quote requests.

(VT) Varying context—the system is placed under a workload fluctuation setting 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 3) unexpectedly fails, and forces a full table scans for some tables.

(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).

Unless otherwise is stated, the experimental scenarios and all associated evaluation exercises have been run 30 times and 95% confidence intervals have been computed and shown in the figures.

9.1 Hypothesis #1: Concept Drifts

In this section, we demonstrate the ability of FUSION to learn new emerging patterns of behavior at runtime and the impact of concept drifts on the quality of adaptation decisions made by FUSION. The following subsections answer 3 research questions:

Q1.1	How accurate is the prediction outside of the trained behavior?
Q1.2	How does the quality of prediction increase with the number of adaptation steps after a concept drift?

Q1.3 How good are the adaptation decisions during a concept drift?

9.1.1 [Q1.1]Accuracy of Learning

Throughout this section, we use the term *observation* to indicate an adaptation step made by FUSION and its effect. In other words, 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 Percent (ADP)*, which is defined as: $\left|\frac{Ai-Pi}{Ai}\right| * 100$, where *Pi* is the predicted value of the metric and *Ai* is the actual value that is collected from the running system at observation *i*. In the experiments reported here, learning is initiated if the average absolute difference 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 22 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. The Y-axis represents ADP at a given observation. We used two online Queuing Network (QN) models as a benchmark to ascertain the presence of a significant enough concept drift in the system. By online, we mean that QN model parameters (i.e., workload and services demands) are updated at runtime to cope with variability in the context and the behavior of the system. The QN models are as follows:

1. **QN1**: This queuing model represents the system's behavior in the NT scenario.
2. **QN2**: This queuing model represents the behavior of the system after IF scenario (i.e. after concept drift has taken place). QN2 captures unforeseen contentions at the software level that had appeared as a result of the runtime indexing failure.

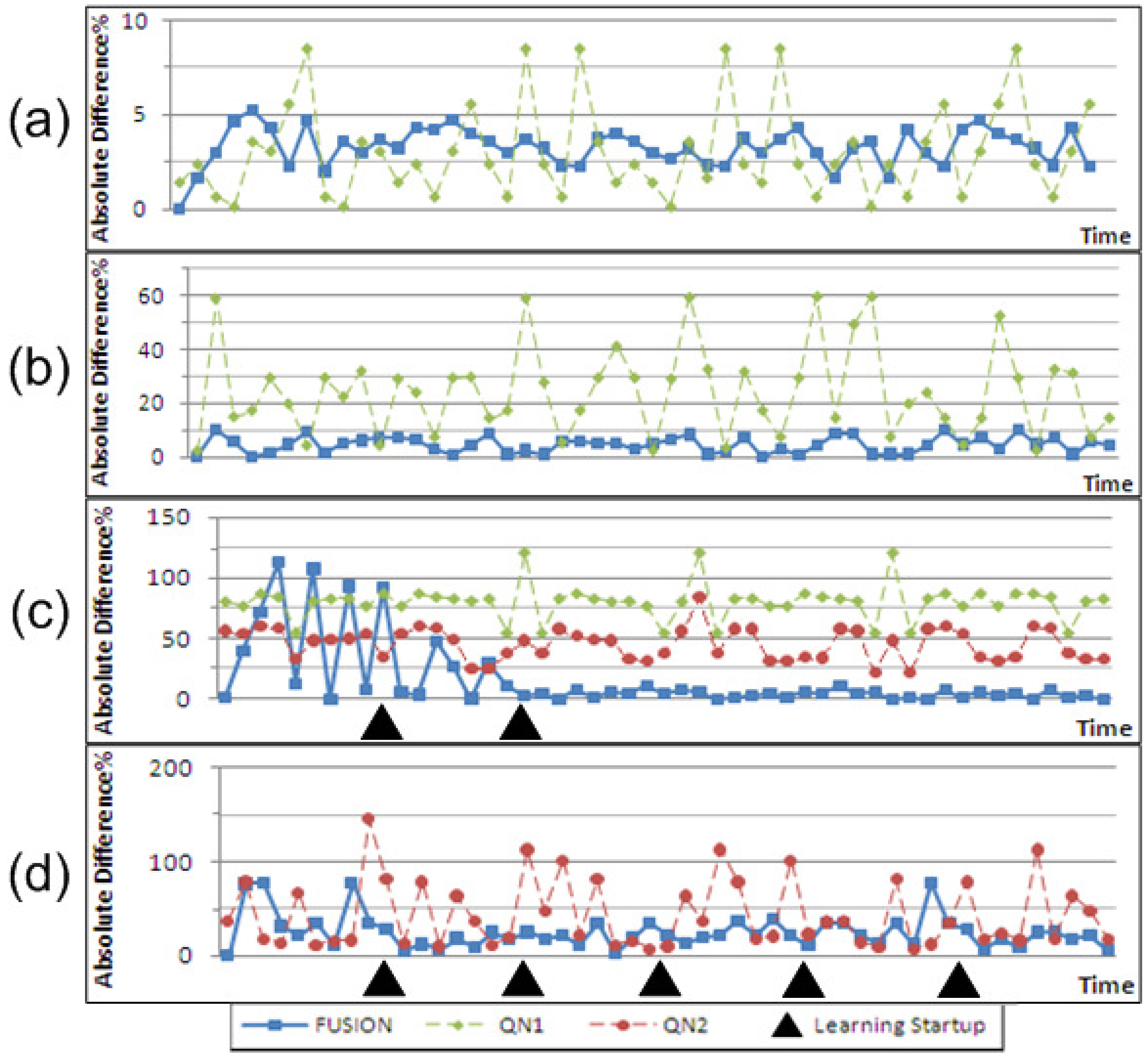


Figure 22. Accuracy of learned functions for "Quote Res. Time" metric: (a) Normal Traffic, (b) Varying Traffic, (c) Database Indexing Failure, (d) Randomized DoS Traffic.

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 analytical models that are tailored for the specific application domain (This is while performance may be only one goal of interest out of many in the system). In the accuracy comparisons reported below for TRS we constructed a subset of QN models that correspond to the feature selections made by FUSION.

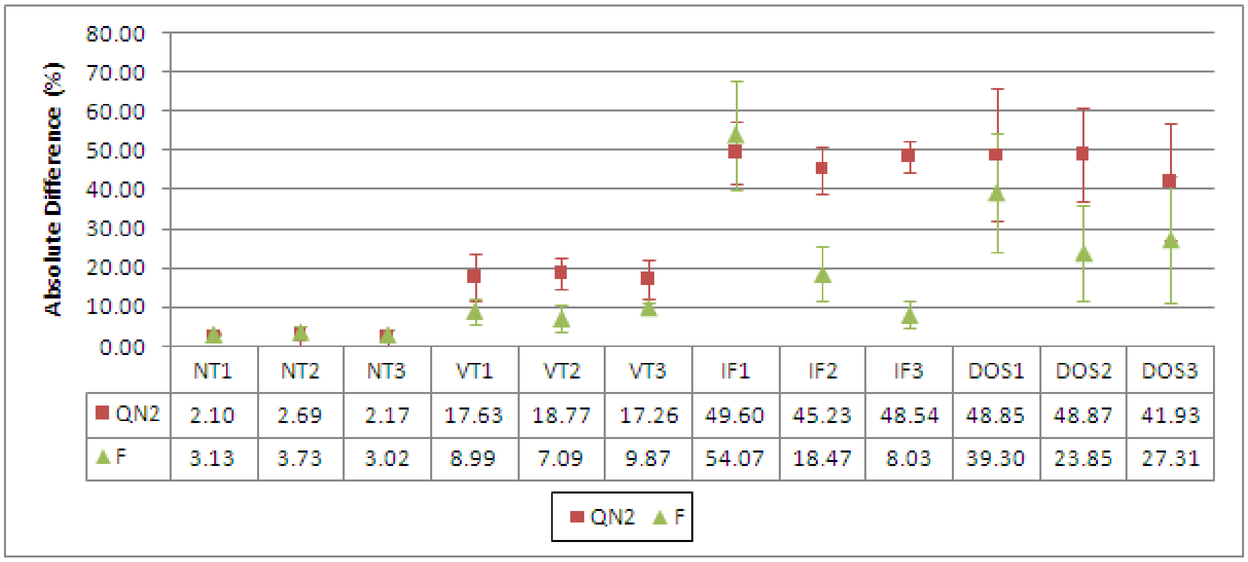
Figure 22a shows the TRS system under the NT scenario, where ADP for both FUSION and QN1 come within 5%, and often less. As expected, this indicates that both FUSION and QN1 achieve good level of accuracy under expected execution conditions. QN1's level of accuracy was within an average ADP of 2.9% and some spikes of 5-8%. This is due to the fact that some service demands in TRS are not fully exponentially distributed as assumed by the model.

Figure 22b 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 22c shows the TRS system under the IF scenario. It shows that when there are concept drifts in the system, FUSION is capable of learning the new behavior and adjusting its model. FUSION's ADP increases up to an average of 54% for the first 10observation. 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. This can be seen by comparing the difference in ADP between QN1 and QN2. Remaining ADP is attributed to transient effects/noise associated with the database failure. 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 QN1 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 DB 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 data.

Figure 22d 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



	NT1	NT2	NT3	VT1	VT2	VT3	IF1	IF2	IF3	DOS1	DOS2	DOS3
🟥 QN2	2.10	2.69	2.17	17.63	18.77	17.26	49.60	45.23	48.54	48.85	48.87	41.93
▲ F	3.13	3.73	3.02	8.99	7.09	9.87	54.07	18.47	8.03	39.30	23.85	27.31

Figure 23. Summary of learned functions' accuracy for different scenarios.

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.

Figure 23 plots the learning accuracy obtained from the experiments at 3 different points in time for each of the 4 evaluation scenarios. The figure depicts 95% confidence interval over 30 runs of the experiments explained above.

Table 6. Impact of feature reduction heuristics.

Exp.	NT 1	NT 2	NT 3	VT 1	VT 2	VT 3	IF 1	IF 2	IF 3	DoS 1	DoS 2	DoS 3
FUSION	5	5	5	5	7	7	5	7	8	5	11	11
CS/TO	78											

9.1.2 [Q1.2] Adaptation in Presence of Concept Drifts

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. FUSION applies domain heuristics to reduce the learning space significantly, which has turned out to be a critical factor in speeding up the fine-tuning process. For instance, in the IF scenario, it recovers after 2 round of learning (i.e., 20 adaptation steps). This is mainly attributed to the feature space reduction heuristics that decrease the learning space significantly. Table 6 demonstrates reduction in learning space for each of the four evaluation scenarios. Note that under DoS scenario, FUSION fails to contain the learning space. This confirms our expectation that due to misleading signals in the environment, learning may establish false correlations between features and metrics.

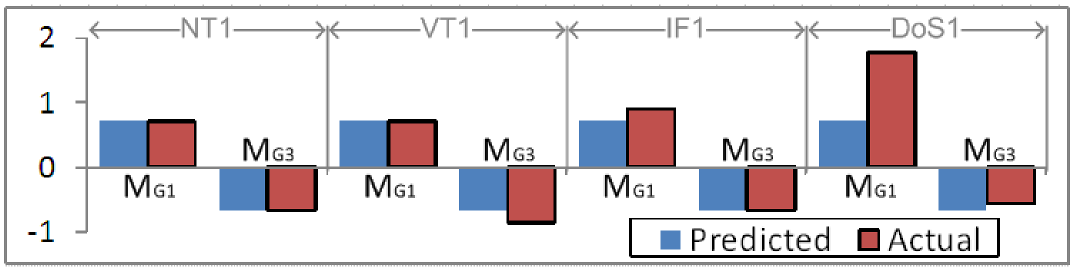


Figure 24. Impact of Feature "Per-Request Authentication" on Metrics "Quote Response Time" and "Quote Quality"

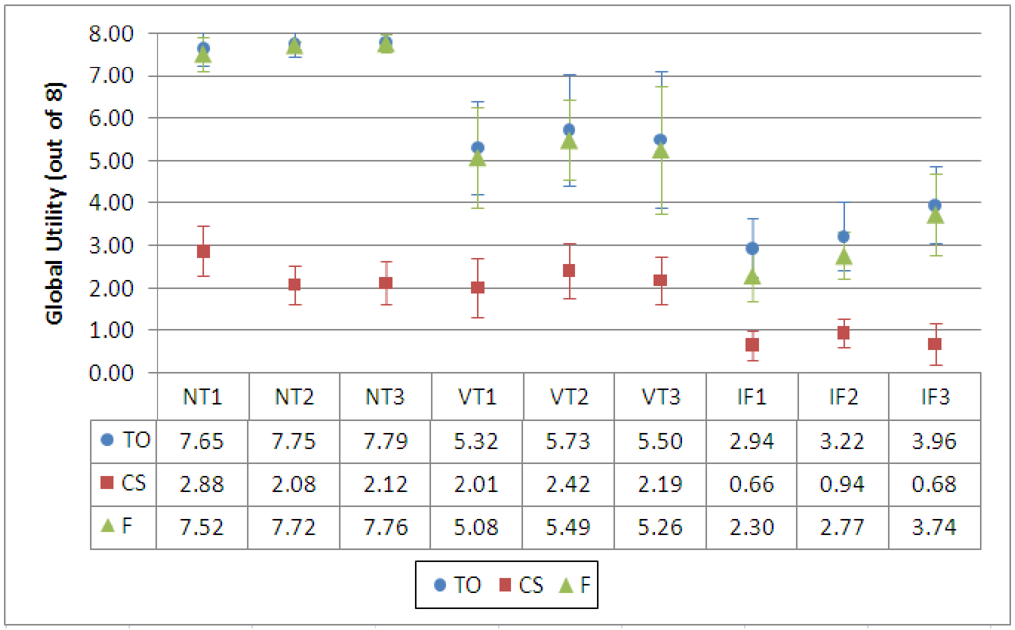
However, the important thing is whether the adaptation decisions made during this period of time (i.e., using an inaccurate model) could further exacerbate the violated goals or not. Figure 24 shows the normalized impact of enabling F_3 on metrics M_{G1} and M_{G3} in the first observation for each of the four scenarios of Figure 22. 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. Instead FUSION makes decisions that are good, but not necessarily optimal, until the knowledge base is refined as will be demonstrated in Section 9.1.3.

9.1.3 [Q1.2] 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 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 Chapter 7, 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 25 plots the global utility obtained from running the optimization at the 3 different points explained earlier for each of the 4 evaluation scenarios discussed earlier. Each data point represents the global utility value (recall the objective function in Section 7.2) 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 deem 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 25 also shows that FUSION find solutions that are significantly better than CS. In turn, this corroborates our assertion in Section 7.2 that FUSION produces a stable fix to goal violations by placing the system in a near-optimal configuration. On the other hand, since CS may find borderline solutions that barely satisfy the goals, due to slight fluctuations in the system, goals may be violated and thus frequent adaptations of the system ensue.

Finally, we should reiterate that the quality of solutions found by FUSION can be affected by the accuracy of induced model. However, as we demonstrated in TRS, FUSION solutions put the system is a better state with respect to global utility.



	NT1	NT2	NT3	VT1	VT2	VT3	IF1	IF2	IF3
● TO	7.65	7.75	7.79	5.32	5.73	5.50	2.94	3.22	3.96
■ CS	2.88	2.08	2.12	2.01	2.42	2.19	0.66	0.94	0.68
▲ F	7.52	7.72	7.76	5.08	5.49	5.26	2.30	2.77	3.74

Figure 25. Global utility for different scenarios.

9.2 Hypothesis #2: Dependencies

In this section, we demonstrate the ability of FUSION to maintain architectural dependencies by enforcing feature model constrains (Section 9.2.1). We also demonstrate how FUSION leverages its knowledge base to even further enhance selection of adaptation steps to ensure stable functioning of the system during adaptation (Section 9.2.2). The following subsections answer 2 research questions:

Q2.1	How can the system be prevented from transitioning to an invalid state during adaptation?
Q2.2	How can the system be prevented from violating additional goals during adaptation?

9.2.1 [Q2.1] Prevention vs. Detection of Architectural Inconsistencies

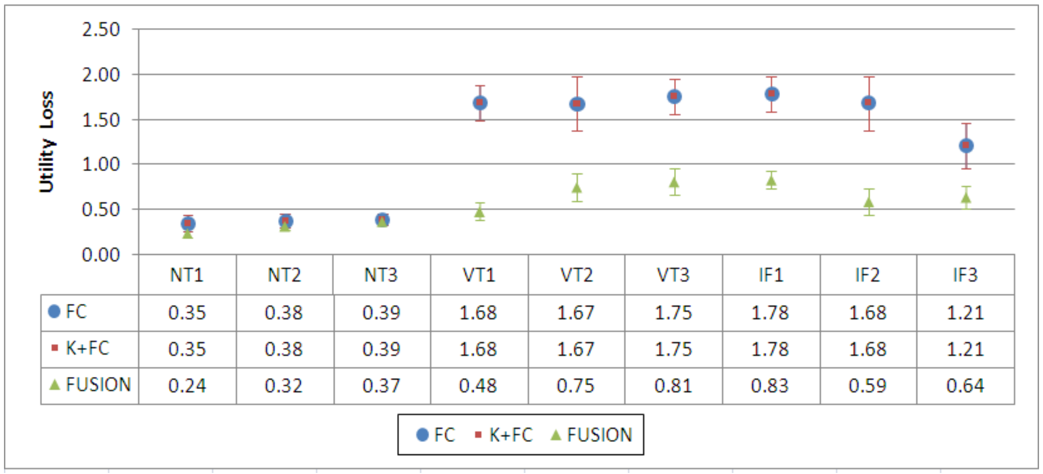
Recall from Section 5.4 that state-of-the-art architecture-based approaches engineers define a significant number of complex constraints most of which are to detect invalid configurations and exclude them from the space. Instead, FUSION eliminates invalid configurations from its decision space completely using experts' knowledge. This is the key advantage of preventing selection of inconsistent configurations over detecting them at runtime, which is state-of-the-art.

One concern would be if engineers make mistakes in the feature model. For instance, a security engineer may ignore a dependency relationship between one of his/her features and a feature that is developed by usability engineers. Usually, automated techniques are used to discover such inter-dependencies due to the natural separation among engineering teams. In this case, several static and dynamic analysis techniques can be used to verify the correctness of the architecture automatically. For instance, in [99], we presented an automated approach for ensuring validity of the feature model relationships by checking for the presence of critical pairs in the underlying UML class, sequence, and state diagrams (*2^F state space*). Although static and dynamic architectural checking are beyond the scope of this thesis, we argue that through the use of such approaches, it reasonable to assume correctness of the feature models produced by the engineers.

9.2.2 [Q2.2] Protecting System Goals During Adaptation

FUSION leverages its knowledge base to find adaptation paths that minimize violation of system goals in order to achieve higher overall stability during adaptation. We compare FUSION results against two competing techniques.

The first technique (referred to as *FC*) uses a path search formulation that enforces feature model constraints during adaptation but does not leverage the knowledge based to prune the search space. The second technique (referred to as *K+FC*) uses a path search formulation that is identical to FC; however, it leverages the knowledge base (K). As you may recall from Section 7.3, FUSION, which can be also refered to as *K+FC+UL*, exhibits three characterstics: 1) It leverages knowledge base (K) tp prune the search space, 2) It finds a path that satisfies feature constraints (FC), and (3) It picks the path that minimizes utility loss (UL).



	NT1	NT2	NT3	VT1	VT2	VT3	IF1	IF2	IF3
● FC	0.35	0.38	0.39	1.68	1.67	1.75	1.78	1.68	1.21
▪ K+FC	0.35	0.38	0.39	1.68	1.67	1.75	1.78	1.68	1.21
▲ FUSION	0.24	0.32	0.37	0.48	0.75	0.81	0.83	0.59	0.64

Figure 26. Quality of adaptation paths.

Figure 26 shows the utility loss obtained from the path search using the three formulations in the first 3 evaluation scenarios. Each data point represents the utility loss value at a given adaptation step. FUSION produces paths that minimize utility loss in all 3 scenarios NT, VT, and IF. To the contrary, FC and K+FC produce paths that worsen the utility loss and violate additional goals in scenarios VT and IF. This demonstrates that FUSION can produce paths that minimize utility loss and enforce feature model constraints.

9.3 Hypothesis #3: Efficiency

In this section, we demonstrate performance benchmarks to demonstrate the efficiency and suitability of FUSION for online reasoning (i.e., analysis and planning). We first cover the overhead of the Induce activity in the learning cycle in Section 6.2. Then, we cover the adaptation cycle to demonstrate the efficiency of the Plan and Effect activities in Sections 9.3.2 and 9.3.3 respectively. The following subsections answer 3 research questions:

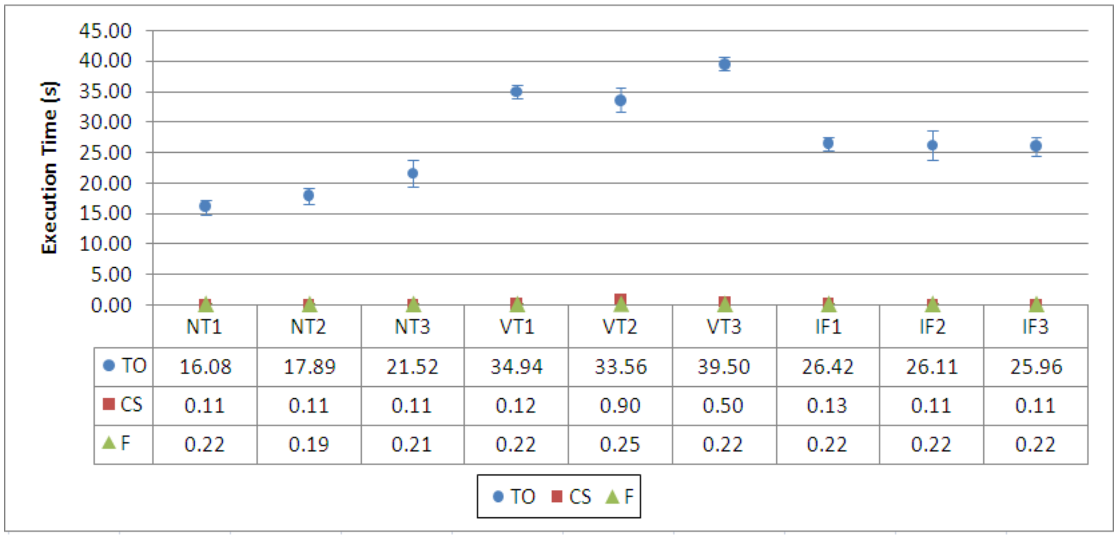
Q3.1	How does learning time increase with the number of observations?
Q3.2	How long does it take to find an optimal configuration of the system?
Q3.3	How long does it take to find an adaptation path that ensures stable functioning during adaptation?

9.3.1 [Q3.1] Overhead of Learning (Induce Activity)

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 online learning. One of the principle factors affecting learning overhead is the number of observations required to make accurate inductions. Table 7 lists the execution time for a given number of observations in the extended TRS system. Simple linear regression takes an insignificant amount of time with large number of observations. In our experiments FUSION performed online learning on a maximum of 30 observations, which from Table 7 could be verified to have presented an insignificant

Table 7. Induction execution time in milliseconds

# of Observations	50	500	528	822	903	1227	1389
Simple Regression	20	30	30	50	60	70	80
M5-Model Trees	60	110	130	130	130	160	230
SVM Regression	190	2310	3230	7330	8740	18700	29830



	NT1	NT2	NT3	VT1	VT2	VT3	IF1	IF2	IF3
● TO	16.08	17.89	21.52	34.94	33.56	39.50	26.42	26.11	25.96
🟥 CS	0.11	0.11	0.11	0.12	0.90	0.50	0.13	0.11	0.11
▲ F	0.22	0.19	0.21	0.22	0.25	0.22	0.22	0.22	0.22

Figure 27. Optimization execution time for different scenarios.

overhead of less than 20 milliseconds. This efficiency is due to the pruning of the feature selection space and significance test described in Chapter 6.

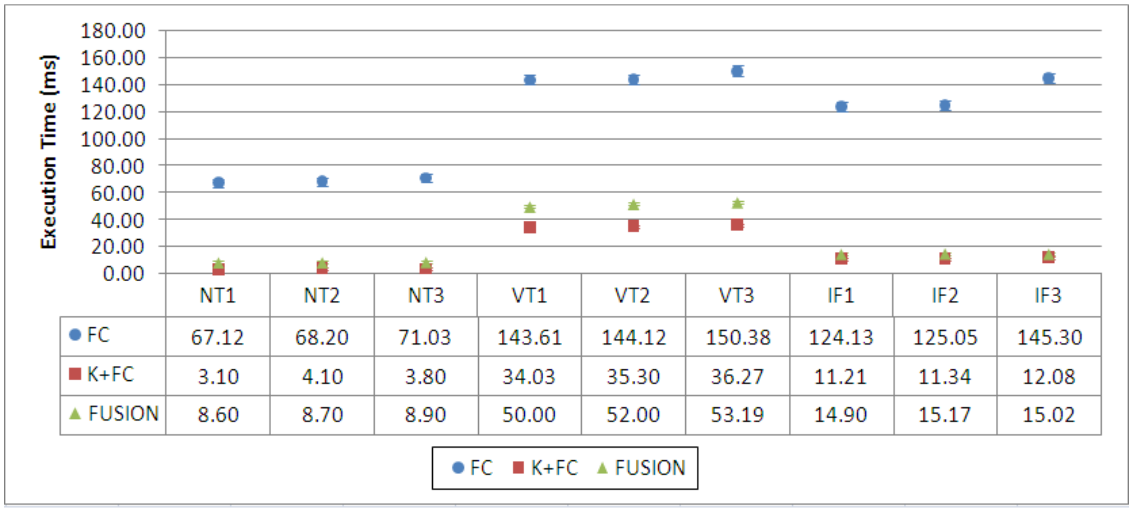
9.3.2 [Q3.2] Efficiency of Optimization (Plan Activity)

In Section 9.1.3, we compared the quality of FUSION with two approaches: Traditional Optimization (TO) and Constraint Satisfaction (CS). In this section we focus on the efficiency aspects of this experiment. As you may recall from Section 7.2, FUSION achieves efficient analysis 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 27 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 25. 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 demonstrates that FUSION is not only able to find solutions that are comparable in quality to those found by TO, but also achieves this at 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.

9.3.3 [Q3.3] Efficiency of Path Search (Effect Activity)

In Section 9.2.2, we compared the quality of FUSION adaptation paths with two approaches: FC and K+FC. In this section, we focus on the efficiency aspects of this experiment. As mentioned in Section 7.3, in contrast to FC, FUSION achieves efficient planning by using the knowledge base to dynamically tailor a search space that is relevant to the violated goals. FUSION also takes in consideration the objective of minimizing



	NT1	NT2	NT3	VT1	VT2	VT3	IF1	IF2	IF3
● FC	67.12	68.20	71.03	143.61	144.12	150.38	124.13	125.05	145.30
■ K+FC	3.10	4.10	3.80	34.03	35.30	36.27	11.21	11.34	12.08
▲ FUSION	8.60	8.70	8.90	50.00	52.00	53.19	14.90	15.17	15.02

Figure 28: Path search execution time in milliseconds for different scenarios.

utility loss during adaptation, which is totally ignored in the K+FC approach.

Figure 28 shows the execution time for solving the path search problem for all 3 scenarios NT, VT, and IF of Figure 26. Incorporating knowledge into the path search problem 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 nature of adaptation decisions taken and the number of features being changed.

Note that execution time of FUSION is overall 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, at execution point "**VT 3"**, the system is running at a peak workload in which the search space has no adjacent nodes that satisfy all goals. As a result, a few additional neighbor nodes 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 without compromising efficiency.

Chapter 10: Conclusion

In this dissertation, 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 [11], and (2) a new method of assessing and reasoning about adaptation decisions through online learning [3].

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

10.1 Contributions

The result of this research is a framework, entitled FeatUre-oriented Self-adaptatION (FUSION), which combines feature-models (i.e., as an explicit representation of domain experts' knowledge) with online machine learning. Domain knowledge, represented in feature-models, adds structure to online learning, which in turn improves the quality of adaptation decisions. The following key contribution of the FUSION framework as an approach for building self-adaptive software systems:

1. FUSION reduces the complexity of building adaptation logic for software systems, since it does not require an explicit representation of the managed system's internal structure. It only requires a black-box specification of the system's adaptation choices (i.e., features) as input.
2. FUSION copes with the changing dynamics of the system, even those that were unforeseen at design time, through incremental observation and induction (i.e., online learning).
3. By encapsulating the engineer's knowledge of the inter-dependencies among the system's constituents as feature-model dependencies, FUSION can ensure stable functioning and protect system goals during and after adaptation.
4. FUSION uses features and inter-feature relationships to significantly reduce the configuration space of a sizable system, making runtime analysis and learning feasible.

10.2 Limitations and Future Work

The following are key limitation areas of FUSION and potential areas of future work:

1. **Learning Uncertainty**: Currently, FUSION handles uncertainty in the system and surrounding context in an adhoc fashion. That is, it does not capture/learn the distribution of uncertainty so that stochastic techniques can be applied to improve the quality of adaptation decisions.

2. **Proactive Adaptation**: FUSION's *Detect* activity monitors the running system and kicks of the adaptation cycle whenever a goal violation is detected. In some real world systems, waiting until a goal violation happens and then reacting to it may be very costly. An alternative approach to this is proactive adaptation. For instance, one approach would be to monitor for certain drop in the utility and initiate adaptation whenever the system is likely to violate a goal. Such an approach requires a forecasting mechanism such as advanced time-series analysis techniques [9].
3. **Self-Training**: We intend to investigate opportunistic self-training as a way of detecting emerging behavior in order to keep FUSION KB up-to-date before adaptation decisions are made on the real system. For instance, one possibility would be applying self-adaptation decisions on a shadow clone of the running system or use external analytical models to train FUSION. Hybrid approaches have proven in the literature to inherit the advantages of combined paradigms.
4. **Generalized Multi-criteria Temporal Planning**: Effect activity can include additional parameters in the path search process. For instance, Effect can use AI planning techniques [58] to address the temporal as well as the consistency aspects of *Effect*. For instance, it can use [1][4] to find: 1) the fastest path of placing the system in an optimal feature selection and 2) a path that has minimal disruption on system components. We also believe additional search heuristics can be developed to enhance the path search process.

5. **Computer Assisted Feature Model Generation**: FUSION relies heavily on the feature model that is exposed by the engineers. Often this results in limiting the space of configurations significantly. Despite the efficiency advantage, this often reduces FUSION's ability to converge to a solution. In some cases, a goal violation can be addressed if the search takes place at the architecture level. We believe that computer assisted feature model generation may be a viable venue to maximize the chance of incorporating optimal configurations at the architecture into the feature selections in a pre-computed manner.

References

References

[1] J. F. Allen, J. A. Koomen. Planning Using a Temporal World Model. *8th International Joint Conference on Artificial Intelligence*, Karlsruhe, Germany, 1983.

[2] An Architectural Blueprint for Autonomic Computing, IBM white paper, 4th edition, June 2006. http://www-01.ibm.com/software/tivoli/autonomic/pdfs/AC_Blueprint_White_Paper_4th.pdf

[3] E, Alpaydın. Introduction to Machine Learning. *Adaptive Computation and Machine Learning,* MIT Press, 2004.

[4] N. Arshad, D. Heimbigner, A.L. Wolf. Deployment and dynamic reconfiguration planning for distributed software systems. *Proceedings 15th IEEE International Conference on Tools with Artificial Intelligence*, Sacramento, CA, Nov 2003.

[5] J. Andersson, R. de Lemos, S. Malek, and D. Weyns. Modeling Dimensions of Self-Adaptive Software Systems. In *Software Engineering for Self-Adaptive Systems, Editors B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, Lecture Notes on Computer Science Hot Topics, Springer*, 2009.

[6] J. Andersson, R. de Lemos, S. Malek, and D. Weyns. Reflecting on Self-Adaptive Software Systems. *ICSE 2009 workshop on Software Engineering for Adaptive and Self-Managing Systems*, Vancouver, Canada, May 2009.

[7] D. Batory. Feature Models, Grammars, and Propositional Formulas. *9th International Conference on Software Product Lines*, Rennes, France, Sep. 2005.

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

[9] C. Chatfield. The analysis of time series, an introduction, sixth edition. *Chapman & Hall/CRC*. New York,2004.

[10] B. Cheng, et al. Software Engineering for Self-Adaptive Systems: A Research Roadmap. *Software Engineering for Self-Adaptive Systems, LNCS Hot Topics,* 2009,

pp. 1-26.

[11] P. Clements, L. Northrop. Software Product Lines: Practices and Patterns. *SEI Series in Software Engineering*, 2001.

[12] E.R. Cook. A time series approach to tree-ring standardization, *Ph.D. dissertation, University of Arizona, Tucson*,1985.

[13] E. Dashofy, A. van der Hoek, R. N. Taylor. An Infrastructure for the Rapid Development of XML-based Architecture Description Languages. *Proceedings of the 24th International Conference on Software Engineering*, pp. 266 - 276, 2002.

[14] G. Edwards, S. Malek, and N. Medvidovic. Scenario-Driven Dynamic Analysis of Distributed Architectures. *International Conference on Fundamental Approaches to Software Engineering*, Braga, Portugal, March 2007.

[15] A. Elkhodary, S. Malek, N. Esfahani. On the Role of Features in Analyzing the Architecture of Self-Adaptive Software Systems. *4th International Workshop on Models at Runtime*, Denver, Colorado, October 2009.

[16] N. Esfahani, S. Malek, J. P. Sousa, H. Gomaa, and D. A. Menasce. A Modeling Language for Activity-Oriented Composition of Service-Oriented Software Systems. *ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems (MODELS 09)*, Denver, Colorado, October 2009.

[17] J.H. Friedman. Multivariate Adaptive Regression Splines. *The Annals of Statistics 19 (1), 1-67*. 1991.

[18] G. Garlan et al. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *IEEE Computer*, 2004, pp. 46-54.

[19] J, Gehrke. Classification and Regression Trees (C&RT). *Encyclopedia of data warehousing and mining*, 2008.

[20] J. C. Georgas, and R. N. Taylor. Towards a knowledge-based approach to architectural adaptation management. *Workshop on Self-healing Systems*, Newport Beach, CA, October 2004.

[21] I. Georgiadis, J. Magee, and J. Kramer. Self-organising software architectures for distributed systems. *Workshop on Self-healing Systems*, Charleston, SC, November

2002.

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

[23] H. Gomaa, M. Hussein. Software Reconfiguration Patterns for Dynamic Evolution of Software Architectures. *Working IEEE/IFIP Conference on Software Architecture*. 2004.

[24] H. Gomaa, K. Hashimoto, M. Kim, S. Malek, D. A. Menasce. Software Adaptation Patterns for Service-Oriented Architectures. *25th ACM Symposium on Applied Computing (SAC 2010), Dependable and Adaptive Distributed Systems track*, Sierre, Switzerland, March 2010.

[25] G. Gross, and C. M. Harris. Fundamentals of queuing theory (2nd ed.). *John Wiley & Sons*, 1985.

[26] S. Hallsteinsen, M. Hinchey, S. Park, K. Schmid. *Dynamic Software Product Lines. IEEE Computer*. 2008.

[27] P.K. Jayaraman, J. Whittle, A. Elkhodary, H. Gomaa. Model Composition in Product Lines and Feature Interaction Detection Using Critical Pair Analysis. *MoDELS 2007*. 151-165.

[28] K. Kang, S. Cohen, et al. Feature-Oriented Domain Analysis (FODA) Feasibility Study. *Technical Report CMU/SEI-90-TR-21*, Software Engineering Institute, Pittsburgh A, November 1990.

[29] M. Kantardzic. Data Mining: Concepts, Models, Methods, and Algorithms. *John Wiley & Sons*. 2003.

[30] J. O. Kephart, and D. M. Chess. The Vision of Autonomic Computing. *IEEE Computer*, vol. 36, 2003, pp. 41-50.

[31] D. Kim, and S. Park. Reinforcement Learning-Based Dynamic Adaptation Planning Method for Architecture-Based Self-Managed Software. *International Workshop on Software Engineering For Adaptive and Self-Managing Systems*, Vancouver, Canada, May 2009.

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

[33] J. Kramer, and J. Magee. Self-Managed Systems: an Architectural Challenge. *International Conference on Software engineering*, Vancouver, Canada, May 2007.

[34] D. Kriesel. A Brief Introduction to Neural Networks. *University of Bonn, Epsilon,* Germany, 2005.

[35] K. Lee, et al. Combining Feature-Oriented Analysis and Aspect-Oriented Programming for Product Line Asset Development. *International Conference on Software Product Line*, Baltimore, MD, August 2006.

[36] J. Lee and K. Kang. A feature-oriented approach to developing dynamically reconfigurable products in Product-Line engineering. *Software Product Lines Conference*. Aug. 2006.

[37] J. Magee, et al. Behavior Analysis of Software Architectures. Proceedings of the TC2. First Working IFIP Conference on Software Architecture (WICSA1), pp. 35 - 50, 1999.

[38] S. Malek, C. Seo, S. Ravula, B. Petrus, and N. Medvidovic. Reconceptualizing a Family of Heterogeneous Embedded Systems via Explicit Architectural Support. *International Conference on Software Engineering (ICSE 2007)*, Minneapolis, Minnesota, May 2007.

[39] S. Malek, G Edwards, Y. Brun, H. Tajalli, J. Garcia, I. Krka, N. Medvidovic, M. Mikic-Rakic, and G. Sukhatme. An Architecture-Driven Software Mobility Framework. *Journal of Systems and Software*, *special issue on Software Architecture and Mobility*, December 2009.

[40] S. Malek, M. Mikic-Rakic, and N. Medvidovic. A Style-Aware Architectural Middleware for Resource-Constrained, Distributed Systems. *IEEE Transaction on Software Engingeering*, 31(5), 2005, pp. 256-272.

[41] S. Malek, R. Roshandel, D. Kilgore, and I. Elhag. Improving the Reliability of Mobile Software Systems through Continuous Analysis and Proactive Reconfiguration. *International Conference on Software Engineering (ICSE 2009), New Ideas and Emerging Results Track*, Vancouver, Canada, May 2009.

[42] S. Malek, N. Esfahani, D. A. Menasce, J. Sousa, and H. Gomaa. Self-Architecting Software Systems (SASSY) from QoS-Annotated Activity Models. *ICSE 2009 workshop on Principles of Engineering Service Oriented Systems (PESOS 2009)*, Vancouver, Canada, May 2009.

[43] S. Malek, N. Beckman, M. Mikic-Rakic, and N. Medvidovic. A Framework for Ensuring and Improving Dependability in Highly Distributed Systems. *In R. de Lemos, C. Gacek, and A. Romanowski, eds., Architecting Dependable Systems III, Springer Verlag*, October 2005.

[44] S. Malek, C. Seo, S. Ravula, B. Petrus, and N. Medvidovic. Providing Middleware-Level Facilities to Support Architecture-Based Development of Software Systems in Pervasive Environments. *International Workshop on Middleware for Pervasive and Ad-Hoc Computing (MPAC 2006)*, Melbourne, Australia, November 2006.

[45] S. Malek. A User-Centric Approach for Improving a Distributed Software System's Deployment Architecture. *PhD Dissertation*, University of Southern California, August 2007.

[46] N. Medvidovic, R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transaction on Software Engineering*, vol. 26, no. 1, January 2000.

[47] D. Menasce, M. Bennani, H. Ruan. On the Use of Online Analytic Performance Models in Self-Managing and Self-Organizing Computer Systems. *Self-Star Properties in Complex Information Systems*. Vol. 3460, Springer Verlag, 2005.

[48] D. Menasce, J. M. Ewing, H. Gomaa, S. Malek, and J. P. Sousa. A Framework for Utility-Based Service Oriented Design in SASSY. *Joint WOSP/SIPEW International Conference on Performance Engineering*, San Jose, California, January 2010.

[49] B. Morin, O. Barais, G. Nain, and J-M. Jzquel. Taming Dynamically Adaptive Systems with Models and Aspects. *31st International Conference on Software Engineering*, Vancouver, Canada, May 2009.

[50] B. Morin, F. Fleurey, N. Bencomo, J-M. Jzquel. Solberg, A. Dehlen, V. and Blair, G. An aspect-oriented and model-driven approach for managing dynamic variability. *ACM/IEEE 11th International Conference on Model Driven Engineering Languages and Systems*, Toulouse, France. October 2008.

[51] ODM, http://www.oracle.com/technology/products/bi/odm

[52] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. *International Conference on Software Engineering*, Kyoto, Japan, April 1998.

[53] D. Perry and A.L. Wolf. Foundations for the Study of Software Architecture, *ACM SIGSOFT Software Engineering Notes*, vol. 17, no. 4, pp. 40-52, Oct. 1992.

[54] V. Poladian, J. P. Sousa, D. Garlan, and M. Shaw. Dynamic Configuration of Resource-Aware Services. *International Conference on Software Engineering*, Edinburgh, Scotland, May 2004.

[55] Princeton Review. http://www.princetonreview.com/home.asp

[56] L. R. Rabiner. A Tutorial on Hidden Markov Models, *in proceedings of the IEEE, vol. 77*, pp. 257-286, 1989.

[57] K. Rieck, P. Laskov. Language Models for Detection of Unknown Attacks in Network Traffic. *Journal in Computer Virology.* December, 2006.

[58] S. Russell and P. Norvig, Artificial Intelligence: A Modern Approach, *Prentice Hall*, Englewood Cliffs, NJ, 1995.

[59] T. Ryutov, L. Zhou, C. Neuman, T. Leithead, K. Seamons. Adaptive Trust Negotiation and Access Control. *ACM Symposium on Access Control Models and Technologies*, pps. 139-146, 2005.

[60] M. Sabhnani, G. Serpen. Application of Machine Learning Algorithms to KDD Intrusion Detection Dataset within Misuse Detection Context. *Proceedings of International Conference on Machine Learning: Models, Technologies and Applications*, pp. 209-215.

[61] A. J. Smola and B. Schölkopf. A tutorial on support vector regression. *Statistics and Computing*, vol. 14, 2004.

[62] GMU Software Engineering Seminar Series. http://cs.gmu.edu/~smalek/seminar.html

[63] J. P. Sousa, V. Poladian, D. Garlan, B. Schmerl, and M. Shaw. Task-based adaptation for ubiquitous computing. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, vol. 36, no. 3, 328-340, May 2006.

[64] J. P. Sousa, R. K. Balan, V. Poladian, D. Garlan, and M. Satyanarayanan. User Guidance of Resource-Adaptive Systems. *International Conference on Software and Data Technologies*, Porto, Portugal, July 2008.

[65] D. Sykes, et al. From goals to components: a combined approach to self-management. *International Workshop on Software Engineering For Adaptive and Self-Managing Systems*, Leipzig, Germany, May 2008.

[66] R. N. Taylor, N. Medvidovic, and E. Dashofy. Software Architecture: Foundations, Theory, and Practice, *John Wiley & Sons*, 2008.

[67] G. Tesauro, N. K. Jong, R. Das, and M. N. Bennani, M.N. A Hybrid Reinforcement Learning Approach to Autonomic Resource Allocation. *International Conference on Autonomic Computing*, Washington, DC, June 2006.

[68] G. Tesauro. Reinforcement Learning in Autonomic Computing: A Manifesto and Case Studies. *IEEE Internet Computing*, vol. 11, no 1, pages 22-30, 2007.

[69] P. Trinidad, A. Ruiz-Cortes, J.P. Na. Mapping feature models onto component models to build dynamic software product lines. *International Workshop on Dynamic Software Product Line*. 2007.

[70] S. Trujillo, D. Batory, O. Diaz. Feature Oriented Model Driven Development: A Case Study for Portlets. *Proceedings of the 29th international conference on Software Engineering*, Minneapolis, MN, 2007.

[71] L. A. Wolsey, Integer Programming, *John Wiley & Sons*, New York, NY, 1998.

[72] Y. Yuan, M.J. Shaw. Induction of fuzzy decision trees. *Fuzzy Sets and Systems, Elsevier North-Holland,* vol. 69, no 2, 1995.

[73] Malek, S. A User-Centric Framework for Improving a Distributed Software System's Deployment Architecture. *In proceedings of the doctoral track at the 14th ACM SIGSOFT Symposium on Foundation of Software Engineering*. Portland, Oregon, 2006.

[74] Malek, S. Mikic-Rakic, M. and Medvidovic, N. A Decentralized Redeployment Algorithm for Improving the Availability of Distributed Systems. *In proceedings of the 3rd International Working Conference on Component Deployment (CD 2005)*, Grenoble, France, Nov. 2005.

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

[76] Dashofy, E. M. Van der Hoek, A. and Taylor, R. N. Towards architecture-based self-healing systems. In *Proceedings of the First Workshop on Self-Healing Systems*, 2002.

[77] Fleurey, F. and Solberg, A. A DSML supporting Specification, Simulation and Execution of Dynamic Adaptive Systems. *Proceedings of the ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems*. Denver, Colorado. 2009.

[78] Matevska-Meyer, J., Hasselbring, W., Reussner, R.: Software Architecture Description supporting Component Deployment and System Runtime Reconfiguration. *Workshop on Component-Oriented Programming (WCOP 2004)*. Oslo, Norway (2004).

[79] Caporuscio, M., Marco, A.D., Inverardi, P.: Model-based system reconfiguration for dynamic performance management. Journal of Systems and Software. 80, 455-473 (2007).

[80] Whittle, J., Sawyer, P., et al.: RELAX: a language to address uncertainty in self-adaptive systems requirement. Requirements Engineering. (2010).

[81] Garlan, D., Schmerl, B.: Model-based adaptation for self-healing systems. *Workshop on Self-healing systems. pp. 27-32,* Charleston, South Carolina (2002).

[82] Bencomo, N., Blair, G.: Using Architecture Models to Support the Generation and Operation of Component-Based Adaptive Systems. *Software Engineering for Self-Adaptive Systems*. pp. 183-200Springer-Verlag (2009).

[83] Whittle, J., Jayaraman, P. Elkhodary, A. et al.: MATA: A Unified Approach for Composing UML Aspect Models Based on Graph Transformation. *Transactions on Aspect-Oriented Software Development VI. pp. 191-237* (2009).

[84] Becker, B., Giese, H.: Modeling of correct self-adaptive systems: a graph transformation system based approach. *International Conference on Soft Computing as Transdisciplinary Science and Technology*. pp. 508-516ACM, Cergy-Pontoise, France (2008).

[85] Giese, H., Wagner, R.: Incremental Model Synchronization with Triple Graph Grammars. *International Conference on Model Driven Engineering Languages and Systems (MoDELS 2006)*. pp. 543--557 (2006).

[86] Vogel, T., Neumann, S., et al.: Model-driven architectural monitoring and adaptation for autonomic systems. *International Conference on Autonomic Computing. pp. 67-68ACM*, Barcelona, Spain (2009).

[87] Vogel, T., Neumann, S., et al.: Incremental Model Synchronization for Efficient Run-Time Monitoring. *Workshop on Models@run.time.* , Denver, Colorado, USA (2009).

[88] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson, Feature-Oriented Domain Analysis (FODA) Feasibility Study. 1990.

[89] Tajalli, H. Garcia, J. Edwards, G. and Medvidovic, N. "PLASMA: a plan-based layered architecture for software model-driven adaptation," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, 2010, pp. 467–476.

[90] Ghanbari, S. Soundararajan, G. Chen, J. and Amza, C. "Adaptive Learning of Metric Correlations for Temperature-Aware Database Provisioning," in *Int'l Conf. on Autonomic Computing*, Jacksonville, Florida, 2007, p. 26.

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

[92] M. I. Jordan and R. A. Jacobs, "Hierarchical mixtures of experts and the EM algorithm," *Neural Comput.*, vol. 6, no. 2, pp. 181-214, 1994.

[93] Cheng, B. Sawyer, P. Bencomo, N. Whittle, J. A Goal-Based Modeling Approach to Develop Requirements of an Adaptive System with Environmental Uncertainty. *Proceedings of the ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems*. Denver, Colorado. 2009.

[94] Bennani, M. Menasce, D. Resource Allocation for Autonomic Data Centers using Analytic Performance Models. *Proceeding of the International Conference on autonomic Computing (ICAC)*, Seattle, Washington, 2005.

[95] Bennani, M. Menasce, D. Assessing the Robustness of Self-Managing Computer systems under Highly Variable Workloads. *Proceeding of the International*

Conference on autonomic Computing (ICAC), New York, 2004.

[96] Tsymbal, A. The problem of concept drift: definitions and related work. *Technical Report. Department of Computer Science. Trinity College Dublin*, Ireland, 2004.

[97] Widmer, G. Kubat, M. Learning in the presence of concept drift and hidden contexts. *Journal of Machine Learning 23*, 1996, pp. 69-101.

[98] Kolter, J.Z. and Maloof, M.A. Dynamic Weighted Majority: An ensemble method for drifting concepts. *Journal of Machine Learning Research 8:2755--2790*, 2007.

[99] Jayaraman, P. Whittle, J. Elkhodary, A. Gomaa, H.: Model Composition in Product Lines and Feature Interaction Detection Using Critical Pair Analysis. *Proceeding of the MoDELS International Conference, pp. 151-165*. 2007.

[100]B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi. Analytic modeling of multitier internet applications. *ACM Transactions on the Web*, 1(1):2–37, 2007.

[101]Ardagna, D. Ghezzi, C. and Mirandola, R. Rethinking the use of models in software architecture. *Proceeding of International Conference Series on the Quality of Software Architectures, pages 1–27*, 2008.

[102]Wang, G. Shan, S. Review of metamodeling techniques in support of engineering design optimization. *Mechanical Design*, 129(4):370–380, 2007.

[103]Menacse, D. Simple analytic modeling of software contention. *ACM SIGMETRICS Performance Evaluation Review. Volume 29 Issue 4,* New York, USA. 2002.

[104]Gambi, A. Toffetti, G. Pezze, M. Protecting SLAs with surrogate models. Proceedings of the 2nd International Workshop on Principles of Engineering Service-Oriented Systems. New York. 2010.

[105]Solomatine, D., Data-driven modelling: paradigm, methods, experiences. Proceeding of the 5th International Conference on Hydroinformatics, UK. 2002.

[106]Ware, M. Frank, E. Holmes, G. Hall, M. Witten, I.H. Interactive machine learning - letting users build classifiers. *International Journal on Human-Computer Studies*. 2000.

[107]Kramer, J. Magee, J. The Evolving Philosophers Problem: Dynamic Change Management. IEEE Transaction on Software Engineering 16, 11. 1990.

[108]Sacks, J., Welch, W. J., Mitchell, T. J. and Wynn, H. P. Design and Analysis of Computer Experiments. *Statistical Science*, 4(4), pp. 409-435. 1989.

[109]Perdu, D. Levis, A. Adaptation as a Morphing Process: A Methodology for the Design and Evaluation of Adaptive Organization Structures. *Computational and Mathematical Organization Theory*, Vol. 4, No. 1, Spring 1998, pp. 5-41.

[110]Damaël, J. Levis, A. On Generating Variable Structure Architectures for Decision Making Systems. *Information and Decision Technologies*, pp. 233-255. 1994.

[111]Lu, Z. & Levis, A. Coordination in Distributed Decision Making, *Proceedings of IEEE International Conference on Systems, Man, and Cybernetics*, pp. 891 - 897. Chicago. 1992.

[112]Sutton, R. Barto, A. Reinforcement Learning: An Introduction. Chapter 6 – Actor-Critic Methods. MIT Press, Cambridge, MA, 1998.

[113]Si, J. Barto, A. Powell, W. Wunsch, D. Handbook of Learning and Approximate Dynamic Programming, pages 359-380. Wiley-IEEE Press. 2004.

[114]Ng, A. Harada, D. Russell, S.. Policy invariance under reward transformations: theory and application to reward shaping. *In the Proceedings of the Sixteenth International Conference on Machine Learning*, June 27-30, pp.278-287. 1999.

[115]Zhao, D. Hu, Z. Supervised adaptive dynamic programming based adaptive cruise control. *IEEE Symposium on Adaptive Dynamic Programming And Reinforcement Learning (ADPRL),* Paris. 2011.

[116]Lu, H. Plataniotis, K. Venetsanopoulos, A. MPCA: Multilinear principal component analysis of tensor objects. *IEEE Transation on Neural Networks, vol. 19, no. 1, pp. 18–39*. 2008.

[117]Object Management Group. "Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT) 1.1". 2011.

[118]Esfahani, N, Malek, S. On the Role of Architectural Styles in Improving the Adaptation Support of Middleware Platforms. The 4th European Conference on Software Architecture (ECSA 2010), Copenhagen, Denmark, August 2010.

[119] Elkhodary, A. Esfahani, N. Malek, S. FUSION: A Framework For Engineering Self-Tuning Self-Adaptive Software Systems. *Proceedings of ACM/IEEE International Conferernce on Foundations of Software Engineering (FSE)*, Santa Fe, 2010.

Curriculum Vitae

Ahmed Elkhodary is a software engineering academic and practitioner with over 8 years of technical experience in academic research, R&D, enterprise architecture, systems engineering, software design and development.

Ahmed has earned a BS degree in Computer Science (King Abdul Aziz University), an MS degree in Software Engineering (George Mason University), and a PhD in Information Technology from George Mason University.

Ahmed is an independent Enterprise Architect with government consulting experience at the Department of The Treasury (Washington DC).