Inferring Software Component Interaction Dependencies for Adaptation Support

NAEEM ESFAHANI, Google Inc.
ERIC YUAN and KYLE R. CANAVERA, George Mason University
SAM MALEK, University of California, Irvine

A self-managing software system should be able to monitor and analyze its runtime behavior and make adaptation decisions accordingly to meet certain desirable objectives. Traditional software adaptation techniques and recent "models@runtime" approaches usually require an *a priori* model for a system's dynamic behavior. Oftentimes the model is difficult to define and labor-intensive to maintain, and tends to get out of date due to adaptation and architecture decay. We propose an alternative approach that does not require defining the system's behavior model beforehand, but instead involves mining software component interactions from system execution traces to build a probabilistic usage model, which is in turn used to analyze, plan, and execute adaptations. In this article, we demonstrate how such an approach can be realized and effectively used to address a variety of adaptation concerns. In particular, we describe the details of one application of this approach for safely applying dynamic changes to a running software system without creating inconsistencies. We also provide an overview of two other applications of the approach, identifying potentially malicious (abnormal) behavior for self-protection, and improving deployment of software components in a distributed setting for performance self-optimization. Finally, we report on our experiments with engineering self-management features in an emergency deployment system using the proposed mining approach.

Categories and Subject Descriptors: D.2.11 [**Software Engineering**]: Software Architectures

General Terms: Algorithms

Additional Key Words and Phrases: Data mining, self-adaptation, component-based software

ACM Reference Format:
Naeem Esfahani, Eric Yuan, Kyle R. Canavera, and Sam Malek. 2016. Inferring software component interaction dependency for adaptation support. ACM Trans. Auton. Adapt. Syst. 10, 4, Article 26 (February 2016), 32 pages.
DOI: http://dx.doi.org/10.1145/2856035

1. INTRODUCTION

A self-managing software system is comprised of two conceptual parts, a *base-level subsystem* that provides the software system's application logic and domain functionalities and a *metalevel subsystem* that manages the behavior of the base-level subsystem to satisfy certain desirable objectives, for example, performance, security, reliability, etc.

This work was supported in part by awards CCF-1252644 from the U.S. National Science Foundation, W911NF-09-1-0273 from the U.S. Army Research Office, and D11AP00282 from the U.S. Defense Advanced Research Projects Agency.
Authors' addresses: N. Esfahani (corresponding author), Google Inc., 1600 Amphitheatre Pkwy, Mountain View, CA 94043; email: naeeem@google.com; E. Yuan and K. R. Canavera, Department of Computer Science, George Mason University, 4400 University Drive, Fairfax, VA 22030; emails: {eyuan, kcanaver}@gmu.edu; S. Malek, University of California, Irvine, 5226 Donald Bren Hall, Irvine, California 92697; email: malek@uci.edu.

DOI: http://dx.doi.org/10.1145/2856035

Such management often takes the form of dynamically changing the structure of the base-level software, for example, replacing software components at runtime.

To make proper decisions, the metalevel subsystem relies on an abstract representation of the software and the environment it executes. The collection of such models is often referred to as *models at runtime*, as they need to be kept in sync with the changes that unfold in a running system and its environment. An example of architectural models that is used extensively in the construction of adaptive software is *component interaction model*, which represents the behavior of the system's components in their explicit interactions (e.g., message exchanges, interface invocations) with one another.

Component interaction models could be used for a variety of purposes in runtime management of software, including (1) determining the dependencies among the system's component to ensure their adaptation (e.g., replacement) does not leave the system in an inconsistent state [Canavera et al. 2012], (2) detecting abnormal interactions among the system's components that are indicative of security attacks to enable self-protection capabilities [Yuan et al. 2014a], and (3) optimizing a software system's performance by collocating components that are highly interactive with one another [Malek et al. 2012]. The construction of such models, however, is a difficult task. First, in a complex software system, manually defining models that represent the component interactions is time consuming. Second, it is not always possible to construct such models *a priori*, before the system's deployment (i.e., during the development phase). In Service-Oriented Architectures (SOA) or peer-to-peer environments, for instance, component behavior may be user driven and nondeterministic. Third, even when such models are built, it is a heavy burden to keep them in sync with the actual implementation of the software. Indeed, they are susceptible to the well-studied problem of architectural decay [Taylor et al. 2009], which tends to occur when changes applied to the software are not reflected in its architecture models.

An approach toward addressing the preceding issues is to automatically mine such models from execution traces of the system, thus alleviating the engineers from defining the models manually. Automated mining-based approaches also allow for their application throughout the system's execution, naturally enabling the refinement of models to changing behavior of the system and its environment.

Our preliminary progress at supporting an association rule mining approach that can learn the component interaction model of a system by simply observing its behavior was described in our prior work [Canavera et al. 2012; Yuan et al. 2014a]. This article builds on our prior work by providing a comprehensive, extended description of the approach as well as our experiences with applying it to an emergency response software system. The approach is comprised of three steps: (1) collect execution traces of the system at runtime, (2) use association rule mining to infer a probabilistic model of the component interactions from the collected execution traces, and (3) continuously monitor the accuracy of the inferred models, and upon detecting substantial variations, refine the models by mining the newly collected data.

We have used the component interaction models inferred using our approach to address three self-management concerns: (1) *Consistency of Adaptation*: safely applying dynamic changes to a running software system without creating inconsistencies; (2) *Self-Protection*: automatically identifying potentially malicious (abnormal) behavior; and (3) *Self-Optimization*: improving the performance of a software system by changing its deployment in a distributed setting. We first provide a detailed description of the first concern (i.e., ensuring consistency of adaptation), and subsequently outline the remaining two to illustrate the broader implications of our approach in the construction of self-managing software systems.

Using our approach in the construction of a self-managing emergency response system has shown to be quite promising. However, mining-based approaches, such as the

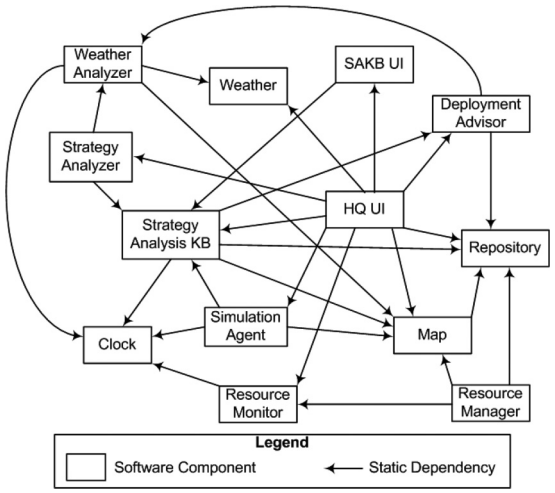

Fig. 1. Subset of EDS software architecture.

one described in this article, present their own unique challenges, which impact their widespread usage. Thus, we also provide an overview of these challenges to frame the future research.

This article extends our prior work [Canavera et al. 2012; Yuan et al. 2014a] along four dimensions: (1) presents a reference architecture for building self-managing software systems using the aforementioned approach; (2) presents a new instance of this architecture for self-optimization of deployment topology, including the corresponding evaluation results; (3) provides a more detailed, comprehensive description of our approach and reports on several new experiments; and (4) articulates the limitations and assumptions underlying our research, as well as an agenda for future research.

The remainder of this article is organized as follows. Section 2 introduces a distributed software system and its adaptation requirements to motivate the research. Section 3 provides an overview of our approach, while Section 4 describes how the approach can be applied in the context of a specific adaptation concern having to do with the consistency of runtime change. Sections 5 and 6 outline applications of our approach for solving two other concerns in runtime management of software. Finally, the article concludes with an overview of the related research in Section 7, and a discussion of the remaining challenges and avenues of future research in Section 8.

2. MOTIVATING EXAMPLE

We illustrate the concepts and evaluate the research using a software system, called Emergency Deployment System (EDS), which is intended for the deployment and management of personnel in emergency response scenarios. Figure 1 depicts a subset of EDS's software architecture, and in particular shows the dependency relationships among its components. EDS is used to accomplish four main tasks: (1) track the emergency operation resources (e.g., ambulances, rescue teams, etc.) using Resource Monitor, (2) distribute resources to the rescue teams using Resource Manager, (3) analyze different deployment strategies using Strategy Analyzer, and finally (4) find the required steps toward a selected strategy using Deployment Advisor. EDS is representative of a large component-based software system, where the components communicate by exchanging messages (events). In the largest deployment of EDS to date, it was deployed on 105 nodes and used by more than 100 users [Malek et al. 2005].

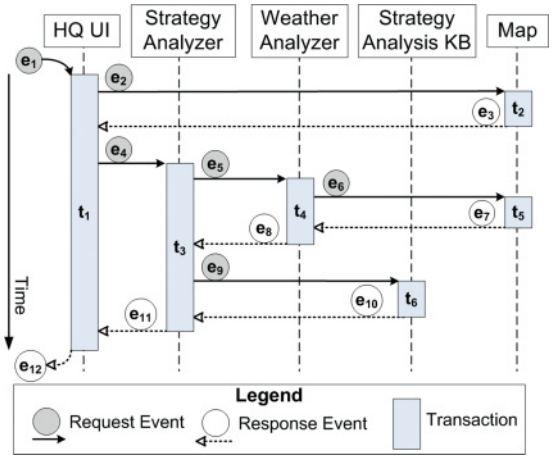

Fig. 2. EDS use case example.

Like any software system, the EDS functionality can be decomposed into a number of use cases. The sequence diagram for one such use case, conducting strategy analysis, is shown in Figure 2 as an example. We see that the execution of the use case involves a sequence of interactions among different software components. Note that a component here represents a coarsely grained software unit that deploys and runs independently from other components (in contrast to lower level entities such as a Java object or a code library). For instance, a component could be an Enterprise JavaBean or a web application that resides on a web server.

Once deployed and operational, a real-world system such as EDS needs to continually evolve to ensure quality, meet changing user requirements, and accommodate environment changes (such as hardware upgrades). The system must satisfy a number of architectural objectives such as availability, performance, reliability, and security.

Nontrivial system adaptations typically require an abstract representation of the components and their interactions at runtime, which can be used to formulate adaptation strategies and tactics [Garlan et al. 2004]. In the case of EDS, a model such as Figure 2 could be used to reason about several adaptation concerns: (1) The model tells us when it is safe to adapt the components. For instance, as shown in Vandewoude et al. [2007], a model such as that of Figure 2 could be used to determine Strategy Analyzer can be safely adapted prior to event e_4 or after event e_{11}, but not in between, as its state is inconsistent. (2) The model could be used in the construction of self-protecting software to detect abnormal (malicious) behavior. For instance, assuming that the model of Figure 2 represents the only possible sequence of interaction among the components under this use case, one could determine a suspicious behavior when Strategy Analyzer interacts with a component it has not previously interacted with, such as Resource Manager. (3) The model could be used in the construction of self-optimizing software by changing the deployment of software, that is, allocation of software components to the system's hardware nodes. For instance, as shown in Malek et al. [2012], to reduce the response time, components that interact frequently could be either collocated on the same hardware node or on nodes that have reliable and fast network connectivity.

Building and maintaining such a component interaction model, however, faces several difficult challenges, as outlined in Section 1. Our approach to addressing these challenges involves learning a *usage proximity* model of dynamic component interactions at runtime, without any predefined behavior specifications. Machine-learning-based approaches alleviate engineers from maintaining the models manually, and also

allow for their automatic adaptation and refinement to changing behavior of the system and its environment.

3. APPROACH OVERVIEW

We first start with some definitions and assumptions to frame the discussion of our mining approach. An event e is defined as a triple tuple $e = <src, dst, time>$, where *src* and *dst* are identifiers for the source and destination components, and *time* is the timestamp of its occurrence. Although an event is also likely to have a payload (e.g., a message in XML format), it is not relevant to this line of research, and thus not modeled. In the EDS example of Figure 2, 12 events (e_1–e_{12}) are depicted.

A transaction t is defined as a triple tuple $t = <start, end, R>$, where *start* and *end* respectively represent the events initiating and terminating the transaction t, while R is a set of transactions that subsequently occur as a result of t. $R \neq \emptyset$ when t is a dependent transaction (e.g., t_1, t_3, and t_4 in Figure 2), and $R = \emptyset$ when t is an independent transaction (e.g., t_2, t_5, and t_6 in Figure 2).

A *top-level transaction* t is a kind of transaction where there is no other transaction x in the system such that $t \in x.R$. In other words, a transaction is top level if its occurrence is not tied to other transactions in the system. A top-level transaction corresponds to the system's use cases (functional capabilities). For instance, t_1 in Figure 2 is a top-level transac980tion, initiated in response to e_1, which represents the user requesting a service from the system.

In this example, we see that the components involved in a use case interact closely with one another. Given enough observations of the system at runtime, it is possible to infer the *stochastic component interaction model* of the system. Such a model not only infers the dynamic dependencies among the components (i.e., information equivalent to that captured in Figure 2), but it also provides a probabilistic measure of the certainty with which events and transactions may occur. Even though such a model is simplistic and by no means captures the complete and precise behavior of the system, it is surprisingly useful in addressing a number of adaptation objectives as we shall see in later sections.

To keep our approach widely applicable, we make minimal assumptions about the available information from the underlying system:

—*Black-Box Treatment*: We assume the software components' implementation is not available. This allows our approach to be applicable to systems that utilize services or commercial off-the-shelf (COTS) components, whose source code is not available. It also enables our approach to naturally support the evolution of software components.
—*Observability of Event*: We assume that events marking the interactions among the system's components are observable. An event could be either a message exchange or a method call, which could be monitored via the middleware facilities that host the components or instrumentation of the communication links.
—*Observability of Transaction Duration*: We assume events *start* and *end*, which as you may recall indicate beginning and termination of a transaction, to be observable. This is a reasonable assumption consistent with several prior research approaches that have dealt with safely effecting runtime changes [Kramer and Magee 1990; Vandewoude et al. 2007; Ma et al. 2011].
—*Top-level transactions can be identified*. Here we assume that a number of "entry point" events exist that initiates top-level transactions. Such events typically represent the starting point of a system use case. An online banking system, for example, may have menu items such as "Withdrawal," "Deposit," or "Check Balance" that trigger different use cases. The EDS system, likewise, has client-server events (such as e_1 in Figure 2) that initiate different use cases.

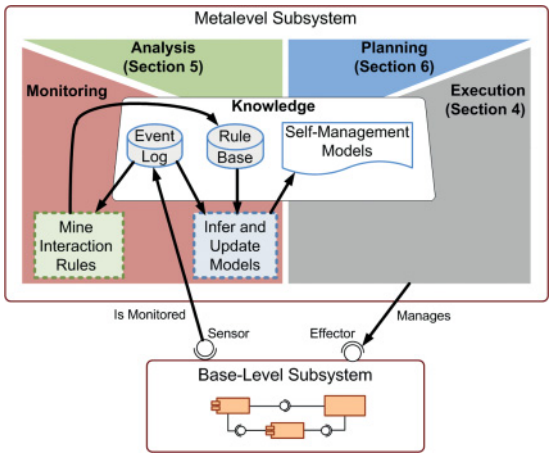

Fig. 3. Overview of our approach in the context of MAPE-K reference architecture.

With these assumptions, we proceed to define a novel approach for automatically deriving the stochastic component interaction model by mining the execution history of the software system. Figure 3 provides an overview of our approach in the context of the well-known MAPE-K reference architecture [Kephart and Chess 2003]. Conceptually, our approach belongs to the Monitoring component of the MAPE-K architecture. The goal of our research is to use the data collected from the execution of the system to infer useful models for self-management of software. Our approach consists of two high-level components: *Mine Interaction Rules* and *Infer and Update Models*.

The *Mine Interaction Rules* component receives the *Event Log* of the system as input and produces *Transaction Association Rules (TARs)* relating the relationship between transactions that are occurring in the system and those that may happen in the future. In essence, a TAR is a probabilistic rule representing the interaction behavior of components comprising the system. This component also continuously compares the predictions based on the inferred rules against the actual execution of the software. Upon detecting an unacceptable level of error in the predictions, it discards the old rules and learns a new set of rules given the latest collection of data.

The *Infer and Update Models* component takes the generated rules together with the events that are occurring in the system to infer a suitable model of the base-level subsystem for the purposes of self-management. It also ensures this model remains up to date as it detects changes in the execution of the base-level software. These models are intended for use by the other three phases of the MAPE cycle, that is, Analysis, Planning, and Execution. The nature of the self-management model produced by this component differs based on the nature of self-management concern. We provide three examples of self-management models that have been inferred using such an approach in our research and describe the challenges of building such models in Section 8.

First, in the next section, we describe in detail how our approach can be used to infer the necessary models for balancing the trade-off between disruption and reachability, while avoiding inconsistencies that may arise when changes are effected in a running software system. The resulting solution is dubbed MOSAIC (Mining Of Safe Adaptation Intervals for Component-based Software). MOSAIC illustrates an example of a self-management model that is intended for use by the Execution phase of the MAPE cycle. Afterward, we describe two more applications of our approach; one for detecting potentially malicious behavior at runtime (Section 5), and the other for self-optimizing a software system's performance through redeployment of its components

(Section 6). These two other examples help illustrate the applications of our approach in the Analysis and Planning phases of the MAPE cycle.

4. SAFE COMPONENT ADAPTATION

Replacing a component in the middle of a transaction could place the system in an inconsistent state. Consider a situation in which the *Strategy Analyzer* component of Figure 2 is replaced after sending the request event e_5, but before receiving the response event e_8. Since the newly installed component does not have the same state as the old one, it may not be able to handle response e_8 and subsequently initiate transaction t_6 via event e_9, resulting in an inconsistency and potentially the system's failure. Three general approaches to this problem have been proposed: *quiescence*, *tranquility*, and *version consistency*.

Quiescence [Kramer and Magee 1990] is the established approach for safe adaptation of a system. A component is in quiescence and can be adapted if (1) it is not *active*, meaning it is not participating in any transaction, and (2) all of the components that may initiate transactions requiring services of that component are passivated. A component is *passive* if it continues to receive and process transactions, but does not initiate any new ones. At runtime, the decision about which part of the system should be *passivated* is made using a *static component interaction model*, such as that shown in Figure 1. For instance, to change the *Map* component, on top of passivating itself, *Weather Analyzer*, *Strategy Analysis KB*, *HQ UI*, *Simulation Agent*, and *Resource Manager* components need to be passivated as well, since those are the components that may initiate a transaction on *Map*.

While quiescence provides consistency guarantees, it is very pessimistic in its analysis and, therefore, sometimes very disruptive. Consider that the static interaction model includes all possible dependencies among the system's components, while at any point in the execution of a software system only some of those dependencies take effect. To address this issue, tranquility [Vandewoude et al. 2007] proposes to use the *dynamic component interaction model* of a system in its analysis, an example of which is shown in Figure 2. Under tranquility *a component can be replaced within a transaction as long as it has not already participated in a transaction that it may participate in again*. For instance, under tranquility, *Map* could be replaced either before it receives event e_2 or after it sends event e_7, but not in between.

A shortcoming of tranquility, as realized in Vandewoude et al. [2007], was lack of support for handling dependent transactions. This issue was addressed in version consistency [Ma et al. 2011], which guarantees a dependent transaction is served by either the old version or a new version of a component that is being changed.

Figure 4 depicts the high-level overview of MOSAIC, which is an instance of the overarching approach depicted in Figure 3. It shows the steps comprising the *Mine Interaction Rules* and *Infer and Update Models* of Figure 3.

The *Mine Interaction Rules* activities, shown on the left, start with the *Construct Itemsets* activity processing the *Event Log* of the system to construct a large number of *Itemsets*. An itemset indicates the events that occur close in time. Itemsets are then passed through a data mining algorithm to derive *TARs* relating the relationship between transactions that are occurring in the system and those that may happen in the future. Since mining may generate a large number of rules, some of which may be invalid and redundant, we *Prune Rules* using heuristics to arrive at a small number of predictive rules that can be applied efficiently at runtime.

The *Infer and Update Models* activities, shown on the right in Figure 4, start with the *Track Active Transactions* activity monitoring the currently running transactions in the system. *Select Relevant TARs* then uses the information about currently active transactions to pick a set of candidate TARs from the *Rule Base* for estimating the

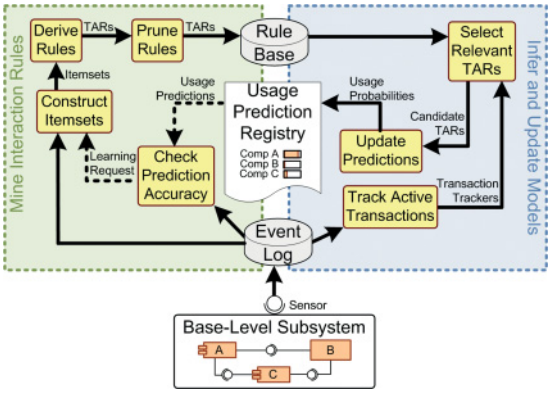

Fig. 4. Overview of MOSAIC.

usage probability of components. *Update Predictions* uses candidate TARs to update the *Usage Prediction Registry*, which is the model used by the Execution phase of the MAPE cycle for determining whether a component should be adapted or not. Essentially, it is a data structure that contains the up-to-date usage predictions for the components in the system. The usage prediction for each component is the probability that the component will imminently be used as a result of the transactions running in the system. These predictions can be calculated either continuously or on an as-needed basis.

Finally, as indicated by *Check Prediction Accuracy*, the predictions are scrutinized at runtime, and if they go above an unacceptable threshold, a new round of mining based on the newly collected log of events is initiated. As a result of a new round of mining, old rules are discarded and a new set of rules are adopted for use. This allows the approach to incorporate changes due to how the software is used or its evolution into the mining process. In the following sections, we describe the details of MOSAIC.

As discussed in detail next, using the usage predictions at runtime, MOSAIC is able to determine "safe" times to adapt a component, in the sense that the usage predictions indicate whether disruption would likely occur if adaptation were to take place for a particular component. Beneficially, MOSAIC is able to use the same minimal assumptions as quiescence in order to provide inconsistency-free adaptation, while also minimizing disruptions that occur as a result of adaptation. While MOSAIC may be deployed independently, one beneficial implementation allows deployment of MOSAIC alongside a quiescence implementation. This approach has the benefit of allowing safe adaptation when disruption-reduced or -free adaptation is possible (using MOSAIC), while allowing fallback to quiescence and its forced disruptions when a high level of usage does not allow disruption-reduced or -free adaptation.

4.1. Mine Interaction Rules

This section describes the *Mine Interaction Rules* activities (recall Figure 4). These activities run asynchronously, separate from the system's execution, and potentially on a different platform. It may repeat throughout the system's execution to adjust the model to the evolving behavior of the software system.

4.1.1. Event Log. Mining operates on an *Event Log* of the system, which represents an execution history of the system for a sufficiently long period of time to be truly representative of how the system is used. Clearly, MOSAIC is not applicable to systems where such a history cannot be collected, or the system's past behavior is not indicative of its future, but we believe most systems do not fall in this category. Since our objective

is to infer the relationship among the transactions, we would like mining to operate on a representation that is in terms of transactions as opposed to events. As a result, the *Event Log* of the system is automatically processed to determine all of the transactions that have occurred by pairing the *start* and the *end* events for each transaction. Recall from the preamble of Section 4 that consistent with the prior work, we assume these types of events are observable and could be used to identify the occurrence of transactions. From this point forward, we will mainly focus on transactions, though the reader should be aware of the relationship to the events.

4.1.2. Constructing Itemsets. The first step to mining the relationship among the transactions is to *Construct Itemsets* (see Figure 4). An *itemset*, as in the data mining literature for association rule mining, is a set of items that have occurred together. In the context of our research, an itemset I is a set of transactions that have occurred temporally close to one another at some particular point during the execution of the system: $I = \{t_1, t_2, \ldots, t_n\}$.

The transaction records for the execution history of the system are transformed into itemsets through a simple process. A new itemset is formed for each top-level transaction, but not the transactions that those top-level transactions initiate. A top-level transaction is automatically detected if its beginning, end, or both do not fall within the beginning and end of another transaction. All other transactions are placed in the itemsets for the transactions whose beginning and end times fully surround the beginning and end times of the present transaction.

In reference to Figure 2, a new itemset would be created for t_1, as its beginning and end (determined by e_1 and e_{12}) do not fall within any other transactions. All the remaining transactions t_2, t_3, t_4, t_5, and t_6 are added to I_{t1} itemset as follows: $I_{t1} = \{t_1, t_2, t_3, t_4, t_5, t_6\}$.

Using this process, an entire segment of a software system's execution history can be transformed into a list of itemsets representing the occurrence of transactions together in time. This list, which contains all the itemsets, is conveniently called *Itemsets*. MOSAIC is able to mitigate the impact of concurrently running top-level transactions by using a sufficiently large usage history. Consider a version of the scenario depicted in Figure 2 in which a second top-level transaction t_7 overlapping partially in time with t_1 starts and itself initiates a transaction t_8 that falls wholly within the beginning and end times of both t_1 and t_7. MOSAIC would include t_8 in both I_{t1} and I_{t7}. However, since transactions t_1 and t_7 are truly independent, the false placement of t_8 in I_{t1} is a random event that is not likely to occur in a significantly large number of itemsets, and thus safely ignored by the algorithm using minimum frequency thresholds. Section 4.4.2 discusses the effectiveness of this technique. While concurrency undoubtedly deteriorates the effectiveness of the approach, the results discussed in Section 4.4.2 demonstrate that the approach is generally effective and demonstrates no notable concurrency-based deterioration until extremely high levels of concurrency are encountered.

After constructing the itemsets, we trim the notion of time from the transactions (recall *time* in the definition of e in Section 3). This essentially transforms transactions into transaction types. In other words, we only use time for forming itemsets and from now on we only use the type of transactions. However, for the sake of simplicity, we keep referring to the transaction types as transactions, in the rest of this section.

4.1.3. Deriving Rules. Several data mining approaches [Han and Kamber 2006] can be used to perform learning on the set of itemsets constructed this way. We found the association rule mining class of algorithms to be the most suitable for our purposes. The output of an algorithm of this type for our problem is a set of *TARs*. TARs are probabilistic rules for predicting the occurrence of transactions as follows: $X \rightarrow Y : p$

A TAR states that the occurrence of a set of transactions X implies the occurrence of a set of transactions Y with probability p. As shown in Figure 4, TARs derived in this way are eventually stored in the *Rule Base* for use during the system's adaptation at runtime.

For association rule mining algorithms, an appropriate value for p is the *confidence* of the implication $X \rightarrow Y$. Confidence is defined as

$$p = \left(\sum_{i=1}^{|Itemsets|} \begin{cases} 1 & \text{if } X \subseteq I_i \wedge Y \subseteq I_i, \\ 0 & \text{otherwise.} \end{cases} \right) \Big/ \left(\sum_{i=1}^{|Itemsets|} \begin{cases} 1 & \text{if } X \subseteq I_i, \\ 0 & \text{otherwise} \end{cases} \right).$$

Confidence is an appropriate metric for p in TARs because it provides a measure of the strength of the implication $X \rightarrow Y$. TARs with strong relations between X and Y have a high confidence value, while TARs with weak relations between X and Y have a low confidence value.

Another metric that is commonly generated by data mining algorithms during the learning phase is *support*:

$$s = \left(\sum_{i=1}^{|Itemsets|} \begin{cases} 1 & \text{if } X \subseteq I_i \wedge Y \subseteq I_i, \\ 0 & \text{otherwise} \end{cases} \right) \Big/ |Itemsets|.$$

While support is not appropriate for the value of prediction probability in TARs (i.e., p in TARs definition), it is useful in that it provides a measure of the frequency with which X and Y occur together. As such, we use a minimum support value during the mining phase in order to filter out rare relationships that represent outliers in the general usage of the system. Thus, the errors introduced in itemsets due to concurrent execution of transactions in the system (recall Section 4.1.2) can be filtered out effectively using a minimum support and confidence threshold.

While the mining algorithm in the *Derive Rules* activity produces logically accurate TARs, it typically produces an excessively large number of TARs, some of which are not useful. As such, the generated rules must be pruned to make them suitable for use at runtime. As shown in Figure 4, the *Derive Rules* step terminates by passing the raw set of generated TARs to *Prune Rules*.

4.1.4. Pruning the Rule Base. An excessively large number of TARs is produced as a result of the *Derive Rules* activity, because we set the minimum confidence for a TAR to be very small, that is, we do not filter out many TARs based on the confidence level. While a TAR with a small p expresses less confidence in the prediction than another TAR with a larger p, both predictions are accurate and can be used in unison as explained in Section 4.2.3. We take this approach because a TAR with a small confidence level may represent a valid transaction that rarely happens; thus, we should not categorically exclude this information.

In addition, many of the unnecessary TARs are produced because the data mining algorithm and its input (i.e., itemsets) do not fully incorporate all of the knowledge that we have about the system. For instance, itemsets are unordered and thereby the resulting TARs incorporate no ordering information. As a result, the mining algorithm produces an excessively large number of TARs that are not useful.

Since we would like to use the rules at runtime, we need to prune them to a subset of highly predictive rules that can be applied efficiently at runtime. To that end, and as depicted in Figure 4, the *Derive Rules* step terminates by passing the raw set of generated TARs to *Prune Rules*. There are three effective heuristics that we have developed for pruning the TARs. While these heuristics are not exhaustive in removing all redundancy in the *Rule Base*, they have proven effective in producing a manageable set of TARs to apply at runtime.

(1) Redundant TAR Pruning Heuristic: Consider TARs satisfying this pattern:
$TAR_1 : X_1 \rightarrow Y : p_1$,
$TAR_2 : X_2 \rightarrow Y : p_2$,
where $(X_2 \subseteq X_1) \wedge (p_1 \approx p_2)$.
In this scenario, TAR_1 and TAR_2 predict the same set of transactions and at the same level of confidence. However, the conditions for satisfying TAR_2 is a subset of those for TAR_1, that is, X_2 is a subset of X_1. As will be explained in Section 4.2.2, a TAR's conditions are considered to be satisfied, when the transactions comprising its left-hand side have been observed. Therefore, TAR_1 and TAR_2 predict the same exact outcome, except TAR_2 requires fewer conditions to be satisfied. We can safely prune TAR_1, since it is redundant.

(2) Less Specific TAR Pruning Heuristic: Consider TARs satisfying this pattern:
$TAR_1 : X \rightarrow Y_1 : p_1$,
$TAR_2 : X \rightarrow Y_2 : p_2$,
$TAR_3 : X \rightarrow Y_3 : p_3$,
where $(Y_1 = Y_2 \cup Y_3)$.
In this scenario, TAR_1 makes a composite prediction of TAR_2 and TAR_3. All three TARs are satisfied with the observation of the same set of transactions X. However, because $Y_1 = Y_2 \cup Y_3$, TAR_1 is a composite prediction of the more specific predictions made by TAR_2 and TAR_3. Given the definition of confidence and its use as the prediction value p, the prediction value p_1 for TAR_1 will always be weaker (lower) than the prediction values of p_2 and p_3 for TAR_2 and TAR_3, respectively. As a result, TAR_1 is a less specific rule and can be pruned.

(3) Misordered TAR Pruning Heuristic: We can also prune rules by incorporating our knowledge of what constitutes a valid behavior. We can prune $TAR : X \rightarrow Y : p$, where $\exists x \in X \wedge y \in Y : x.start.src = y.end.dst$. In this kind of TAR one of the predicted transactions in Y has as its destination the source of one of the observed transactions in X. Therefore, the TAR is useless because it predicts the use of a component that must have already been used. It is important to note that, while this type of TAR seems illogical and perhaps presumptively unlikely to be generated, the association rule mining algorithm and its input (i.e., itemsets) do not recognize any transaction ordering. Furthermore, these types of TARs can be highly predictive and are very common. Essentially they predict that the transaction necessary for another transaction to occur will in fact occur with that transaction. Therefore, this pruning step removes many useless rules and has the largest impact in MOSAIC.

At the completion of this activity a subset of generated rules remains, which is stored in the *Rule Base* and used for runtime prediction of component usage. In EDS, these heuristics were able to reduce the number of rules to one-tenth without losing their expressive power. Section 4.4.1 evaluates rule reduction in the context of EDS.

4.2. Infer and Update Model

We now describe the activities comprising the *Infer and Update Model* from Figure 4.

4.2.1. Tracking Active Transactions. *Track Active Transactions* step processes any observed event $t_o.start$ and $t_o.end$, indicating the beginning and termination of transaction t_o, respectively. To that end, we use a data structure, called *Top-Level Tracker*, and represented as set *TLT*, for each top-level transaction active (i.e., currently running) in the system. The purpose of TLTs is to keep account of the present transaction activity in the system.

Upon observing $t_o.start$, the state of TLTs is updated as follows. If t_o is a top-level transaction, a new TLT is created. But if t_o is not a top-level transaction, its identifier is added to all open TLTs, that is, t_o is associated with every top-level transaction that may have caused it. This is done because there is no way of knowing which top-level

transaction has actually initiated this transaction. Upon observing $t_o.end$, if t_o is not a top-level transaction, it is ignored. On the other hand, if t_o is a top-level transaction, then the TLT corresponding to t_o is closed.

Changes to TLTs impact the *Usage Prediction Registry*. In the following subsections, we describe the process assuming $t_o.start$ has been observed, but revisit the situation in which $t_o.end$ is observed before concluding.

4.2.2. Selecting the Relevant Rules. The updated TLTs are used to determine what new predictions can be made about the probability with which components will be used. All predictions of the system activity are made by using the TARs stored in the *Rule Base*. We must determine what new TARs, if any, are implicated by the observation of $t_o.start$.

To that end, we iterate over all TARs in the *Rule Base*. A $tar \in RuleBase$ can only be implicated by the observation of $t_o.start$, if t_o is a member of set X of that tar. That is to say, we cannot make a new prediction based on the given tar, unless t_o contributes to the prediction. If this criterion is met, then we look to see if the tar is satisfied by any open top-level transaction as tracked by TLTs. For a tar to be satisfied, all transactions in X must have been observed during the processing of at least one TLT. Furthermore, the tar's prediction (i.e., Y) should have new transactions other than the ones that have already occurred during the processing of the satisfying TLT. Stated differently, the tar is only considered to have a useful prediction if (1) all of its prerequisites have been seen, and (2) at least some of its predictions are unseen. If both of these conditions are met, then the tar is added to the set $CTAR$, which is a set of all new TARs that are candidates for being applied at that given point in time.

The TLT that satisfies the conditions for the presence of a tar in CTAR is said to be a *basis* for the application of that tar. This basis information is tracked along with the tar and used in the next stages.

4.2.3. Updating the Usage Prediction Registry. The next step is to apply the implicated TARs to update the *Usage Prediction Registries*. Given a component c, there are typically more than a single TAR predicting its usage probability. While some may be due to the new observation $t_o.start$, others may be due to the prior observations. Therefore, we must combine the various p values from all of the satisfied TARS into a single prediction value u_c. In the following discussion, we will use an example with $c = Clock$.

Before describing how u_c can be calculated, we need to define three sets: (1) $CTAR_c$ is a set of candidate TARs that are supposed to affect a given component c and defined as $CTAR_c = \{tar | tar \in CTAR \wedge (\exists t \in tar.Y : t.start.dst = c)\}$. These are the new TARs based on the observation $t_o.start$. (2) $ATAR_c$ is the set of active TARs currently contributing to u_c due to observations made prior to $t_o.start$. (3) Finally, $PTAR_c = CTAR_c \cup ATAR_c$ is the complete set of TARs that determine the new value of u_c.

As an example, consider that $t_o.start = <StrategyAnalyzer, StrategyAnalysisKB>$.[1] Let us assume that we have $ATAR_{Clock} = \{tar_1 : x_1 \rightarrow y_1 : 0.5, tar_2 : x_2 \rightarrow y_2 : 0.3\}$, where

$x_1 = \{<HQUI, StrategyAnalyzer>, <StrategyAnalyzer, WeatherAnalyzer>\}$,
$y_1 = \{<WeatherAnalyzer, Clock>\}$,
$x_2 = \{<HQUI, DeploymentAdvisor>, <DeploymentAdvisor, WeatherAnalyzer>\}$,
$y_2 = \{<WeatherAnalyzer, Clock>\}$.

This is to say, prior to observation of the present $t_o.start$, two TARs were predicting the usage of $Clock$ and u_{Clock} was 0.65.

[1]For the sake of brevity, in this example, we are dropping the *time* value of the events and depict transactions only by their start events.

Based on the observed $t_o.start$, $CTAR_{Clock}$ would be the set of all TARs in $CTAR$ such that $<StrategyAnalyzer, StrategyAnalysisKB> \in tar.X$ and $<some\_component, Clock> \in tar.Y$. For this example, let us assume that we have $CTAR_{Clock} = \{tar_3 : x_3 \rightarrow y_3 : 0.6, tar_4 : x_4 \rightarrow y_4 : 0.3, tar_5 : x_5 \rightarrow y_5 : 0.2\}$, where

$x_3 = \{<HQUI, StrategyAnalyzer>, <StrategyAnalyzer, StrategyAnalysisKB>\}$,
$y_3 = \{<StrategyAnalysisKB, Clock>\}$,
$x_4 = \{<StrategyAnalyzer, StrategyAnalysisKB>\}$,
$y_4 = \{<StrategyAnalysisKB, Clock>\}$,
$x_5 = \{<HQUI, DeploymentAdvisor>, <StrategyAnalyzer, StrategyAnalysisKB>\}$,
$y_5 = \{<DeploymentAdvisor, WeatherAnalyzer>, <WeatherAnalyzer, Clock>\}$.

We can now describe how u_c is calculated in five steps:

(1) Removing duplicate TARs: We do not need to reconsider a $tar \in CTAR_c$, which is already actively predicting the usage of component c (i.e., $tar \in ATAR_c$). Therefore, we remove any such tar from $CTAR_c$ (i.e., $CTAR_c = CTAR_c - ATAR_c$).

In our example, $PTAR_c$ contains no duplicates (all five TARs are different), so no modifications are made to the sets at this step.

(2) Removing superseded TARs: A superseding relationship occurs when we have TARs satisfying this pattern:

$TAR_1 : X_1 \rightarrow Y : p_1$,
$TAR_2 : X_2 \rightarrow Y : p_2$,
where $(X_1 \subseteq X_2)$.

In this scenario, TAR_2 predicts the same set of transactions as TAR_1, however, TAR_2 makes use of more information than TAR_1 and hence makes a more informed prediction. Therefore, TAR_1 is removed from its set (i.e., either $CTAR_c$ or $ATAR_c$, depending on which one it came from).

In our example, tar_3 supersedes tar_4. In particular, tar_3 predicts a more specific execution path to the execution of $<StrategyAnalysisKB, Clock>$, so tar_3 makes a more informed prediction. As such, tar_4 is removed from $CTAR_{Clock}$, leaving us with $ATAR_{Clock} = \{tar_1, tar_2\}$ and $CTAR_{Clock} = \{tar_3, tar_5\}$.

(3) Selecting the best candidate: Even after removing the redundant rules, we may still have some partially overlapping ones. Partially overlapping rules express the various execution paths that may eventually result in the use of same component. Consider the following two TARs:

$TAR_1 : \{t_1, t_o\} \rightarrow \{t_3, t_c\} : p_1$,
$TAR_2 : \{t_2, t_o\} \rightarrow \{t_4, t_c\} : p_2$,
where $(t_c.start.dst = c)$.

Since $TAR_1.Y \neq TAR_2.Y$, the superseding relationship cannot be used to remove one of the TARs. However, the observation of a single $t_o.start$ should at most result in a single prediction for the component c. We use a heuristic and choose the TAR with the highest p value to be the best candidate. This TAR expresses the greatest risk that c will be used. After this step, $CTAR_c$ must have a single member.

In our example, tar_3 has the highest p value at 0.6. Therefore, tar_5 is removed from $CTAR_{Clock}$, leaving us with $ATAR_{Clock} = \{tar_1, tar_2\}$ and $CTAR_{Clock} = \{tar_3\}$.

(4) Trimming $PTAR_c$: Analogous to the logic in the previous step, it is reasonable to expect each top-level transaction to make a single prediction for a component c. When there is more than one active top-level transaction, we cannot know with certainty which top-level transaction actually initiated $t_o.start$. However, based on the number of active TLTs (recall Section 4.2.1), we know *how many* top-level transactions are active in the system when $t_o.start$ is observed. Therefore, we approximate by limiting the number of TARs contributing to u_c to the number of top-level transactions active at that point in time. As with the reduction of $CTAR_c$ in the previous step, we choose to be conservative by keeping the TARs with the highest p values. We remove the TARs

with the lowest p value from $PTAR_c$ until the size of $PTAR_c$ is equal to the number of active TLTs.

In our example, $PTAR_{Clock}$ has three members: tar_1, tar_2, and tar_3. However, because $ATAR_{Clock}$ only had two members prior to observation of $t_o.start$, we know that there are only two TLTs and thus two active top-level transactions. As such, $PTAR_{Clock}$ is now limited to the two TARs with the highest p values. As such, tar_2 is removed from $PTAR_{Clock}$, leaving us with $PTAR_{Clock} = \{tar_1, tar_3\}$.

(5) Combining the predictions: At this point, we let the $ATAR_c$ to be equal to $PTAR_c$. We can now recalculate u_c based on the updated $ATAR_c$. Because there are no duplicate, overlapping, or related TARs in $ATAR_c$, we calculate u_c by combining the prediction values from individual TARs in $ATAR_c$ as independent probabilities:

$u_c = 1 - probability\ c\ is\ not\ used = 1 - \prod_{i=1}^{|ATAR_c|}(1 - p_i)$.

This follows from the fact that according to each $TAR_i \in ATAR_c$, the probability of c not being used as a result of the relationship modeled in TAR_i is $1 - p_i$.

In our example, the new $ATAR_{Clock}$ is $PTAR_{Clock} = \{tar_1, tar_3\}$. Therefore, we can calculate $u_{Clock} = 1-(1-0.5)*(1-0.6) = 1-(0.5)*(0.4) = 1-0.2 = 0.8$. Therefore, the new *Usage Prediction Registry* value for *Clock* is 0.8, up from the previous value of 0.65.

So far, we explained how the *Usage Prediction Registries* are updated when $t_o.start$ is observed. However, the observation of $t_o.end$ can also update the *Usage Prediction Registries*. If $t_o.end$ is a top-level transaction, the tlt_o corresponding to $t_o.end$ is removed. As a result, all the TARs that have tlt_o as their only basis are removed from $ATAR_c$. Since in this case $CTAR_c = \emptyset$, steps 1–4 are skipped, and step 5 is performed to propagate the impact of these deletions on all of the components' predictions.

The *Usage Prediction Registry* is either updated each time a transaction and its corresponding events are observed, or on an as-needed basis.

4.3. Using Registry for Adaptation

The ultimate goal in our research is to use the models obtained through the mining approach for finding a proper time to execute the adaptation decisions. In particular, the predictive power of models allows us to achieve proper adaptation of components in the Execution phase of the MAPE cycle. The probabilistic rules inferred using MOSAIC collectively represent the stochastic dependency model of the system. Such a model could be used in the context of both tranquility [Vandewoude et al. 2007] and version consistency [Ma et al. 2011] for adaptation. In our current approach, we employ a technique similar to that described in tranquility, where we temporarily buffer (store) events intended for a component during the time it is being replaced. Alternatively, we could have employed a technique similar to that of version consistency, where two instances of a component are leveraged, and incrementally new top-level transactions are shifted to use the new version.

4.3.1. Guaranteeing Consistency. As specifically noted in the preamble of Section 4, inconsistency could result *if a component is adapted at a time in which it has already participated in a transaction that it participates in again*. That is to say, to maintain *consistency*, a component must not be adapted if it has been used in some top-level dependent transaction *until that top-level dependent transaction terminates.* Our predictions would approximate that type of protection, given that a component that is used typically ends up with a high usage prediction in its register and that value will not dissipate until the top-level transaction that caused it terminates. However, there is a slight risk that MOSAIC as described up to this point would not fully guarantee consistency, because one cannot guarantee the accuracy of mined rules.

In situations where such a risk is unacceptable, we make a slight modification to MOSAIC described in Section 4.2.3 that allows us to provide consistency guarantees.

When we observe a transaction t_o, which engages component c (i.e., $t_o.start.dst = c$), we lock the value of $u_c = 1$, since we now know c has participated in a transaction, and changes to it may result in inconsistencies. Locking the value of u_c prevents c from being adapted. Once the top-level transaction that caused t_o finishes, we can remove the lock from the value of u_c to roll back to the mechanism described in Section 4.2.3 for updating u_c. However, as you recall from Section 4.2.1, we cannot determine which top-level transaction caused t_0 (i.e., we do not know the TLT). Therefore, to be on the safe side, we take the conservative approach and keep the u_c locked at 1 until all of the TLTs that are the basis of that observation have closed. In this way, once a component has been engaged, it will not be allowed to adapt until all top-level transactions that could have caused that engagement are complete. Further, because this technique is based on observed usage and not predicted usage, this technique guarantees consistency despite the probabilistic nature of the *Rule Base*.

Although expected to be rare, if a transaction is observed that reflects a component dependency that was not observed during *Mine Interaction Rules*, the approach makes the most conservative assumption that the transaction is a top-level transaction. This maintains the consistency guarantee even in the presence of previously unobserved component dependencies. Such an observation may also be used to initiate a new iteration of *Mine Interaction Rules*, as the observation demonstrates that the actual component dependencies of the system have evolved from those reflected in the *Rule Base*.

4.3.2. Disruption versus Reachability. When $u_c = 1$, we do not adapt c, since the change is likely to leave the system in an inconsistent state. However, when $u_c < 1$, c has not yet been in a top-level transaction, but could still be used at anytime in the future. If we adapt c, we may disrupt the system, as events sent to that component would be buffered until the adaptation has finished. To eliminate disruption, it is tempting to use $u_c = 0$ as a condition for adapting c. It may, however, take a long time for u_c to become 0 and this could create a *reachability problem*, that is, a situation in which one has to wait a long time, or even forever, before the condition for adaptation is met.

On the other hand, as you recall from Section 4.2.1, since we cannot determine the top-level transaction that caused a transaction t_o, it is very likely that we associate t_o with other top-level transactions that have not caused t_o, and hence, it will impact the usage prediction of many other components that are only involved in other top-level transactions. This conservative approach, which puts consistency at the forefront, results in overestimating the usage prediction. Therefore, the value of u_c would be nonzero most of the time, and hence, in practice, it is often reasonable to allow adaptation when $u_c < \epsilon$.

The value of ϵ impacts *disruption* and *reachability*. Disruption and reachability present a trade-off. We can either assign a very low value to ϵ, and wait until the system is not busy, or assign a very high value to ϵ, and just make the change. The first option has a limited disruption, however, reachability of the adaptation is not guaranteed as the system may constantly remain busy. The second option causes disruption in the system with the goal of making the adaptation reachable faster. The trade-off between disruption and reachability boils down to selecting the right value for ϵ. As described next, we take an approach that allows us to base this trade-off on the past behavior of the system in combination with a user understandable threshold specified by the user.

Figure 5 depicts a hypothetical example that shows how the registries behave in practice. A typical registry goes through this motion many times over the execution of the system: starting at 0 when a top-level transaction is initiated, rising as new observations are made and TARs are applied, and falling back to 0 once the top-level transaction has terminated. The steps in these functions represent the times at which

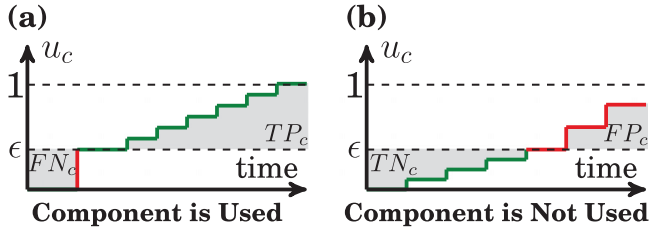

Fig. 5. Hypothetical behavior of u_c as $t_o.start$ is observed for some top-level transactions over time: (a) c is eventually used, and (b) c is not used.

the registries are updated. Finally, when the rules are accurate, we expect the step function to be skewed to the left (similar to $log(x)$ function) when the component is eventually used, and skewed to the right (similar to e^x function) otherwise. This is because typically when a component is eventually used, additional observations are made that subsequently satisfy more TARs, which combine to increase the component's usage probability.

As depicted in Figure 5, when a component has a $u_c \geq \epsilon$ at the time of adaptation decision and the component actually gets used before the end of that transaction (*active*), we say it is a True Positive (TP) result. When a component has a $u_c < \epsilon$ at the time of adaptation decision and the component is eventually used (active), we say it is a False Negative (FN) result. Similarly, False Positive (FP) and True Negative (TN) results can be defined when a component is not eventually used (*inactive*) as depicted in Figure 5(b). FN impacts the disruption in the system, as it depicts the period of time in which we allow adaptation, even if the component could be used. On the other hand, FP impacts reachability of safe interval, as it depicts the period of time in which we could allow adaptation, but refuse to do so.

The remaining challenge is how to pick a value for ϵ that is meaningful. We define ϵ in terms of another parameter r, which represents the tolerable rate of all adaptations that may result in disruption. Essentially, we ask the user of the system as to what is the acceptable level of disruption in the system. We then use the history of the system to find a right value for ϵ to maximize the reachability of adaptation within the boundary of acceptable disruption. We believe r is a reasonable threshold that can be specified by the user, for example, the user stating that on average no more than 0.05 (5%) of adaptations should result in a disruption. In essence, r is used to derive the value of ϵ, which is the basis of the trade-off between reachability and disruption.

To be able to calculate ϵ based on r, we have to relate a system-wide user-provided threshold defined by r to a component-specific threshold defined by ϵ. We do this from a probability distribution of prior predictions, embodied in the recorded u_c values, that were calculated in the past. Let U_a represent the set of all recorded predictions for components that were eventually used (active), and U_i represent the set of all recorded predictions for components that were eventually not used (inactive). In essence, U_a represents the set of all recorded u_c values corresponding to the step function of Figure 5(a) for all components in the system (we refer to these values as u_a). Similarly, U_i represents the set of all recorded u_c values corresponding to the step function of Figure 5(b) for all components in the system (we refer to these values as u_i). As a result, U_a indicates the situations in which adaptations could have possibly disrupted the system in the past.

Given U_a and U_i, it is possible to build the corresponding frequency distributions P_{U_a} and P_{U_i} as shown in Figures 6(a) and 6(b), respectively (though ϵ is not known when these are first built). Using conventional techniques [Bertsekas and Tsitsiklis 2008], we can derive the Cumulative Distribution Functions (CDFs) F_{U_a} and F_{U_i} from P_{U_a} and

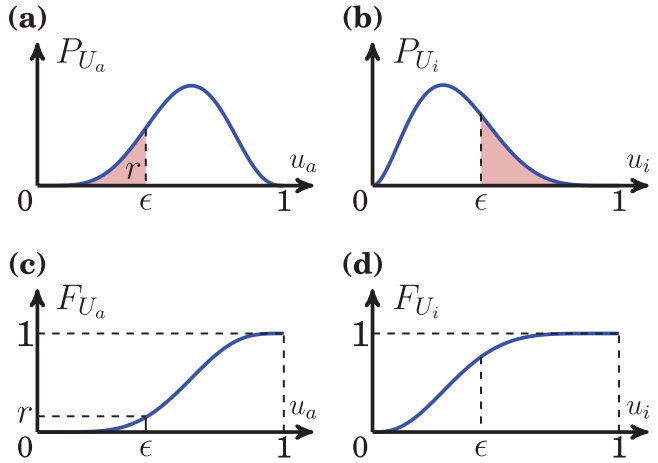

Fig. 6. (a) Frequency distribution for U_a, (b) frequency distribution for U_i, (c) CDF for U_a, and (d) CDF for U_i.

Table I. Experimental Systems Used in Evaluation, and Effects of TAR Pruning Heuristics

# of Users	# of TLT Observed	Concurrency Errors		# of TAR	
		Rate	Per Itemset	Initial	Remain
1	500	0.00%	0.00	38,582	1,683
10	1,628	1.69%	0.13	34,050	2,190
28	2,787	4.51%	0.35	38,248	2,331
40	3,330	10.94%	0.92	38,460	1,758
80	11,920	36.32%	4.19	35,168	3,126
137	3,543	60.77%	11.26	31,442	3,143

P_{U_i}, as depicted in Figures 6(c) and 6(d), respectively. $F_{U_a}(\epsilon)$ defines the fraction of all $u_a \in U_a$ samples where $u_a \leq \epsilon$. In other words, $r = F_{U_a}(\epsilon)$. Thus, we can calculate ϵ based on the r value specified by the user as the inverse $\epsilon = F^{-1}_{U_a}(r)$. In terms of probability theory, this means that ϵ is the r-quantile of the probability distribution [Bertsekas and Tsitsiklis 2008]. The CDF would need to be updated either periodically or as needed based on the execution history of the system.

It should be apparent from Figures 6(a) and 6(b) that the key to limiting error in MOSAIC is to skew P_{U_a} toward high values of u and P_{U_i} toward low values of u. This will result in F_{U_a} remaining at low values and then escalating quickly as it approaches 1.0, while F_{U_i} escalates quickly and then grows gradually to 1.0. This difference in F_{U_a} and F_{U_i} can be seen in Figures 6(c) and 6(d) based on the slight difference in skewing shown in P_{U_a} and P_{U_i} in Figures 6(a) and 6(b). If MOSAIC is able to skew the distributions for active and inactive components differently, then it effectively achieves the real goal of MOSAIC: it distinguishes between active and inactive components in advance.

4.4. Evaluation

We have developed a prototype of MOSAIC using *Apriori*—an association rule-mining algorithm with an implementation provided in WEKA [Hall et al. 2009]. As explained in Section 4.1.4, we intentionally use very low confidence and support thresholds: $p = 0.05$ and $s = 0.045$. We performed experimentation on runtime adaptation of EDS (recall Section 2). To evaluate MOSAIC, we used several versions of EDS as shown in Table I.

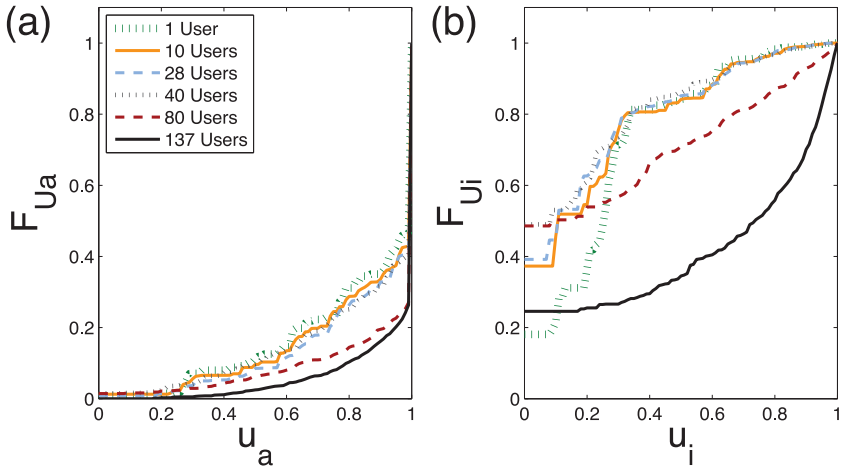

Fig. 7. The results from the experiments: (a) CDF of U_a and (b) CDF of U_i.

We used a baseline version of EDS with a single user. We then repeated the evaluations on higher concurrency systems to evaluate the susceptibility of MOSAIC to concurrency errors. The 80 and 137 experiments were simulated by using hyperactive dummy users, as EDS never naturally reached that level of concurrency error. Therefore, the values for users are merely projections, and the precise values for concurrency error rate should receive primary focus. The *Concurrency Errors* column in Table I shows what percentage of all recorded transactions were allocated to more than one top-level transaction due to concurrency, as well as the average number of these erroneously recorded transactions per top-level transaction. Across all experiments, an average of 7.47 transactions occurred as part of each top-level transaction, with minimum and maximum values of 7.27 and 7.78 occurring in the 137 and 1 experiments, respectively. Finally, to assess the accuracy and performance of MOSAIC under different conditions, the 80-user experiment was intentionally allowed to execute for a longer period of time, which resulted in collection of significantly more top-level transactions.

4.4.1. Effectiveness of TAR Reducing Heuristics. We first show the effectiveness of our rule pruning heuristics (recall Section 4.1). Significant reduction in TAR volume in the *Prune Rules* stage took place in all of the experiments. The reduction number can be seen in Table I. This reduction can only be truly appreciated when considered with two other facts: (1) the reduced rule base does not significantly degrade the accuracy as evaluated next, and (2) because of this reduction, the remaining rules can be applied very efficiently at runtime (evaluated in Section 4.4.4).

4.4.2. Accuracy of Component Usage Predictions. A crucial evaluation dimension for MOSAIC is the degree to which it correctly predicts the usage of a component. As discussed in Section 4.3.2, the accurate prediction is manifested through skewing F_{U_a} to a slow growth function that then escalates quickly at high values of u_a, while at the same time skewing F_{U_i} to a quickly escalating function that then grows only gradually over high values of u_i. Figures 7(a) and 7(b) show F_{U_a} and F_{U_i} for the various experimental systems that we used. It is clear from comparison of these two charts that MOSAIC achieved significant differentiation between active and inactive components.

Using these CDFs, we can quantify the effectiveness of MOSAIC in terms of FN, TP, TN, and FP. As discussed in Section 4.3.2, MOSAIC uses ϵ to fix the FN rate at r.

Table II. Error and Accuracy Rates for the Experimental Systems

# of Users	False Negative	True Positive	True Negative	False Positive	ϵ Value
1	0.212	0.788	0.951	0.049	0.66
10	0.204	0.796	0.947	0.053	0.71
28	0.203	0.797	0.951	0.049	0.74
40	0.203	0.797	0.946	0.054	0.72
80	0.204	0.796	0.937	0.063	0.92
137	0.202	0.798	0.825	0.175	0.95

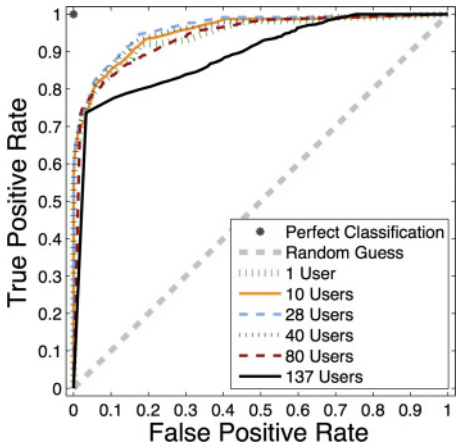

Fig. 8. ROC Curve for various experiments.

Therefore, the effectiveness of MOSAIC must be measured in its ability to minimize the FP rate based on the fixed value of FN. Because MOSAIC achieved significant differentiation between F_{U_a} and F_{U_i}, for $r = 0.20$, we were able to set ϵ at relatively high values and achieve the very favorable error rates as shown in Table II. As seen, in all experiments except for that with the highest concurrency, the unfixed error rate of FP was held to below 7%, well below the fixed FN error rate. Beyond demonstrating accuracy in the prediction of component activity, these ratios also demonstrate that MOSAIC was not noticeably impacted by an increase in concurrency in the system until concurrency reached extreme levels.

This quality of differentiation can be viewed with a *Receiver Operating Characteristic curve (ROC curve)* [Fawcett 2006; Tan et al. 2005], often used to evaluate a binary classifier independent of any other binary classifiers, as shown in Figure 8. In our case, the ROC curve depicts the change in the ratio of TP to FP as different ϵ thresholds are chosen. The extreme of $\epsilon = 1.0$ exists at the origin of the ROC plot, while the extreme of $\epsilon = 0.0$ exists at the point (1, 1) of the ROC plot. Therefore, it can be seen how the TP and FP rates respond by moving the ϵ threshold to balance between (1) rate of disruption and (2) reachability of adaptation. The ROC curve shows that MOSAIC is able to achieve a high rate of true positives despite changes in the ϵ threshold.

The comparison of the different experiments also shows the effect of concurrency on MOSAIC. As seen in Table II, higher values for ϵ are needed to achieve $r = 0.20$ as concurrency increases. This occurs because, with many users in the system, there are many more observations that allow MOSAIC to predict usage of a component c, when c is actually used. Therefore, as concurrency increases, the values for u_a are more skewed toward 1.0 until, at a concurrency error rate of roughly 60% for EDS (i.e., case of 137 users), active components are constantly at $u_a = 1.0$ until the transactions they

Table III. Tracking of False Negative (FN) Threshold

# of Users	Mean False Neg. Rate	95% Conf. Interval
1	0.209	[0.204, 0.215]
10	0.200	[0.196, 0.205]
28	0.203	[0.199, 0.207]
40	0.210	[0.207, 0.213]
80	0.206	[0.191, 0.222]
137	0.208	[0.203, 0.213]

participate in subside. While this is beneficial because it approaches perfect classification of active components (as can be seen in Figure 8, higher concurrency systems actually escalate to the (0, 1) point more directly), it results in two detriments to MOSAIC.

First, once the concurrency rate forces ϵ to be set to 1.0 given some r value, ϵ has reached its maximum value and as such cannot compensate for the increasing false positive rate by moving to a higher value. Therefore, once concurrency forces ϵ to be set to 1.0 to achieve r, MOSAIC can no longer compensate for the higher FP rates caused by even further increases in concurrency. Second, as concurrency increases to greater levels, components remain active for greater portions of time. But, since at that point all active components are effectively always at $u_a = 1.0$, problems of reachability may occur *if* the components never become inactive. An implementation of MOSAIC based on version consistency [Ma et al. 2011] would address this problem, by bringing a new version of the component on line to service the new top-level transactions, while the old component gradually transitions to an inactive state. In cases that having two versions of a component is not possible (e.g., due to limited resources), MOSAIC can fall back on quiescence [Kramer and Magee 1990], which by its nature disrupts the system, as discussed in the preamble of Section 4. In general, since MOSAIC is an opportunistic (passive) approach, in settings where a software component never reaches a suitable state for adaptation without explicit intervention, one has to fall back on disruptive methods, such as quiescence, for adaptation. That said, we have never been able to recreate such an extreme scenario in EDS, using real user loads or even the highly extreme simulated cases.

4.4.3. Accuracy of Desired Disruption Rate. The third point of evaluation is the degree to which MOSAIC achieves the desired r rate of disruption during adaptation. The evaluation results presented in the previous section and shown in Table II were *prospective* error rates due to setting ϵ at the specified level based on *historic* prediction values. In this section, then, we look to see how well the false negative rate r was tracked once ϵ was set. Table III shows the mean false negative rates and 95% confidence intervals for those false negative rates for the different experimental systems. These statistics were calculated based on 450-sample moving averages that were recalculated at 45 sample intervals. As shown, the system very effectively tracks the chosen $r = 0.20$ and maintains a fairly tight confidence interval around its mean. Furthermore, it should be noted that the rate of concurrency does not noticeably affect the tracking of r.

4.4.4. Performance and Timing. The next evaluation criteria are the performance benchmarks of *Infer and Update Models* activities. We have collected these numbers on a MacBook Pro laptop with 2.53GHz Intel Core i5 processor and 4GB 1067MHz DDR3 memory. The performance of updating the predictions at runtime consists of two primary elements: retrieval of relevant TARs (recall *CTAR* from Section 4.2.2) and update of the *Usage Prediction Registry* by applying the rules. For the former, MySQL database version 5.5.8 is used to store the rule base. However, because retrieval of TARs from MySQL was observed to take typically between 1.355s and 0.959s, we implemented a

Table IV. Performance of Rule Application

# of Users	Mean Time for Rule Application (ms)	95% Confidence Interval (ms)
1	3.23	[3.087, 3.378]
10	3.80	[3.587, 4.016]
28	2.88	[2.700, 3.056]
40	2.14	[2.093, 2.184]
80	4.90	[4.602, 5.204]
137	5.04	[4.962, 5.126]

Table V. Time to Derive Rules

# of Users	Time for Rule Learning
1	2min 36.425s
10	2min 38.946s
28	2min 11.308s
40	2min 35.236s
80	3min 28.568s
137	3min 26.992s

simple caching of the *Rule Base*. Based on this caching, the combined time of retrieving relevant TARs and updating the *Usage Prediction Registry* by applying the rules takes very little time. The mean processing times and 95% confidence intervals for those processing times are given in Table IV. As seen, the processing times are quite short, tightly bound in the 95% confidence intervals, and not noticeably effected by the increase in concurrency except for a few millisecond gain in mean processing time for larger rule bases.

4.4.5. Learning Overhead. The final evaluation criteria are the performance benchmarks of *Mining Interaction Rules* activities. We have collected these numbers on a MacBook Pro laptop with 2.53GHz Intel Core i5 processor and 4GB 1067MHz DDR3 memory. The data used in the evaluation of the mining activities was collected from 30min execution of EDS under the varying number of use cases shown in Table I. The performance of these activities was generally given less focus, since the *Mine Interaction Rules* activities are performed asynchronous to the operation of the system. As such, the overhead involved in the *Mine Interaction Rules* activities do not impact the operation of the system and can be executed independently thereof. Nonetheless, we found that the mining of the event logs to generate the rules has been extremely fast. Although we set our support and confidence values very low, resulting in a large number of rules to be generated, *Apriori* has always completed that in less than 2s in all of the experiments described here. When including additional processes involved in the learning of rules from the *Event Log*, the rule learning overhead amounted to less than 4 minutes in all experiments as shown in Table V. Of additional note, the increase in concurrency level and observed transactions in the system across the experiments did not result in a substantial increase in the time duration of learning rules. Therefore, it is expected that the rule learning activities would be scalable even as the level of concurrency in the system increases.

5. DETECTING ANOMALOUS BEHAVIOR FOR SELF-PROTECTION

This section outlines another application of our mining-based approach for inferring the models for self-management. More specifically, we describe an application of our approach in mining models that can be used for detecting anomalous, possibly malicious behaviors in component-based software. Such capability would typically be realized in the Analysis phase of the MAPE cycle in a self-protecting software system.

5.1. Background

As modern software systems become increasingly modular, distributed, and interactive, they are also facing unprecedented security challenges, especially under new computing paradigms such as mobile and cloud computing, prone to cyber attacks. Conventional techniques for securing software systems, often manually developed and statically employed, are therefore no longer sufficient. This has motivated active research in dynamic and adaptive security approaches [Chess et al. 2003]. In particular, active research has focused on self-protecting software systems, a class of systems capable of autonomously defending itself against security threats at runtime [Yuan et al. 2014b].

The first step toward autonomic and responsive security is the timely and accurate detection of compromises and vulnerabilities at runtime, which is a daunting task in its own right. Data mining techniques have been widely applied in this regard, however, most security-oriented data mining research to date has focused on "lower layers" of a software system in an architectural sense, that is, mining data at network, host machine, or source code levels. As a result, such approaches mainly address specific types of threats that are tactical in nature, but the "big picture" understanding of attacker strategy and intent, as well as overall security posture of the system appears to be lacking. Furthermore, these approaches typically can do very little to address the growing concern of insider threats, where attackers use the system with legitimate credentials instead of intrusions [Salem et al. 2008].

In contrast, our research has focused on developing a threat detection approach based on *software component interactions* as opposed to mining data collected from network traffic or source code. Our underlying insight is that many cyber attacks *misuse* the system in a way that deviates from normal system behavior. Take the EDS system for instance; since it is an online system that manages sensitive information such as personnel records and locations, it may be subject to various intrusions and exploits including SQL Injection, Cross-Site Scripting (XSS), and Cross-Site Request Forgery (CSRF), just to name a few [OWASP 2013]. As a concrete example, suppose an attacker obtains a valid user login and hijacks the Strategy Analyzer component through the Headquarters User Interface (HQUI) component. Instead of calling *Weather Analyzer* and *Strategy Analysis KB* components as prescribed in the use case in Figure 2, the attacker sends requests from Strategy Analyzer to the *Resource Manager* and *Repository* components to retrieve sensitive information about all deployed resources, as shown in Figure 9. Such a violation of system usage occurs at the application level and is therefore much harder to detect and thwart using conventional firewalls and intrusion detection devices, which are primarily concerned with ports and protocols.

To be able to effectively detect potentially malicious behavior at the application level, there are two main categories of techniques: *signature-based* or *anomaly-based*. Signature-based techniques attempt to capture the signatures or specifications of attacks as the basis for detection, which are usually very accurate and efficient but require constant maintenance as attack strategies and tactics evolve ever so rapidly. Nor can these techniques detect unknown threats. Anomaly-based techniques, on the other hand, seek to build a "normal" system usage model as the basis for threat detection. Our research shows that the generic mining approach depicted in Figure 3 can be used as an effective model following the latter category.

5.2. Anomaly Detection Based on Association Rules

After association rules are generated from the mining phase as described in Section 4.1, we are not quite ready to apply them directly to detect anomalous behavior from the system's event execution streams. In fact, this problem scenario is the opposite of typical uses of association rules—instead of using them to predict what item(s) are

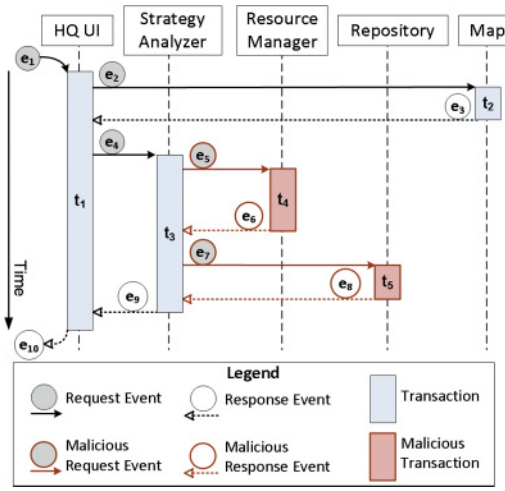

Fig. 9. Example attack scenario.

frequently associated with a given set of items, the question we ask here is, "is this transaction *infrequently* associated with the given itemset of transactions, so much as to warrant an alarm?" Our basic approach can be roughly outlined as follows:

For an itemset containing a top-level transaction T and a set of child transactions $t_1, t_2, \ldots, t_n$, by observing the start and end times of the messages, we can find all the Enclosing Transaction Sequences (ETS) $<x_1, \ldots, x_{j-1}, x_j>$ such that each transaction is the child of the preceding transaction. In Figure 2, for example, $<t_1, t_2>$, $<t_1, t_3, t_4, t_5>$, and $<t_1, t_3, t_6>$ are such sequences. For normal use cases, all such sequences should have been captured in the rule base after we have observed enough transactions from the event log. For malicious uses of the system, however, some of the ETS such as $<t_1, t_3, t_4>$ and $<t_1, t_3, t_5>$ in Figure 9 occur at a much lower frequency and thus not found in the rule base. Each not-found ETS will be marked as a violation, and we can mark the itemset as anomalous when the number of violations reach a certain threshold.

Note that, given the fact that anomalous events are rare events occurring with low frequency, the vast majority of the generated association rules represent normal system use. Therefore, unlike traditional associations mining, we set the minimum support and confidence levels of the Apriori algorithm to a very low level (e.g., 0.1).

Initial evaluations in our prior work [Yuan et al. 2014a], however, showed that even though the approach works effectively for a single-user scenario, the precision of detection deteriorates rapidly as the number of concurrent users grows. The reason is obvious: as the system concurrency increases, a child transaction in the event log may be observed to fall under multiple overlapping top-level transactions, thereby causing more and more erroneous ETS sequences to be observed that resulted in False Positives. To address this issue, we enhanced our approach by developing a *quantitative anomaly likelihood* that accounts for multiple overlapping top-level transactions, in lieu of a simple yes/no flag for marking ETS violations.

For any transaction t, suppose by observing the start and end timestamps of messages we find m top-level transactions $T_1, T_2, \ldots, T_m$ under which t falls. For a normal transaction, one of the top-level transactions should be a true parent, with matching ETS sequences captured in the rule base that "explains" the existence of t. Conversely, if none of the T_i's can explain t, we have reason to believe it is an anomaly. In other words,

Table VI. Detection Results Under Varying Concurrency, Confidence Threshold = 0.90

#Active users	10	20	50
TP Count	50	47	40
FP Count	10	9	5
FN Count	12	18	13
TN Count	35,056	34,505	34,398
TP Rate (TPR)	0.806	0.723	0.755
FP Rate (FPR)	2.85E-4	2.61E-4	1.45E-4
Precision	0.833	0.839	0.889
Recall	0.806	0.723	0.755
F-Measure	0.820	0.777	0.816
$TPR = TP/(TP + FN); FPR = FP/(FP + TN)$			
$Precision = TP/(TP + FP); Recall = TPR$			
$F\text{-}Measure = 2TP/(2TP + FP + FN)$			

Size of data: 5000 itemsets/use cases per user.

the likelihood of t being an anomaly is the conjunctive probability of t **not** explained by T_i for all $i = 1, \ldots, m$. More formally, assuming probabilistically independent actions among concurrent users, we have

$$P_{anomaly}(t) = \left[\prod_{i=1}^{m}(1 - conf_i)\right]^{\frac{1}{m}}, \tag{1}$$

where $conf_i$ is the highest confidence of the TARs (recall Section 4.1) we can find in the rule base for the ETS's of t among the children of T_i. Note that the anomaly likelihood is normalized by taking the geometric mean. Obviously, $P_{anomaly}(t) \in [0, 1]$, and when it exceeds a configurable detection threshold, the transaction is detected to be an anomaly.

5.3. Evaluation

Our experimentation environment involves a customized instance of the original EDS system in a similar setup as introduced in the previous section. In addition to the normal system use cases, we injected the attack scenario outlined in Figure 9 to the simulation runs according to a predefined anomaly probability, which is set at ∼0.3% (3 σ or standard deviations of a normal distribution) under the assumption that covert malicious attacks are rare events.

We use the same Apriori implementation from WEKA for association rule generation. Both simulation and data analysis are run on a quad-core Mac OS X machine. Table VI shows our new evaluation results with different numbers of users. The new results show our enhanced algorithm is very effective in detecting the anomalous use of the system in an automated, unattended fashion, with both high recall and high precision. In the 10-user scenario, for example, our approach detects 80.6% of the anomalous events with a 83.3% precision. This demonstrates that our approach can be used as an effective mechanism to enhance both overall system security and security administrator productivity. As such, our approach does not seek to replace existing security mechanisms, such as network- and host-based Intrusion Detection Systems (IDS), but rather complement them as an added line of defense against sophisticated threats that may otherwise go unnoticed.

More importantly, the new results show no apparent degradation in accuracy over an increase in system concurrency all the way to 50 concurrent users, validating the effective use of the normalized anomaly likelihood. Note that the concurrent users in our simulation runs are "intense" users used to generate a heavy load on the system,

therefore *they actually represent a much larger number of human users in a real-world system*.

6. SELF-OPTIMIZATION OF DEPLOYMENT TOPOLOGY

As our final illustration, we outline an application of our approach for improving the performance of software through redeployment of its components. Such capability would typically be realized in the Planning phase of the MAPE cycle in a self-optimizing software system.

6.1. Background

The design and development of large-scale, component-based software systems today are influenced by modern software engineering practices as embodied by architecture styles (e.g., pipe and filter), design patterns (e.g., proxies), and coding paradigms (e.g., aspect orientation). A direct consequence is that the deployment of such systems becomes more complex and fluid, with hundreds or perhaps even thousands of options and parameters to consider, along dimensions such as location, capacity, timing, sequencing, service levels, security, etc. Many of them may be interdependent and possibly conflicting. Due to the combinatorially large problem space, the values of these parameters are usually set and fine-tuned manually by experts, based on rules of thumb and experience.

An objective for autonomic systems is therefore to intelligently navigate the solution space and seek ways to optimally (re)deploy the system to improve the system performance and cost [Kephart and Chess 2003].

To illustrate the self-optimization challenge, we turn our attention to the deployment topology of the EDS system. As a geographically distributed system, some of the components, such as HQUI (recall Figure 1), need to reside at the headquarters (HQ) facility, while some, such as the Resource Monitor, are required to be at a remote site to be collocated with emergency response equipment. Other components are more flexible and can be deployed at either location. Depending on the topology, intercomponent messages can be either local (via interprocess communication on a single computer or over a LAN), or remote over a WAN, with the latter having a much larger network latency. Take the strategy analysis use case outlined in Figure 2 for example; if the *Strategy Analyzer* and *Strategy Analyzer KB* components reside at different sites, transaction t_6 may take a much longer time than what it would be if the two components were collocated, adversely affecting the response time experienced by the end user. Obviously, the system should employ a deployment topology that minimizes remote transactions to reduce overall network latency, subject to other constraints.

It is worth noting that this is by no means a new problem, and has been manifested in various settings such as system resource management [Poladian et al. 2004], cloud performance optimization [Casalicchio et al. 2013], wireless network configuration [Malek et al. 2007], etc. However, traditional approaches, including our own prior work [Malek et al. 2007, 2012], assume the availability of a detailed component interaction model, that includes information about the component dependencies, frequency of interactions among the components, size of exchanged data, processing sequences, etc. As pointed out earlier in the article, such a model is difficult to come by and costly to maintain.

Our proposed approach, on the other hand, leverages the same component interaction model for dynamic optimization of the deployment topology at runtime, a model that needs no prior development and can stay up to date even when the system behavior shifts. Many modern middleware frameworks that support adaptation (such as [Malek et al. 2012]) provide facility for redeploying components across distributed locations in a software system. Thanks to new advances in computing infrastructures such as virtualization and cloud computing, dynamically redeploying software components is

being made easier than ever. Puppet software [PuppetLabs 2015] and CloudFormation service from Amazon Web Services [Amazon Web Services 2015], for example, provide programmable mechanisms to create, configure, and manage virtualized computing resources on the fly.

6.2. Applying Association Rules

It is easy to see that the problem of determining deployment topology, namely, assigning component c_i to location S_j, can be framed as a clustering problem. Intuitively, transactions that have a higher probability of occurring together should be local (i.e., in the same cluster). One straightforward implementation approach is to use an agglomerative hierachical clustering algorithm [Tan et al. 2005]: we start with individual components as single-point clusters, then successively merge the two closest clusters until only the desired number of clusters remain. Typically, the "closeness" between two clusters is based on a proximity/distance measure such as the Euclidean distance. In our problem context, we could of course simply observe and compute the average frequency of pairwise transaction events between two component clusters as the proximity measure. This approach is effective in grouping frequently interacting components together, but has known limitations such as the tendency to reach local optima due to the lack of a global objective—our evaluation will later confirm this.

Given the fact that our mining approach produces a rule base that captures not only the frequency count of single pairwise transactions (note that a single transaction can be viewed as a TAR of the form $X \rightarrow Y$ where X is the empty set ϕ and $Y = \{t\}$), but also the probability of the *co-occurrence* of a set of transactions, we can define a better proximity measure. In this application scenario, we are only interested in the support value s of an itemset $X \cup Y$ as defined in Section 4.1.3, which is readily available as an intermediate result from the Apriori algorithm. We also denote t_{ij} as a transaction that is initiated from component i to component j (i.e., $i = t.start.src$ and $j = t.start.dst$), or vice versa.

More formally, we define the *cohesion* measure of a component cluster C as

$$Cohesion(C) = \left(\sum_{U \in 2^T} s(U) \right) / |C|,$$

where $T = \{t_{ij} | \forall i \in C \wedge j \in C\}$ is the set of "local" transactions within cluster C, and 2^T is the powerset of T. In other words, the cohesion measure of a cluster of components is the sum of the support values of all subsets of local transactions within C, normalized by the cluster size $|C|$.

Using the cohesion measure, the proximity between two clusters C_1 and C_2 is therefore defined as the *cohesion gain* resulting from their would-be merge:

$$proximity(C_1, C_2) = Cohesion(C_1 \cup C_2) - Cohesion(C_1) - Cohesion(C_2).$$

Intuitively, this measure encourages the merge of two sets of components if the merge results in more localized transaction sequences. We can see that our component interaction model captured from associations mining can be used to provide a *probabilistic proximity measure* that is informed by system-wide transactions rather than simple pairwise events.

6.3. Evaluation

To evaluate the effectiveness of applying the TARs in improving EDS deployment topology, we instrumented the server-side Java code to simulate network latency based on a configurable topology "metamodel." Before making an intercomponent method call, each calling component (StrategyAnalyzer, Repository, etc.) queries the metamodel

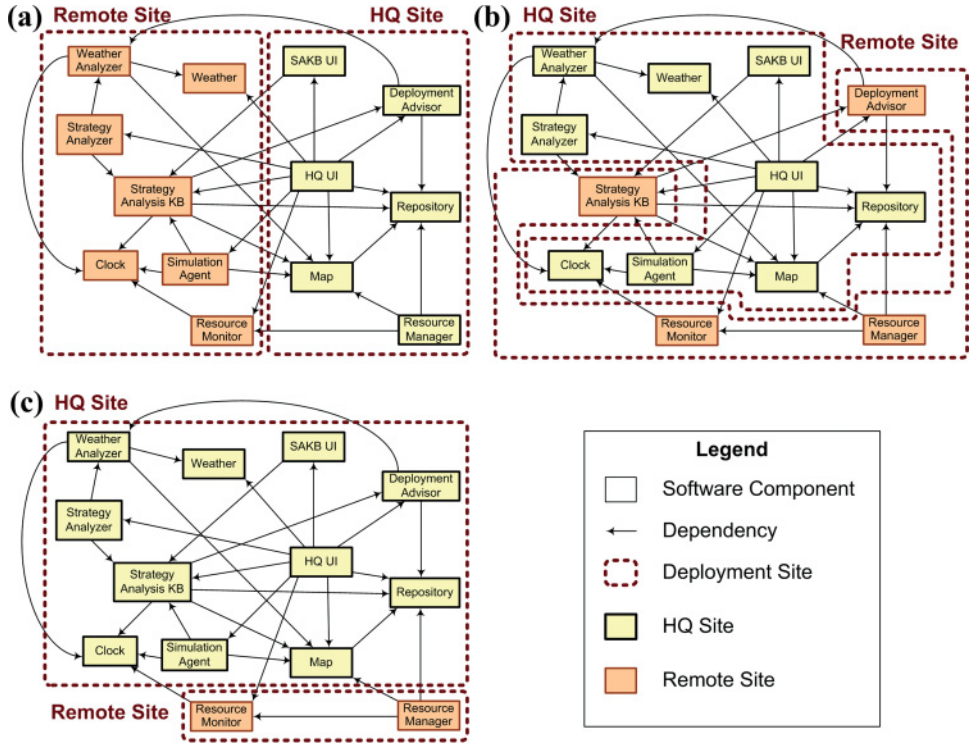

Fig. 10. Alternative EDS deployments: (a) initial deployment, (b) optimized deployment based on the clustering technique, and (c) optimized deployment based on the mined component interaction model.

to determine if it is a local (LAN) or remote (WAN) call and generates a Gaussian-distributed network latency time accordingly.

In test runs with 20 active concurrent users, we set the mean network latency for LAN and WAN at 10ms and 100ms, respectively. The first test run of the system used an arbitrary deployment topology, as shown in Figure 10(a). Here, no special attention was given to the locality of the components: they were divided up more or less evenly between the HQ site and the remote site. System execution logs showed that average transaction latency is about 44ms.

As an alternative technique for comparisons sake, we first used a basic hierarchical clustering algorithm that used pairwise event frequency $\sigma(t_{ij})$ between any two components i and j as the distance function. The algorithm recommended a different topology as shown in Figure 10(b). The second set of test runs conducted under this new topology showed that average transaction latency was reduced to about 30ms, indicating a significant improvement.

Now we turn to evaluate our proposed approach based on the component interaction model. After mining the system execution traces, the Apriori algorithm created a rule base that doubled as a probabilistic proximity matrix for any two sets of components. Feeding the cohesion gain based function introduced in Section 6.2 to the hierachical clustering algorithm, a third topology emerged, as shown in Figure 10(c). Test runs based on the new topology showed that average transaction latency was further reduced to 20ms, a 33% reduction compared with the pairwise transaction frequency based clustering and a 54% reduction compared with the original topology.

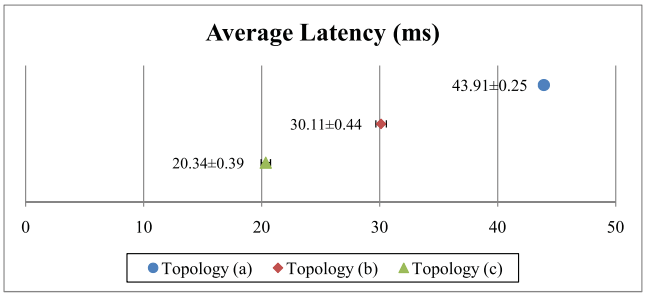

Fig. 11. System latency for three deployment topologies (average latency at 95% confidence interval).

We conducted 20 batch runs for each deployment topology, eliminating transients by taking observations only after the system entered a steady state. The average latency for each deployment topology, along with the corresponding 95% confidence interval are shown in Figure 11. During the clustering process, we ensured that the *HQUI* and *ResourceMonitor* components are preassigned to the HQ site and the remote site, respectively (otherwise the optimization would result in all components being assigned to a single site). Statistical tests can also confirm that the latency improvements are not trivial. We have validated that taking into account system-wide event co-occurrences can help overcome the local optima during the clustering process, resulting in improved system performance.

Note that in practice, optimal deployment of resources depends on many other factors besides network latency, such as cost of component redeployment, hardware capacity at each location, etc. A more holistic approach needs to formulate a higher-level objective function that weighs benefits against various costs and constraints (e.g., as developed in Bobroff et al. [2007] and Malek et al. [2012]). In that case, the component-wise probabilistic proximity measure from our model can become an input to the larger optimization algorithm.

The advantage of the architecture-based approach is evident in this application scenario: Even though the mapping from software components to hosts and sites is changed as the result of automated redeployment, the self-adaptation happens within the network and host configurations of the base-level subsystem (e.g., new URLs for Java RMI calls). The abstract component interaction model within the metalevel subsystem, in the form of TARs captured in the rule base, is transparent to the deployment topology change, therefore remains intact and does not need to be relearned.

7. RELATED WORK

Researchers have used log of event data collected from a system to construct a model of it for various purposes. Cook et al. [Cook and Wolf 1998a] use the event data generated by a software process to discover the formal sequential model of that process. In a subsequent work [Cook and Wolf 1998b], they have extended their work to use the event traces for a concurrent system to build a concurrency model of it. Gaaloul et al. [2008] discover the implicit orchestration protocol behind a set of web services through structural web service mining of the event logs and express them explicitly in terms of Business Process Execution Language (BPEL). Motahari-Nezhad et al. [2011] present an algorithmic approach for correlating individual events, which are scattered across several systems and data sources, semiautomatically. They use these correlations to find the events that belong to the same business process execution instance. Wen et al. [2009] use the start and end of transactions from the event log to build Petri nets corresponding to the processes of the system. To our knowledge, except our recent

work [Canavera et al. 2012], no previous work has used mining of execution log to understand the dynamic behavior of the system for the purpose of self-adaptation.

As mentioned earlier in this article, even though data mining techniques, including anomaly detection, have been extensively used in the security arena for decades, most of the research has centered around (a) intrusion detection, especially at network and host levels (e.g., Lee et al. [1999]) and (b) malware/virus detection at source code and executable level (e.g., Schultz et al. [2001]). Among those, several efforts share our approach of unsupervised learning, that is, using unlabeled or "noisy" training data. Portnoy et al. [2001], for example, used a distance-based clustering for detecting network intrusions. Lane and Brodley used unsupervised machine learning based on a similarity measure to classify user behavior in UNIX command shells [Lane and Brodley 1997]. Eskin et al. developed a geometric framework that projects unlabeled data to a high-dimensional feature space before applying clustering algorithms to detect anomalies in sparse regions of the feature space [Eskin et al. 2002]. Still others used mining algorithms such as Support Vector Machines (SVM) [Khan et al. 2007], Hidden Markov Models (HMM) [Warrender et al. 1999], ensemble-based learning [Parveen et al. 2011], graph mining [Christodorescu et al. 2008], etc.

Our approach differs from these approaches in that (a) none of them used associations mining, except for the ADAM framework, which used associations mining for network intrusions in supervised mode [Barbará et al. 2001]; and (b) little research has focused on detecting malicious behavior at the architecture/component level. We believe detecting malicious behavior at the architectural level is a prerequisite for developing self-protection mechanisms that modify the system's architecture to mitigate the security threats; (c) furthermore, most existing data mining research assume normal program behavior is deterministic and stable, whereas we assume the behavior for an interactive system is inherently fluid and user driven, and hence continually updates the model based on recent system execution traces.

Data mining techniques are increasingly applied in the software engineering domain to improve software productivity and quality [Xie et al. 2009]. The datasets of interest include execution sequences, call graphs, and text (such as bug reports and software documentation). One body of research, for instance, focuses on mining software specifications—frequent patterns that occur in execution traces [Lo et al. 2009], which is similar to our problem but the focus is on mining API call usages for purposes such as bug detection, not for self-adaptation; their techniques (such as libSVM) are also different.

Finally, our research is related to models@runtime approaches [Bencomo et al. 2014], where the models are considered to be the abstract representation of the system during execution. In our approach, we use data mining techniques to derive these models from the running system, while some other techniques (e.g., Bencomo et al. [2013]) keep the running system up to date with the models.

8. DISCUSSION AND FUTURE WORK

The underlying assumption in the current version of our approach is that a single data mining algorithm can process all the events/transactions in the system and build the stochastic component interaction models. This may not be possible, especially when we consider distributed software systems that permeate boundaries of several enterprises. An enterprise may be unwilling to share its internal structure and event logs with an entity that is out of its control for various reasons (e.g., protecting competitive edge, security concerns, etc.). Therefore, we are working on a distributed version of our approach, which achieves the same goal by running multiple local data mining algorithms. In fact, initial results show that confining the algorithms to the boundaries of enterprises improves the precision, not to mention scalability. The natural structure

imposed by the boundary of an enterprise only allows certain components to talk to the outside world (i.e., other enterprises). This knowledge helps to reduce the concurrency error significantly.

Regardless of the adaptation needs, the proposed mining approach needs to be highly efficient and as scalable as the base subsystem itself in order to process the system execution traces as they occur and provide dynamic, near-real-time predictions. For this reason we plan to conduct an in-depth analysis of the computational characteristics of association mining algorithms and ideally leverage elastic, on-demand computing platforms (e.g., MapReduce) to speed up the mining performance.

The data mining algorithm that we used to build the stochastic component interaction models is based on set theory. Therefore, it is not able to leverage the frequency of event occurrences nor the temporal ordering among events, which are already available in the execution log of the system. We believe using tis extra information can increase the accuracy of the inferred models, and in turn, make our approach more precise. Hence, we are studying the application of other types of data mining algorithms (e.g., sequential pattern mining [Tan et al. 2005]) that can use the extra information.

The accuracy of mined rules depends on the availability of a sufficiently large usage history of the software, exercising the interactions among the system's component. Such data could either be collected through benchmark of the system or its previous deployments. However, determining how much data is needed to allow for generation of accurate rules is challenging. The notion of component interaction coverage metric [Williams and Probert 2001] provides a good starting point in addressing this issue. In addition, we plan to investigate how this approach would work in a "cold-start" mode, that is, when a system is initially launched. One solution would be to start off with pessimistic (conservative) predictions, until actual usage patterns are learned. In addition, we plan to explore the use of data stream mining [Gaber et al. 2005] in this context, which allows for the mining to be performed incrementally using the real-time stream of observations from the system.

Another issue worth considering is the scenario in which adaptations affect the functional behavior of the system, for example, replacement of a component that introduces new behaviors. In such a setting, the changes may make part of the historical data collected from the system, and thereby the learned rules, obsolete. MOSAIC is currently able to detect such changes through its *Check Prediction Accuracy* activity. However, in such settings, MOSAIC would need additional data to be able to update the rules. During the time it takes to collect the additional data, MOSAIC would still be able to guarantee the consistency of adaptation through the conservative approach described in Section 4.3.1, but may not be able to achieve an accurate trade-off between disruption and reachability (recall Section 4.3.2). In situations where such inaccuracies are not acceptable, MOSAIC could still be used for adaptations that do not change the functional behavior of the components, but change their nonfunctional properties (e.g., security, availability, etc.). In general, MOSAIC is most effective in settings where the frequency of adaptation is not faster than the time it takes to learn the new rules.

Our ongoing research for anomalous behavior detection focuses on more extensive evaluation of the detection algorithm to enhance its accuracy and robustness. In particular, we will evaluate the algorithm's sensitivity against different input parameters (e.g., minimum support and confidence levels) to better understand its "sweet spots" and limitations. Last but not least, we seek to prove that our approach is effective against unknown threats, which we hypothesized in Section 5.1.

REFERENCES

Amazon Web Services. 2015. Amazon Web Services (AWS) CloudFormation. Retrieved from http://aws.amazon.com/cloudformation/.

Daniel Barbará, Julia Couto, Sushil Jajodia, and Ningning Wu. 2001. ADAM: A testbed for exploring the use of data mining in intrusion detection. *ACM Sigmod Record* 30, 4 (2001), 15–24.

Nelly Bencomo, Amel Bennaceur, Paul Grace, Gordon Blair, and Valerie Issarny. 2013. The role of models@run.time in supporting on-the-fly interoperability. *Computing* 95, 3 (March 2013), 167–190.

Nelly Bencomo, Robert B. France, Betty H. C. Cheng, and Uwe Aßmann (Eds.). 2014. *Models@run.time: Foundations, Applications, and Roadmaps*. Springer.

Dimitri P. Bertsekas and John N. Tsitsiklis. 2008. *Introduction to Probability* (2nd. ed.). Athena Scientific.

Norman Bobroff, Andrzej Kochut, and Kirk Beaty. 2007. Dynamic placement of virtual machines for managing SLA violations. In *IFIP/IEEE International Symposium on Integrated Network Management*, 119–128.

Kyle R. Canavera, Naeem Esfahani, and Sam Malek. 2012. Mining the execution history of a software system to infer the best time for its adaptation. In *Internatonal Symposium on the Foundations of Software Engineering*, 18:1–18:11.

Emiliano Casalicchio, Daniel A. Menasc, and Arwa Aldhalaan. 2013. Autonomic resource provisioning in cloud systems with availability goals. In *ACM Cloud and Autonomic Computing Conference*, 1:1–1:10.

David M. Chess, Charles C. Palmer, and Steve R. White. 2003. Security in an autonomic computing environment. *IBM Systems Journal* 42, 1 (2003), 107–118.

Mihai Christodorescu, Somesh Jha, and Christopher Kruegel. 2008. Mining specifications of malicious behavior. In *Proceedings of the 1st India Software Engineering Conference*. ACM, 5–14.

Jonathan E. Cook and Alexander L. Wolf. 1998a. Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering Methodology* 7, 3 (July 1998), 215–249.

Jonathan E. Cook and Alexander L. Wolf. 1998b. Event-based detection of concurrency. In *International Symposium on the Foundations of Software Engineering*, 35–45.

Eleazar Eskin, Andrew Arnold, Michael Prerau, Leonid Portnoy, and Sal Stolfo. 2002. A geometric framework for unsupervised anomaly detection. In *Applications of Data Mining in Computer Security*, Daniel Barbará and Sushil Jajodia (Eds.). Number 6 in Advances in Information Security. Springer US, 77–101.

Tom Fawcett. 2006. An introduction to ROC analysis. *Pattern Recognition Letters* 27, 8 (June 2006), 861–874.

Walid Gaaloul, Karim Baina, and Claude Godart. 2008. Log-based mining techniques applied to Web service composition reengineering. *Service Oriented Computing and Applications* 2, 2–3 (May 2008), 93–110.

Mohamed Medhat Gaber, Arkady Zaslavsky, and Shonali Krishnaswamy. 2005. Mining data streams: A review. *SIGMOD Record* 34, 2 (June 2005), 18–26.

David Garlan, Shang Wen Cheng, An Cheng Huang, Bradley Schmerl, and Peter Steenkiste. 2004. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer* 37, 10 (Oct. 2004), 46–54.

Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. 2009. The WEKA data mining software: An update. *SIGKDD Explorations Newsletter* 11, 1 (Nov. 2009), 10–18.

Jiawei Han and Micheline Kamber. 2006. *Data Mining: Concepts and Techniques*. Morgan Kaufmann.

Jeffrey O. Kephart and David M. Chess. 2003. The vision of autonomic computing. *IEEE Computer* 36, 1 (Jan. 2003), 41–50.

Latifur Khan, Mamoun Awad, and Bhavani Thuraisingham. 2007. A new intrusion detection system using support vector machines and hierarchical clustering. *The VLDB Journal the International Journal on Very Large Data Bases* 16, 4 (2007), 507–521.

Jeff Kramer and Jeff Magee. 1990. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering* 16, 11 (Nov. 1990), 1293–1306.

Terran Lane and Carla E. Brodley. 1997. An application of machine learning to anomaly detection. In *Proceedings of the 20th National Information Systems Security Conference*, Vol. 377, 366–380.

Wenke Lee, S. J. Stolfo, and K. W. Mok. 1999. A data mining framework for building intrusion detection models. In *IEEE Symposium on Security and Privacy*, 120–132.

David Lo, Hong Cheng, Jiawei Han, Siau-Cheng Khoo, and Chengnian Sun. 2009. Classification of software behaviors for failure detection: A discriminative pattern mining approach. In *International Conference on Knowledge Discovery and Data Mining*, 557–566.

Xiaoxing Ma, Luciano Baresi, Carlo Ghezzi, Valerio Panzica La Manna, and Jian Lu. 2011. Version-consistent dynamic reconfiguration of component-based distributed systems. In *International Symposium on the Foundations of Software Engineering*, 245–255.

Sam Malek, Nenad Medvidovic, and Marija Mikic-Rakic. 2012. An extensible framework for improving a distributed software system's deployment architecture. *IEEE Transactions on Software Engineering* 38, 1 (Feb. 2012), 73–100.

Sam Malek, Marija Mikic-Rakic, and Nenad Medvidovic. 2005. A style-aware architectural middleware for resource-constrained, distributed systems. *IEEE Transactions on Software Engineering* 31, 3 (March 2005), 256–272.

Sam Malek, Chiyoung Seo, Sharmila Ravula, Brad Petrus, and Nenad Medvidovic. 2007. Reconceptualizing a family of heterogeneous embedded systems via explicit architectural support. In *International Conference on Software Engineering*, 591–601.

Hamid Reza Motahari-Nezhad, Regis Saint-Paul, Fabio Casati, and Boualem Benatallah. 2011. Event correlation for process discovery from web service interaction logs. *The VLDB Journal* 20, 3 (June 2011), 417–444.

OWASP. 2013. OWASP Top Ten Project. Retrieved from https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project.

Pallabi Parveen, Zackary R. Weger, Bhavani Thuraisingham, Kevin Hamlen, and Latifur Khan. 2011. Supervised learning for insider threat detection using stream mining. In *2011 23rd IEEE International Conference on Tools with Artificial Intelligence (ICTAI),* IEEE, 1032–1039.

Vahe Poladian, Joao Pedro Sousa, David Garlan, and Mary Shaw. 2004. Dynamic configuration of resource-aware services. In *International Conference on Software Engineering*, 604–613.

Leonid Portnoy, Eleazar Eskin, and Sal Stolfo. 2001. Intrusion detection with unlabeled data using clustering. In *In Proceedings of ACM CSS Workshop on Data Mining Applied to Security (DMSA-2001)*. 5–8.

PuppetLabs. 2015. Puppet software. Retrieved from http://puppetlabs.com/.

Malek Ben Salem, Shlomo Hershkop, and Salvatore J. Stolfo. 2008. A survey of insider attack detection research. In *Insider Attack and Cyber Security*, Salvatore J. Stolfo, Steven M. Bellovin, Angelos D. Keromytis, Shlomo Hershkop, Sean W. Smith, and Sara Sinclair (Eds.). Number 39. Springer, 69–90.

Matthew G. Schultz, Eleazar Eskin, Erez Zadok, and Salvatore J. Stolfo. 2001. Data mining methods for detection of new malicious executables. In *IEEE Symposium on Security and Privacy*, 38–49.

Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. 2005. *Introduction to Data Mining* (1st. ed.). Addison Wesley.

Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy. 2009. *Software Architecture: Foundations, Theory, and Practice*. Wiley.

Yves Vandewoude, Peter Ebraert, Yolande Berbers, and Theo D'Hondt. 2007. Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates. *IEEE Transactions on Software Engineering* 33, 12 (Dec. 2007), 856–868.

Christina Warrender, Stephanie Forrest, and Barak Pearlmutter. 1999. Detecting intrusions using system calls: Alternative data models. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy.* IEEE, 133–145.

Lijie Wen, Jianmin Wang, Wil M. Aalst, Biqing Huang, and Jiaguang Sun. 2009. A novel approach for process mining based on event types. *Journal of Intelligent Information Systems* 32, 2 (April 2009), 163–190.

Alan W. Williams and Robert L. Probert. 2001. A measure for component interaction test coverage. In *ACS/IEEE International Conference on Computer Systems and Applications*, 304–312.

Tao Xie, Suresh Thummalapenta, David Lo, and Chao Liu. 2009. Data mining for software engineering. *IEEE Computer* 42, 8 (Aug. 2009), 55–62.

Eric Yuan, Naeem Esfahani, and Sam Malek. 2014a. Automated mining of software component interactions for self-adaptation. In *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 27–36.

Eric Yuan, Naeem Esfahani, and Sam Malek. 2014b. A systematic survey of self-protecting software systems. *ACM Transactions on Autonomous and Adaptive Systems* 8, 4 (Jan. 2014), 17:1–17:41.

Received October 2014; revised October 2015; accepted December 2015