Department of Computer Science
George Mason University Technical Reports

4400 University Drive MS#4A5
Fairfax, VA 22030-4444 USA
http://cs.gmu.edu/ 703-993-1530

Mining the Execution History of a Software System to Infer the Best Time for its Adaptation

Kyle R. Canavera, Naeem Esfahani, and Sam Malek
{kcanaver, nesfaha2, smalek}@gmu.edu

Technical Report GMU-CS-TR-2012-2

Abstract

An important challenge in dynamic adaptation of a software system is to prevent inconsistencies (failures) and disruptions in its operations during and after change. Several prior techniques have solved this problem with various tradeoffs. All of them, however, assume the availability of detailed component dependency models. This paper presents a complementary technique that solves this problem in settings where such models are either not available, difficult to build, or outdated due to the evolution of the software. Our approach first mines the execution history of a software system to infer a *stochastic component dependency model*, representing the probabilistic sequence of interactions among the system's components. We then demonstrate how this model could be used at runtime to infer the "best time" for adaptation of the system's components. We have thoroughly evaluated this research on a multi-user real world software system and under varying conditions.

1 Introduction

As engineers have developed new techniques to address the complexity associated with the construction of modern-day software systems, an equally pressing need has risen for mechanisms that automate and simplify the management of those systems after they are deployed, i.e., during runtime. This has called for the development of (self-)adaptive software systems [12, 14]. However, the construction of such systems has been shown to be significantly more challenging than traditional software systems [3, 15].

One important challenge is the management of the runtime change to avoid inconsistencies during and after the adaptation. Informally, an *inconsistent application state* is one from which the system progresses towards an error state [13]. In a component-based software system, *application transactions* change the state of the system. An application transaction is defined as a set of related interactions among two or more software components. The important observation is that while a transaction is in progress, the internal state of the participating components may be mutually inconsistent [13]. To avoid inconsistencies, replacement of components should be delayed until the transaction has ended and the participating components have a stable state.

In their seminal work [13], Kramer and Magee developed a technique, known as *quiescence*, that from a *static component dependency* model of the system (e.g., UML Component Diagram) calculates the components that have to be halted (passivated) before a component can be safely adapted. Reliance on a static component dependency model, however, makes quiescence rather pessimistic in its analysis, which could lead to significant delays and disruptions. This is an issue that has been tackled in two recent approaches, *tranquility* [21] and *version-consistency* [16], which have showed that by leveraging the *dynamic component dependency* model of the system (e.g., UML Sequence Diagram) it is possible to become more refined in the analysis and thus reduce the unnecessary overhead.

All of these approaches, however, assume the availability of an accurate component dependency model of the system. While this may be true in some cases, often such models are either not available or provide an inaccurate representation of the system's evolving dependencies. For instance, consider that the majority of existing open-source software systems lack such models, and when not, the models are not necessarily up-to-date with the system's implementation. Moreover, in emerging software systems, such as those comprised of externally provided services, the dependencies among the system's components are constantly changing, making it difficult to maintain such models.

In this paper, we present a novel approach that determines the "best time" for adapting a system's software components in settings where an accurate model of their dependencies is not available. We define "best

time" to be the time at which the adaptation of a given component results in neither inconsistency (failure), nor significant disruption to the system. The underlying insight is that by collecting a software system's execution history for a sufficiently long period of time, it is possible to mine a *stochastic component dependency* model of the system. This model provides a new kind of probabilistic information that has been lacking in the models used for making adaptation decisions in the prior research [16, 21]. We first leverage data mining techniques to infer a set of probabilistic rules representing the dynamic component dependencies among a system's software components. The rules are then used at runtime for determining the likelihood of a component being in an appropriate state for adaptation at a given point in time. Finally, by checking our predictions against the actual behavior of the system, we are able to continuously refine the dependency models to the system's evolving interactions.

Our experiences with thorough evaluation of this approach in the context of a large distributed software system have been very positive. The results have shown the ability to infer precise models that can be used to effectively manage the interruptions caused by adaptation. We have also developed and evaluated a novel technique that prevents inconsistencies, even when our predictions are off.

The remainder of this paper is organized as follows. Section 2 describes a software system used for illustration of the research and its evaluation. Section 3 provides the necessary background, while Section 4 motivates the research in the context of prior work. Section 5 provides an overview of our approach. Sections 6 to 8 delve into the details. Section 9 presents the evaluation. The paper concludes with an overview of prior research and avenues of future research.

2 Illustrative Example

We illustrate the concepts using a software system, called Emergency Deployment System (EDS) [17], and 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 resources 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. Interested reader may find a more detailed description of EDS in [17]. It suffices to say that EDS is representative of a large component-based software system, where the components communicate by exchanging messages

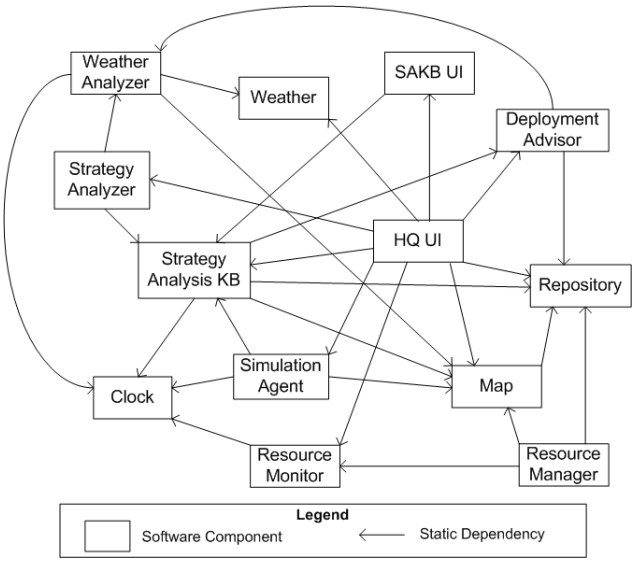

Figure 1: Subset of the Emergency Deployment System's software architecture.

(events). In the largest deployment of EDS to-date , it was deployed on 105 nodes and used by more than 100 users [17].

Systems such as EDS are often deployed in highly unpredictable and dynamic settings. Therefore, it is often desirable to be able to adapt such systems at runtime to deal with changes that may affect the system's functional or non-functional properties. However, such changes should occur in a manner that do not lead to inconsistency or significant disruption in the services provisioned to the users.

3 Research Background

Kramer and Magee [13] showed that for a component to remain in a consistent state during/after adaptation, it should not be changed in the middle of a transaction. They defined *transaction* to be exchange of event between two components by which the state of a component is affected. A *dependent transaction* is in turn a transaction whose completion depends on the completion of consequent transactions.

We first formally define and then illustrate these concepts using a subset of transactions comprising EDS below. Figure 2 shows the transactions corresponding to the *strategy analysis capability*, which is only one of the use cases in EDS.

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, 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. In this area of research, it is assumed that events, including their source

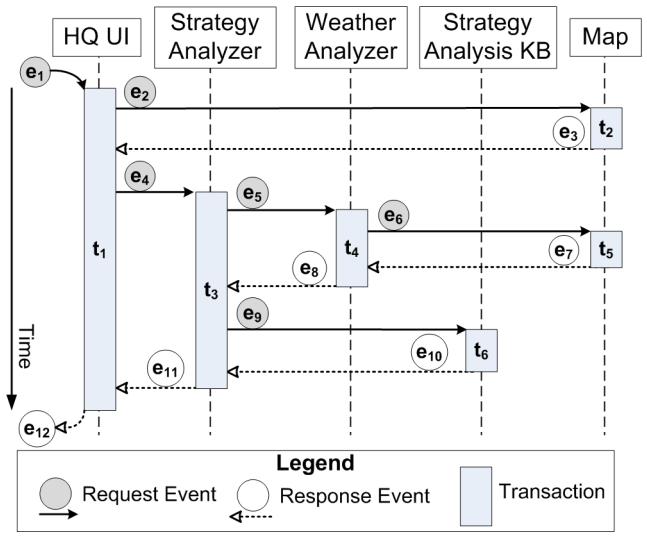

Figure 2: Transactions comprising strategy analysis scenario of EDS.

and destination, are observable.

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 transaction, initiated in response to e_1, which although not depicted in the figure represents the user requesting a service from the system via its GUI.

Replacing a component in the middle of a transaction could place the system in an inconsistent state. Consider a situation in which *Strategy Analyzer* component of Figure 2 is replaced after sending 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.

Even if the component is stateless, inconsistency problems may arise. Consider a stateless compression component that compresses and decompresses data using two interfaces that are reverses of one another. Replacing this component with one that uses a different type of compression algorithm in the middle of a transaction could break the system's functionality, since the decompression cannot be performed on data that was compressed using the old component. By the same reasoning, state transfer in the case of stateful components is not sufficient to address inconsistency due to adaptation.

Three general approaches to this problem have been proposed: *quiescence*, *tranquility*, and *version-consistency*.

Quiescence [13] 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 dependency 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 dependency 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 [21] proposes to use the *dynamic component dependency 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 [21], was lack of support for handling dependent transactions. This issue was addressed in version-consistency [16], which guarantees a dependent transaction is served by either the old version or new version of a component that is being changed.

4 Motivation and Objectives

Similar to the prior research [16, 21], we believe using static dependency models for achieving consistency to be overly disruptive in most cases. However, unlike prior research, we do not assume the availability of dynamic component dependency models (e.g., UML Sequence Diagram) for the following reasons.

- *Manually Intensive:* Dependency models are not always available and do not come for free. To develop these models, one has to understand the internal logic of components, which is a manual, cumbersome process, specially if the developer of those

models is not part of the team that implemented those components.

- *Dynamism and Evolution:* Determining the dependencies prior to system's deployment in emerging and increasingly dynamic paradigms, such as service-oriented and mobile domain, is difficult. As the system evolves, the internal logic of its components changes, making the manually constructed models inaccurate representations of the system, which if used for making adaptation decisions may break the system's consistency. Therefore, even when dependency models are available, keeping them up-to-date is a challenge.
- *Non-determinism:* Finally, and perhaps most importantly, component dependencies are often *non-deterministic*, i.e., a component depends on another component under some circumstances, but not others. The model depicted in Figure 2 is deterministic, since it assumes the transaction t_1 always results in the same exact sequence of subsequent events and transactions. No prior research has developed mechanisms for ensuring consistency and managing disruption in a non-deterministic setting.

In this research we aim to infer the *stochastic component dependency* 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. Thus, our approach does not compete with the prior research (i.e., tranquility and version-consistency), but rather paves the way for those techniques to be applicable in settings where dynamic dependency models are not available.

To keep our approach widely applicable, we make minimal assumptions about the available information from the underlying system. These assumptions are the same as those made in the prior research:

1. *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 COTS components, whose source code is not available. It also enables our approach to naturally support the evolution of software components.
2. *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.
3. *Observability of Transaction Duration*: We assume events *start* and *end*, which as you may recall from Section 3 indicate beginning and termination of a transaction, to be observable. This is a reasonable assumption that has also been made by all prior research [13, 21, 16]. For instance, in the example of Figure 2, the *HQ UI* component should be able to determine and record the occurrence of dependent transaction t_1 in terms of request e_1, which corresponds to the user clicking on a button on the GUI, and its termination via response e_{12}, which corresponds to the results to be displayed on the GUI.

Our approach makes no further pertinent assumptions and requires no additional information from the system. Based on this minimal information, our objective is to infer the stochastic component dependency model of the system. The crux of this is the ability to identify the causal relationship among the transactions. In other words, our objective is to determine the set R for every transaction occurring in the system (recall the formal definition of transaction in Section 3). This is a challenging problem to solve by simply monitoring the system, given that there may be multiple concurrently running top-level transactions at any point in time using the same set of components. Moreover, components in our approach could act non-deterministically, producing different behaviors under different conditions.

5 Approach Overview

We present a novel approach for automatically deriving the stochastic component dependency model by mining the execution history of the software system. The result of mining is a set of rules expressing the probabilistic relationship among the occurrences of transactions in the system. This set of rules represents our stochastic component dependency model. Given a set of active transactions in the system, these rules can be used to predict the probability with which a component can be changed at a point in time without jeopardizing the system's functionality, while minimizing the interruptions. Additionally, by continuously monitoring the transactions and the accuracy of predictions, the approach provides the means to adjust the rules as new patterns of interaction emerge.

Figure 3 provides an overview of our approach, consisting of two complementary asynchronously running cycles: *Mining Rules* and *Applying Rules*.

The *Mining Rules* cycle starts by 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 *Transaction Association Rules (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* the generated rules to arrive at a small number of useful rules that can be applied efficiently at runtime.

The *Applying Rules* cycle starts with the *Track Active Transactions* activity that monitors the currently run-

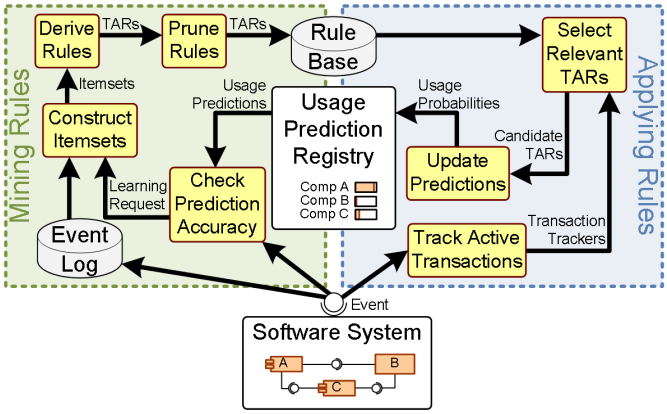

Figure 3: Approach overview.

ning 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 usage probability of components. *Update Predictions* uses candidate TARs to update the *Usage Prediction Registry*, which 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. 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 our approach.

6 Mining Rules

This section describes the *Mining Rules* cycle (recall Figure 3). This cycle runs 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.

6.1 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 our approach 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 Section 3 that consistent with the prior work [13, 21, 16], 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.

6.2 Constructing Itemsets

The first step to mining the relationship among the transactions is to *Construct Itemsets* (see Figure 3). 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: $I = \{t_1, t_2, ..., t_n\}$.

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 set of itemsets representing the occurrence of transactions together in time. Given a sufficiently large usage history, the approach compensates for concurrently running top-level transactions. 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. The approach will 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 data-mining algorithm using minimum frequency thresholds.

6.3 Deriving Rules

Several data mining approaches [11] 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 *transaction association rules (TARs)*. TARs are probabilistic rules for predicting the occurrence of transactions as follows $X \rightarrow Y : p$. A TAR states that the occurrence of set of transactions X implies the occurrence of a set of transactions Y with probability p. As shown in Figure 3, 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:

$$p = \frac{\sum_{s_i} \begin{cases} 1 & \text{if } X \in s_i \wedge Y \in s_i, \\ 0 & \text{otherwise.} \end{cases}}{\sum_{s_i} \begin{cases} 1 & \text{if } X \in s_i, \\ 0 & \text{otherwise.} \end{cases}}$$

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*:

$$s = \frac{\sum_{s_i} \begin{cases} 1 & \text{if } X \in s_i \wedge Y \in s_i, \\ 0 & \text{otherwise.} \end{cases}}{Number of Itemsets}$$

While support is not appropriate for the value of p in TARs, 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 6.2) 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 3, the *Derive Rules* step terminates by passing the raw set of generated TARs to *Prune Rules*.

6.4 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, i.e., we do not filter out many TARs based on the confidence level. We take this approach contrary to many other applications of associate rule mining because, while a TAR having a small p expresses less confidence in the prediction than does a different TAR having a larger p, both predictions are accurate and can be used in unison as explained in Section 7.3.

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 an optimum set of highly predictive rules that can be applied efficiently at runtime. To that end, and as depicted in Figure 3, the *Derive Rules* step terminates by passing the raw set of generated TARs to *Prune Rules*. There are three highly effective heuristics that we have developed for pruning the TARs.

(1) Redundant TAR Pruning Heuristic: Consider TARs satisfying this pattern:
$TAR_1 : X_1 \rightarrow Y_1 : p_1$
$TAR_2 : X_2 \rightarrow Y_2 : p_2$
where $(X_2 \subseteq X_1) \wedge (Y_1 = Y_2) \wedge (p_1 = p_2)$
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, i.e., X_2 is a subset of X_1. As will be explained in Section 7.2, 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:
$TAR_1 : X_1 \rightarrow Y_1 : p_1$
$TAR_2 : X_2 \rightarrow Y_2 : p_2$
$TAR_3 : X_3 \rightarrow Y_3 : p_3$
where $(X_1 = X_2 = X_3) \wedge (Y_1 = Y_2 \cup Y_3)$
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 $X_1 = X_2 = X_3$. 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 our approach.

At the completion of this activity a small subset of generated rules remains, which is stored in the *Rule Base* and used for runtime prediction of component usage.

7 Applying Rules

In this section, we describe the activities comprising the *Applying Rules* cycle from Figure 3.

7.1 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, i.e., 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.

Algorithm 1: Building the candidate TARs for application at runtime.

```
1 foreach tar ∈ Rule Base do
2     if t_o ∈ tar.X then
3         foreach tlt ∈ {TLT_1, ..., TLT_n} do
4             if (tar.X ⊆ tlt) ∧ (tar.Y − tlt ≠ ∅) then
5                 add tar to CTAR
```

7.2 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 use Algorithm 1, which iterates over all TARs in the *Rule Base* (Line 1). A $tar \in RuleBase$ can only be implicated by the observation of t_o, if t_o is a member of set X of that *tar* (Line 2). 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 (Line 3). For a *tar* to be satisfied, all transactions in X must have been observed during the processing of at least one *tlt* (first condition in Line 4). 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* (second condition in Line 4). Stated differently, a *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 $CTAR$ (Line 5), 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 presence of a *tar* in CTAR (Line 4) is said to be a *basis* for the application of that *tar*. Although not shown in Line 5 for the sake of simplicity, this basis information is tracked along with the *tar* and used in the next stages.

7.3 Updating the Usage Prediction Registry

In the previous sections, we described the process for generating the set $CTAR$ in response to a single observation $t_o.start$. The next step is to apply these TARs to update the *Usage Prediction Registries*, represented as set UP. Given a component c, there are typically more than a single TAR predicting its usage probability $u_c \in UP$. While some may be due to the new observation $t_o.start$, others may be due to the prior observa-

tions. Therefore, we must combine the various p values from all of the satisfied TARS into a single prediction value u_c.

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.

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., $CTAR_c = CTAR_c - ATAR_c$).

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

$TAR_1 : X_1 \rightarrow Y_1 : p_1$
$TAR_2 : X_2 \rightarrow Y_2 : p_2$
where $(Y_1 = Y_2) \wedge (X_1 \subseteq X_2)$

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).

(3) Selecting the best candidate: Even after removing the redundantly overlapping 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 the same component. Consider, for example, 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.

(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 are more than one active top-level transactions, we cannot know with certainty which top-level transaction actually initiated $t_o.start$. However, based on the number of active TLTs (recall Section 7.1), 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 number of active TLTs.

(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 $1 - p_i$.

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.

8 Using Registry for Adaptation

The ultimate goal in our research is to use the predictions for making adaptation decisions. The probabilistic rules inferred using our approach collectively represent the stochastic dependency model of the system. Such a model could be used in the context of both tranquility [21] and version-consistency [16] 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. Our approach could be used to both *guarantee consistency* and *minimize disruption* as described in detail below.

8.1 Guarantying Consistency

As specifically noted in Section 3, 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 very nearly 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 our approach as described up to this point would not fully guarantee consistency, because after all one cannot guarantee the accuracy of mined rules.

In situations where such a risk is unacceptable, we make a slight modification to the approach described in Section 7.3 that allows us to provide consistency guarantees. When we observe a transaction t_o, where $t_o.start.dst = c$, we lock the value of $u_c = 1$ to prevent c from being adapted, since we now know it has participated in a transaction, and changes to it may result in inconsistencies. However, since we do not know in which top-level transaction it has participated (i.e., we do not know the TLT), we keep the prediction locked at 1 until all of the TLTs that are the basis of that prediction have closed, at which point we roll back to the mechanism described in Section 7.3 for updating its prediction value.

8.2 Minimizing Disruption

As will be shown in Section 9, our predictions are highly accurate, allowing us to avoid changing components when they are likely to be used, thus reducing the disruption in the system. Recall from the previous section that 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 condition for adapting c. It may, however, take a long time for u_c to become 0 and this could create a *reachability problem*, i.e., a situation in which one has to wait a long time, or even forever, before the condition for adaptation is met. In practice, it is often reasonable to accept the potential for a slight disruption and allow adaptation when $u_c < \epsilon$.

Figure 4 exemplifies how this approach works. Although the examples are hypothetical, the registries indeed behave similarly 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 the registries are updated. Finally, when the rules are accurate, we expect the step function to be skewed to the right when the component is eventually used, and skewed to the left 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 4, 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 4b.

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 for a given system. We believe r is a reasonable threshold that can be specified by the user, e.g., 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 make a trade-off between reachability and disruption. To be able to calculate ϵ based on r, we have to relate a system-wide threshold defined by r to a component-specific threshold defined by ϵ. For that, we build a probability distribution for prior usage predictions and then use it to calculate ϵ based on r.

We calculate ϵ in terms of r by looking at the past predictions embodied in the recorded u_c values. However, r represents the probability of disruption in the system, whereas u_c represents the probability that a component c could be used, which may or may not result in a disruption. In order to determine a value for ϵ, we must relate our predictions u_c to the value of r. However, they not only have different *semantics* (i.e., one representing disruption probability and the other usage probability), but also different *domains* (i.e., one is in terms of component and the other system).

We let U_a represent the set of all recorded predictions for components that were eventually used (active),

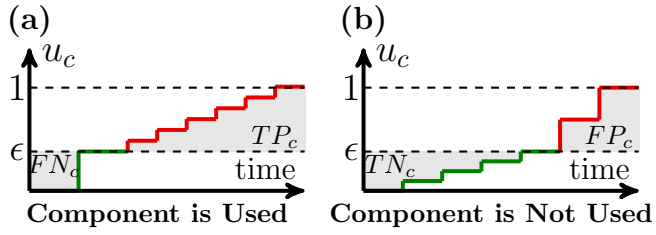

Figure 4: Hypothetical behavior of u_c as $t_o.start$ is observed for some top-level transactions over time: (a) c is eventually used, and (b) c is not used.

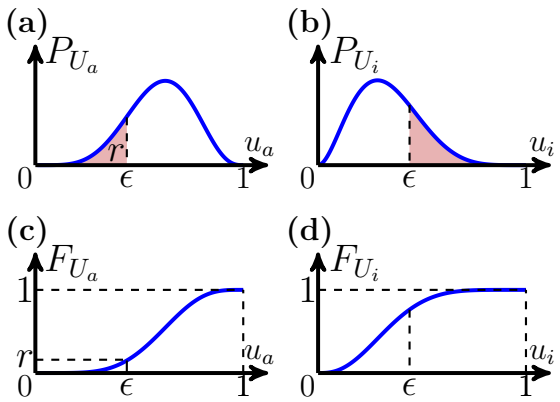

Figure 5: (a) Frequency distribution for U_a, (b) Frequency distribution for U_b, (c) CDF for U_a, and (d) CDF for U_i.

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 values corresponding to the step function of Figure 4a for all components in the system, while U_i represents the same except for Figure 4b. As a result, U_a indicates the situations in which adaptations could have possibly disrupted the system in the past. So now U_a and r have the same semantic, but to be able to relate them, we also need them to have the same domain.

To change the domain, we build a probability distribution by collecting a large sample of u_a values from U_a. Based on this we build the frequency distribution for U_a, which provides a system-wide representation of those predictions. A hypothetical probability distribution for U_a is shown in Figure 5a as P_{U_a}, while an analogous version for U_i as P_{U_i} is shown in Figure 5b (though ϵ is not known when these are first built). From this frequency distribution and using conventional techniques [2] we can derive the cumulative distribution function (CDF), which we call F_{U_a}, an example of which is depicted in Figure 5c. The analogous version for U_i is shown in Figure 5d. In this CDF function, $F_{U_a}(\epsilon)$ defines the fraction of all 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 [2]. We can see from Figure 5c that ϵ can be determined by finding the intersection of r and F_{U_a} and tracing down to the x-axis. This value of ϵ can then be transferred to F_{U_i}, P_{U_a}, and P_{U_i} as shown in Figure 5. We can thus see that all of P_{U_a} to the left of ϵ is a false negative and equal in quantity to r, while all of P_{U_i} to the right of ϵ is a false positive. This can therefore be seen as setting the ϵ cut-off between FN and TP in Figure 4a and between TN and FP in Figure 4b. The CDF is updated periodically based on the recent execution profile of the system.

It should be apparent from Figure 5a and b that the key to limiting error in the approach is to skew P_{U_a} towards high values of u and P_{U_i} towards 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 Figure 5c and d based on the slight difference in skewing shown in P_{U_a} and P_{U_i} in Figure 5a and b. If the approach is able to skew the distributions for active and inactive components differently, then it effectively achieves the real goal of this approach: it distinguishes between active and inactive components in advance. The following section discusses how well our approach achieved this aim.

9 Evaluation

We have developed a prototype of the approach using *Apriori*—an association rule-mining algorithm with an implementation provided in WEKA [10]. As explained in Section 6.4, 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 the approach, we used several versions of EDS as shown in Table 1. We used a baseline version of EDS with a single user. We then repeated the evaluations on higher and higher concurrency systems to evaluate the susceptibility of the approach to concurrency errors. The 80 and 137 user 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. As previously discussed, concurrency in the system means that an observed transaction must be attributed to all open top-level transactions. This in turn results in duplication of the observed transaction for each top-level transaction. Table 1 shows what percentage of all recorded transactions were actually these erroneous duplicates, as well as the average number of these erroneously recorded transactions per top-level transaction. Each experiment had roughly 8 true transactions per top-level transaction.

Table 1: Experimental systems used in evaluation, and effects of TAR pruning heuristics.

# of	TL Trans.	Concurrency Errors		Number of TARs	
Users	Observed	Rate	Per Itemset	Initial	Remained
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

Table 2: Error and accuracy rates for the experimental systems.

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

9.1 Effectiveness of TAR Reducing Heuristics

We first show the effectiveness of our rule pruning heuristics (recall Section 6). 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 1. 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 9.4).

9.2 Accuracy of Component Usage Predictions

A crucial evaluation dimension for our approach is the degree to which it correctly predicts the usage of a component. As discussed in Section 8.2, 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. Figure 6a and 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 our approach achieved significant differentiation between active and inactive components.

Using these CDFs, we can quantify the effectiveness of the approach in terms of FN, TP, TN, and FP. As discussed in Section 8.2, the approach uses ϵ to fix the FN rate at r. Therefore, the effectiveness of the approach must be measured in its ability to minimize the FP rate based on the fixed value of FN. Because our approach 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 favourable error rates as shown in Table 2. As seen, the unfixed error rate of FP was held to below 7% in all experiments except for that with the highest concurrency, well below the fixed FN error rate. Beyond demonstrating accuracy in the prediction of component activity, these ratios also demonstrate that the approach was not noticeably impacted by an increase in concurrency in the system until concurrency reached extreme levels.

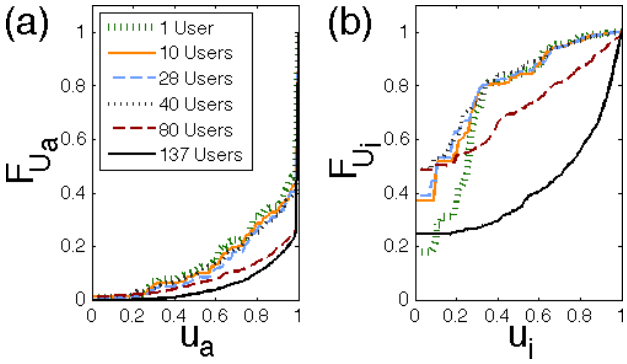

Figure 6: The results from the experiments: (a) CDF of U_a and (b) CDF of U_i.

This quality of differentiation can be further viewed with a ROC curve as shown in Figure 7. ROC curve is a conventional approach for evaluating classifiers [7, 20]. 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 the approach does an incredible job of achieving true positives despite changes in the ϵ threshold. The approach achieves greater than 0.7 true positive rate with nearly every level of false positive rate.

The comparison of the different experiments in Table 2 and Figure 7 also serves to show the way the approach compensates for concurrency and how high levels of concurrency eventually prevent the approach from compensating further. As seen in Table 2, higher values for ϵ are needed to achieve $r = 0.20$ as concurrency increases. This occurs because, with many users in the

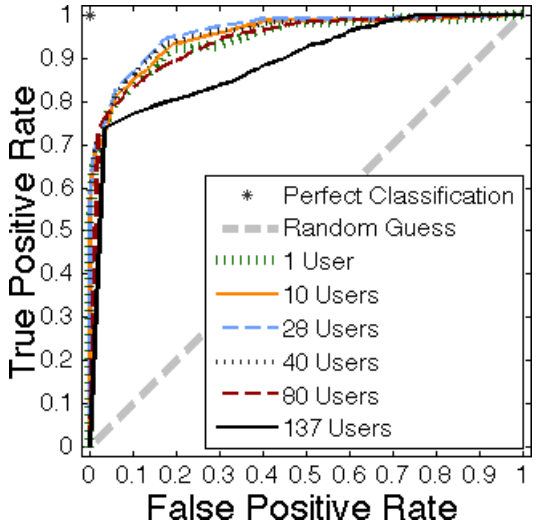

Figure 7: ROC Curve for the various experiments.

system, there are many more observations that allow the approach to predict usage of a component c, when c is actually used. Therefore, as concurrency increases, the values for u_a are more skewed towards 1.0 until, at a concurrency error rate of roughly 60% for EDS (i.e., case of 137 users), active components are constantly at $u_c = 1.0$ until the transactions they participate in subside. While this is beneficial because it approaches perfect classification of active components (as can be seen in Figure 7, higher concurrency systems actually escalate to the (0 , 1) point more directly), it results in two detriments to the approach.

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, the approach 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 our approach based on version-consistency [16] 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. 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. While our current implementation is based on tranquility, which simply buffers events, there is nothing fundamentally different from applying the version-consistency model of adaptation that does not have such limitations.

9.3 Accuracy of Desired Disruption Rate

The third point of evaluation is the degree to which the approach achieves the desired r rate of disruption during adaptation. The evaluation results presented in the previous section and shown in Table 2 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 3 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.

Table 3: 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]

9.4 Performance and Timing

The final evaluation criteria are the performance benchmarks of *Mining Rules* and *Applying Rules* cycles. We have collected these numbers on a MacBook pro laptop with 2.53 GHz Intel Core i5 processor and 4 GB 1067 MHz DDR3 memory. 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 2 seconds in all of the experiments described here.

The performance of updating the predictions at runtime consists of two primary elements: retrieval of relevant TARs (recall $CTAR$ from Section 7.2) 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 1.355 seconds and 0.959 seconds, we implemented a 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 4. 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.

10 Related Work

In Section 3, we described the most related approaches, namely quiescence [13], version-consistency [16], and tranquility [21], including their relationship to this work. Here we focus on other related literature.

Table 4: 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]

Researchers have used log of event data collected from a system to construct a model of it for various purposes. Cook et al. [4] use the event data generated by a software process to discover the formal sequential model of that process. In a subsequent work [5], they have extended their work to use the event traces for a concurrent system to build a concurrency model of it. Gaaloul et al. [8] 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 BPEL. Motahari-Nezhad et al. [18] presents an algorithmic approach for correlating individual events, which are scattered across several systems and data sources, semi-automatically. They use these correlations to find the events that belong to the same business process execution instance. Wen et al. [22] use the start and end of transactions from the event log to build petri-nets corresponding to the processes of the system. None of these approaches aim to understand the behavior of the system for the purpose of adaptation.

Software architecture has been shown to provide an appropriate level of abstraction and generality to deal with the complexity of dynamically adapting software systems [14, 19]. Gomaa and Hussein [9] developed the notion of reconfiguration pattern, which is a repeatable sequence of steps for placing a software component in the quiescence state. In a recent work [6], we adopted this concept to provide safe adaptation support on top of a middleware platform.

Finally, mining software repositories (e.g., source control systems, bug tracking systems, etc.) is a thriving thrust of research (see [1]). Although related, our objective in this paper is fundamentally different from that line of research. To the best of our knowledge, data mining has not been applied for determining the time at which changes should occur in a running software system.

11 Conclusion

We provided an overview of a mining-based approach that from the execution history of a software system infers a stochastic component dependency model of its components. We have used this model for the purpose of determining the time at which a component can be replaced without leaving the system in an inconsistent state and creating a significant disruption. Our approach can be applied with minimal effort. All that is needed is the ability to monitor the interactions among the components of that system. Our approach can be used to learn the emerging component dependencies as the software system evolves. The evaluation of our approach using a real software system and numerous users have empirically shown the accuracy of the models inferred in this way, and the ability to leverage those models to make timely and effective adaptation decisions.

In the future, we plan to experiment with other types of data mining algorithms that leverage frequency of item occurrence, as well as temporal and ordering relationship among events. With these sorts of approaches, we hope to leverage information that is already available, but not effectively utilized in our current approach to further improve its precision and performance.

12 Acknowledgments

This work is partially supported by grant CCF-0820060 and CCF-1217503 from the National Science Foundation and grant N11AP20025 from Defense Advanced Research Projects Agency.

References

[1] Mining software repositories. http://msrconf.org/.

[2] Bertsekas, D. P., and Tsitsiklis, J. N. *Introduction to Probability, 2nd Edition.* Athena Scientific, July 2008.

[3] Cheng, B. et al. Software engineering for Self-Adaptive systems: A research roadmap. In *Software Engineering for Self-Adaptive Systems, LNCS Hot Topics.* 2009, pp. 1–26.

[4] Cook, J. E., and Wolf, A. L. Discovering models of software processes from event-based data. *ACM Trans. Softw. Eng. Methodol. 7*, 3 (July 1998), 215–249.

[5] Cook, J. E., and Wolf, A. L. Event-based detection of concurrency. In *Int'l Symp. on the Foundations of Software Engineering* (Lake Buena Vista, Florida, Nov. 1998), pp. 35–45.

[6] Esfahani, N., and Malek, S. Utilizing architectural styles to enhance the adaptation support of middleware platforms. *Journal of Information and Software Technology* (2012).

[7] Fawcett, T. An introduction to ROC analysis. *Pattern Recogn. Lett. 27*, 8 (June 2006), 861–874.

[8] Gaaloul, W., Baina, K., and Godart, C. Log-based mining techniques applied to web service composition reengineering. *Service Oriented Computing and Applications 2*, 2-3 (May 2008), 93–110.

[9] Gomaa, H., and Hussein, M. Software reconfiguration patterns for dynamic evolution of software architectures. In *Working IEEE/IFIP Conf. on Software Architecture* (Oslo, Norway, June 2004), pp. 79–88.

[10] Hall, M. et al. The WEKA data mining software: an update. *SIGKDD Explor. Newsl. 11*, 1 (Nov. 2009), 10–18.

[11] Han, J., and Kamber, M. *Data mining: concepts and techniques*. Morgan Kaufmann, 2006.

[12] Kephart, J. O., and Chess, D. M. The vision of autonomic computing. *IEEE Computer 36*, 1 (Jan. 2003), 41–50.

[13] Kramer, J., and Magee, J. The evolving philosophers problem: Dynamic change management. *IEEE Trans. Softw. Eng. 16*, 11 (Nov. 1990), 1293–1306.

[14] Kramer, J., and Magee, J. Self-Managed systems: an architectural challenge. In *Int'l Conf. on Software Engineering* (Minneapolis, Minnesota, May 2007), pp. 259–268.

[15] Lemos, R. d. et al. Software engineering for Self-Adpaptive systems: A second research roadmap. In *Software Engineering for Self-Adaptive Systems* (Dagstuhl, Germany, June 2011), R. d. Lemos, H. Giese, H. Muller, and M. Shaw, Eds.

[16] Ma, X., Baresi, L., Ghezzi, C., Panzica La Manna, V., and Lu, J. Version-consistent dynamic reconfiguration of component-based distributed systems. In *Int'l Symp. on the Foundations of Software Engineering* (Szeged, Hungary, Sept. 2011), ACM, pp. 245–255.

[17] Malek, S., Mikic-Rakic, M., and Medvidovic, N. A Style-Aware architectural middleware for Resource-Constrained, distributed systems. *IEEE Trans. Softw. Eng. 31*, 3 (Mar. 2005), 256–272.

[18] Motahari-Nezhad, H. R., Saint-Paul, R., Casati, F., and Benatallah, B. Event correlation for process discovery from web service interaction logs. *The VLDB Journal 20*, 3 (June 2011), 417–444.

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

[20] Tan, P., Steinbach, M., and Kumar, V. *Introduction to Data Mining*, 1 ed. Addison Wesley, May 2005.

[21] Vandewoude, Y., Ebraert, P., Berbers, Y., and D'Hondt, T. Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates. *IEEE Trans. Softw. Eng. 33*, 12 (Dec. 2007), 856–868.

[22] Wen, L., Wang, J., Aalst, W. M., Huang, B., and Sun, J. A novel approach for process mining based on event types. *J. Intell. Inf. Syst. 32*, 2 (Apr. 2009), 163–190.