Ensuring the Consistency of Adaptation through Inter- and Intra-Component Dependency Analysis

ALIREZA SADEGHI, University of California, Irvine
NAEEM ESFAHANI, Google Inc.
SAM MALEK, University of California, Irvine

Dynamic adaptation should not leave a software system in an inconsistent state, as it could lead to failure. Prior research has used inter-component dependency models of a system to determine a safe interval for the adaptation of its components, where the most important tradeoff is between disruption in the operations of the system and reachability of safe intervals. This article presents Savasana, which automatically analyzes a software system's code to extract both inter- and intra-component dependencies. In this way, Savasana is able to obtain more fine-grained models compared to previous approaches. Savasana then uses the detailed models to find safe adaptation intervals that cannot be determined using techniques from prior research. This allows Savasana to achieve a better tradeoff between disruption and reachability. The article demonstrates how Savasana infers safe adaptation intervals for components of a software system under various use cases and conditions.

CCS Concepts: • **Software and its engineering → Software architectures**; *Maintaining software*; Automated static analysis

Additional Key Words and Phrases: Adaptive software, component-based software, update criteria

ACM Reference Format:
Alireza Sadeghi, Naeem Esfahani, and Sam Malek. 2017. Ensuring the consistency of adaptation through inter- and intra-component dependency analysis. ACM Trans. Softw. Eng. Methodol. 26, 1, Article 2 (May 2017), 27 pages.
DOI: http://dx.doi.org/10.1145/3063385

1. INTRODUCTION

An increasingly important requirement for modern software systems is the ability to *dynamically adapt*, that is, change parts of the software as it executes. The construction of dynamically adaptive software is significantly more challenging than traditional software systems [Cheng et al. 2009; Lemos et al. 2011]. One important challenge is the management of runtime changes to avoid *inconsistencies* during and after the adaptation.

An *inconsistent* application state is one from which the system progresses toward an error state [Kramer and Magee 1990]. A classical example used to illustrate this issue is the *evolving philosophers* problem, a variant of the well-known *dining philosophers*

This work was supported in part by awards CCF-1252644, CNS-1629771, and CCF-1618132 from the National Science Foundation; D11AP00282 from the Defense Advanced Research Projects Agency; W911NF-09-1-0273 from the Army Research Office; HSHQDC-14-C-B0040 from the Department of Homeland Security; and FA95501610030 from the Air Force Office of Scientific Research.
Authors' addresses: A. Sadeghi and S. Malek, Department of Informatics, School of Information and Computer Sciences, University of California, Irvine, CA 92697; emails: {alirezs1, malek}@uci.edu; N. Esfahani, Google Inc., 1600 Amphitheatre Parkway Mountain View, CA 94043; email: naeeem@google.com.

DOI: http://dx.doi.org/10.1145/3063385

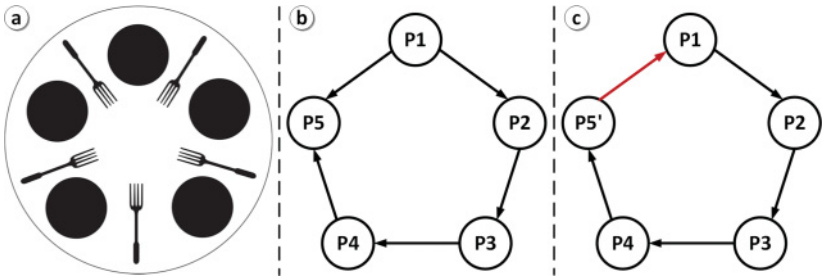

Fig. 1. (a) Dining Philosophers Problem: (b) a consistent situation and (c) an inconsistent (deadlock) situation after replacing $P5$ with $P5'$.

problem. As shown in Figure 1(a), philosophers wait on their neighbors to obtain the two forks needed to be able to eat. Here, philosopher is a metaphor for a software component and a wait-for-fork relationship is a metaphor for component dependency. In the context of this problem, a consistent application state requires there to be no cycle in the dependency relationship among the philosophers, indicating that at any given point in time at least one philosopher is eating, as shown in Figure 1(b). The important observation is that while philosophers are eating, their internal state may be mutually inconsistent [Kramer and Magee 1990]. Dynamic replacement of a philosopher could lead to an inconsistent state. Figure 1(c) demonstrates the case, where the philosophers are deadlocked (i.e., there is a cycle) after replacing $P5$ with $P5'$.

Quiescence [Kramer and Magee 1990] solved this problem using a *static inter-component dependency* model of a system (e.g., UML Component Diagram) by determining the components that have to be halted (passivated) before a component can be safely adapted. Reliance on the static inter-component dependency model, however, makes quiescence rather pessimistic in its analysis and leads to significant disruptions.

Tranquility [Vandewoude et al. 2007] and Version-Consistency [Ma et al. 2011] showed that by leveraging a *dynamic inter-component dependency* model of a system (e.g., UML Sequence Diagram), it is possible to become more refined in the analysis and thus increase the speed with which adaptations are effected.

All of these approaches, however, share a common trait: They have adopted a *black-box approach*, that is, they have abstracted the analysis to the system's software components and their dependencies on one another. We found that since the black-box approaches have abstracted from the state of the application logic, they are susceptible to miss opportunities for consistent adaptation of software and thus enact changes later than needed.

This article presents *Savasana*, the first *white-box approach* for ensuring consistency during dynamic adaptation of software.[1] Unlike the black-box approaches, Savasana determines when a component can be adapted by considering both its internal behavior as well as its interactions with other components. To that end, Savasana statically analyzes the implementation of a component to derive a detailed model of its behavior, referred to as *intra-component dependency model*. At runtime, Savasana uses both intra- and inter-component dependency models to ensure the changes do not place the system in an inconsistent state.

Since Savasana is white-box, it has more information at its disposal, including how internal elements of components are defined and used. It leverages this information to identify more opportunities for consistent adaptation of software than what is possible with prior techniques. In other words, Savasana enacts the changes in the software

[1]Savasana is a yoga pose in which the body is completely relaxed.

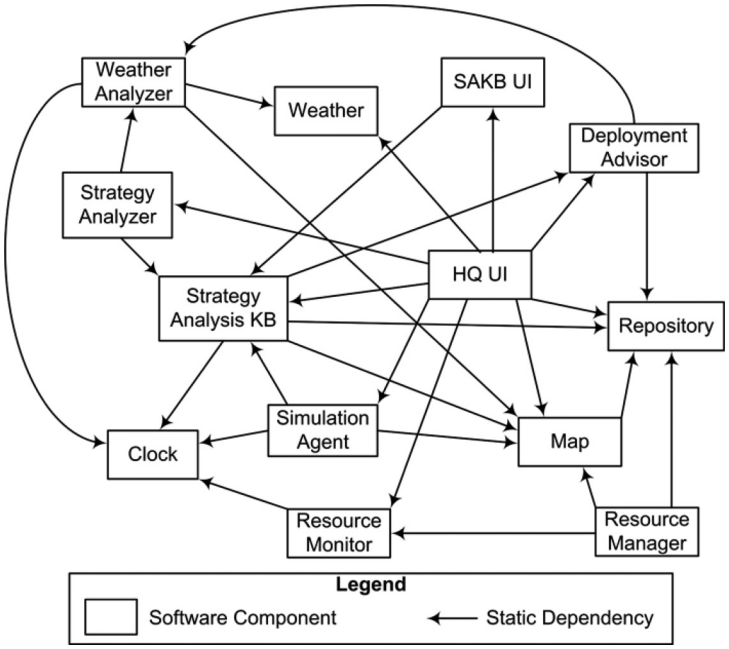

Fig. 2. Subset of the Emergency Deployment System's software architecture.

faster than prior approaches. In addition, Savasana overcomes the complexity of manually sifting through the implementation logic to determine the dependencies in the software through a fully automated static program analysis capability. This way, Savasana is able to automatically derive up-to-date models for consistent adaptation of software and update the models as the software evolves.

We have developed an implementation of Savasana for applications built on top of the Spring framework—a widely used commercial technology for the development of component-based software in Java. Our experiences with thorough evaluation of this approach in the context of a large software system have been very positive. The results corroborate our ability to infer precise models that can be used to effect changes in the running software faster than prior research.

The remainder of the article is organized as follows. Section 2 describes a software system used to illustrate and evaluate the research. Section 3 provides an intuitive overview of the approach. Section 4 presents the theoretical details of Savasana, while Section 5 describes its implementation for Java programs. Section 6 presents the evaluation of the research. Finally, the article concludes with an outline of the related research and the future work.

2. ILLUSTRATIVE EXAMPLE

For illustrating the concepts in this article, we use a software system called the Emergency Deployment System (EDS) [Malek et al. 2005; Malek 2007]. EDS is intended for the deployment and management of personnel in emergency response scenarios. Figure 2 depicts a subset of EDS's software architecture and in particular shows the dependency relationships among its components.

EDS is used to accomplish three main tasks: *Strategy Analysis (SA)* determines the tradeoffs among different deployment strategies using the *Strategy Analyzer* component, *Deployment Advising (DA)* recommends alternative deployments of resources and crew using the *Deployment Advisor* component, and *Resource Estimation (RE)*

```
 (a)Map Interface
public interface Map {
    public void initMap(double[] initLoc);
    public void updateLocation(double[] newLoc);
    public double[] retrieveLocation();
    public double getDistanceToTarget();
    // Updates time to the target by moving with the speed
    public void updateTimeToTarget(double[] targetLoc, double speed);
}
-----------------------------------------------------------
 (b)Map Implementation: Old Version
public class MapImpl implements Map {
    double timeToTarget;
    double latitude;
    double longitude;
  ...
  public void updateTimeToTarget(double[] targetLoc, double speed) {
    // Estimating distance to the target
    double distanceToTarget = Math.sqrt(Math.pow((targetLoc[0] - latitude), 2) + Math.pow((targetLoc[1] -
         longitude), 2));
    // Finding time to the target based on distance & speed
    timeToTarget = distanceToTarget / speed;
  }
}
-----------------------------------------------------------
 (c)Map Implementation: New Version
public class MapImpl implements Map {
    double timeToTarget;
    double latitude;
    double longitude;
  ...
  public void updateTimeToTarget(double[] targetLoc, double speed) {
    // Estimating distanceToTarget same as the Old Version
    double distanceToTarget = ...
    //Checks the speed to be not zero
    if (speed == 0) return;
    timeToTarget = distanceToTarget / speed;
  }
}
```

Listing 1. Map component's implementation in Java: (a) partial interface description, (b) old implementation of updateTimeToTarget interface, and (c) new implementation of updateTimeToTarget interface.

incrementally simulates the effect of allocating resources to rescue teams on the performance of rescue operations using the *Simulation Agent* component. The interested reader may find a more detailed description of EDS in Malek et al. [2005] and Malek [2007]. It suffices to say that EDS is representative of a large component-based software system. EDS could be deployed as either a distributed system on multiple machines or a centralized system on a single machine. While the focus of this article, including the examples and implementation prototype, is on a centralized (local) deployment, the theoretical contribution of our work is also applicable in a distributed setting.

It is often desirable to be able to adapt systems such as EDS at runtime to deal with changes that may affect the system's (non-)functional properties. For example, consider the following hypothetical scenario involving a fault in the implementation of a *Map* component that is discovered after the system has been deployed. Listing 1(a) shows the component's interfaces. As shown in Listing 1(b), the implementation of *updateTimeToTarget* interface, which estimates the required time to reach a target with a given speed, is susceptible to division by zero fault when *speed* is equal to zero. Dynamic adaptation provides the ability to deal with such problems by replacing the *Map* component with a new version that has the correct implementation of this interface as shown in Listing 1(c). Consistent with prior research, we assume two versions of the Map component provide the same interchangeable functionalities. This change, however, should occur in a manner that does not lead to inconsistency or significant

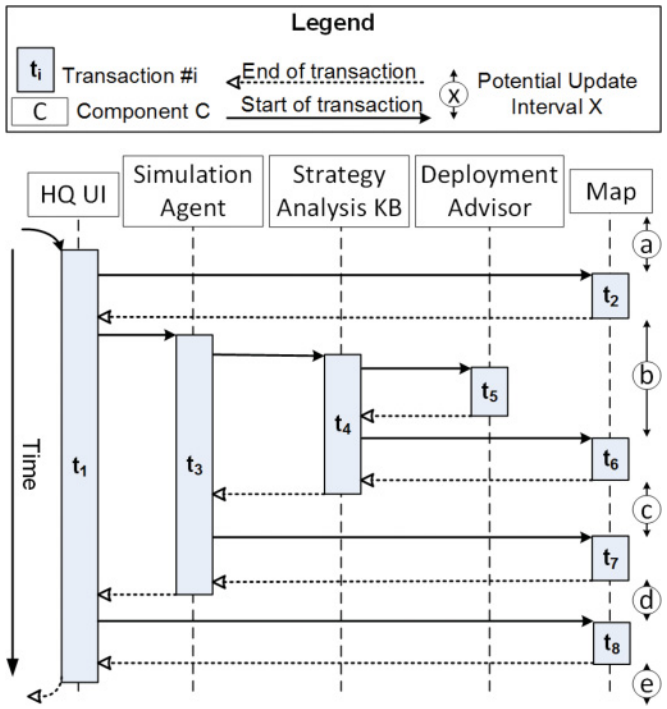

Fig. 3. Sequence of transactions comprising the RE scenario in EDS (i.e., `DeT(RE)`).

disruption in the services provided to the users. The following section provides an overview of the prior solutions to this problem, followed by an intuitive overview of Savasana's approach.

3. RESEARCH OVERVIEW

A system is generally viewed as moving from one consistent state to the next. Application transactions modify the state of the application, and while in progress, have transient state distributed in the system. In a component-based software, a *transaction* is exchange of information 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. A *root transaction* is a type of dependent transaction that corresponds to the functionalities (use cases) of the software. Figure 3 shows the root transaction corresponding to the Resource Estimation functionality of EDS. Applications are often encoded with a particular sequence of transactions to ensure the components are in a proper state for use. Consider, for example, a Map component that is first requested to load a proper geographical location or a Camera component that is first requested to point at a proper angle before use.

The goal of consistent adaptation is to ensure that changes do not lead to an unstable state of components participating in a transaction. Three general approachesto prevent inconsistency during adaptation have been proposed: Quiescence, Tranquility, and Version-Consistency. In this section, we provide a comparative analysis of these techniques as well as Savasana. Table I summarizes their key differences.

Quiescence [Kramer and Magee 1990] prevents inconsistency by ensuring that changes do not occur in the middle of an active transaction. To that end, Quiescence uses a *static inter-component dependency model* of a system, such as the model depicted in

Table I. Comparison between Four Adaptation Approaches

	Quiescence	Tranquility	Version-Consistency	Savasana
Adaptation	Add/Remove/Replace	Replace	Add/Remove/Replace	Replace
Model Type	Static	Dynamic	Dynamic	Dynamic
Model Extraction	Manual	Manual	Manual	Automatic
Model Analysis	Inter-Component	Inter-Component	Inter-Component	Inter- and Intra-Component
Root Transaction	No Identifier	No Identifier	Unique Identifier	No Identifier

Figure 2, to determine the components that should be passivated (halted). For instance, to dynamically replace the *Map* component, Quiescence requires all components that depend on the *Map*, and thus that may initiate transactions requiring its participation, to be passivated. From Figure 2, we can see that *HQ UI*, *Simulation Agent*, *Strategy Analysis KB*, *Weather Analyzer*, and *Resource Manager* would need to be passivated. Quiescence is pessimistic in its analysis and, hence, very disruptive. The reason for this is that a static inter-component dependency model (e.g., Figure 2) 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. Quiescence can be used to avoid inconsistency when replacing a component with another functionally equivalent component, as well as when adding a new or permanently removing a component.

To provide a less disruptive adaptation, another approach, Version-Consistency [Ma et al. 2011], assumes the existence of a *unique identifier* to distinguish among the different root transactions. It then uses the identifier to guarantee that all dependent transactions belonging to the same root transaction are served by either the old version or the new version of a component. Moreover, to realize version-consistency approach, Ma et al. proposed a management framework for distributed transactions to ensure the safe dynamic reconfiguration of component-based distributed systems.

Tranquility [Vandewoude et al. 2007] also aims to reduce the disruptions caused by adaptation. However, unlike Quiescence and Version-Consistency, the focus of Tranquility is only on the dynamic replacement of a component. Tranquility uses a *dynamic inter-component dependency model* of a system, such as that shown in Figure 3 for the Resource Estimation scenario of EDS. Under Tranquility, *a component can be safely replaced, unless it has already participated in an active transaction that it may participate in again* [Vandewoude et al. 2007].[2] For instance, in the scenario of Figure 3, Tranquility does not allow changing *Map* from the first time it is used (i.e., t_2) until the last time it is used (i.e., t_8). According to Tranquility, the only safe update intervals to change *Map* component are ⓐand ⓔ.

Similarly to Tranquility, Savasana uses the dynamic inter-component dependency model and focuses on dynamic replacement of components. But, unlike the other approaches, it also takes advantage of latent intra-component dependencies that can be extracted from a component's implementation. In contrast to other approaches that rely on manually constructed models, Savasana statically analyzes a component's implementation to derive a detailed model of its internal behavior. It then uses both inter- and intra-component models to identify safe update intervals.

[2]A more formal definition of Tranquility is provided in Section 4.2, Definition 2.

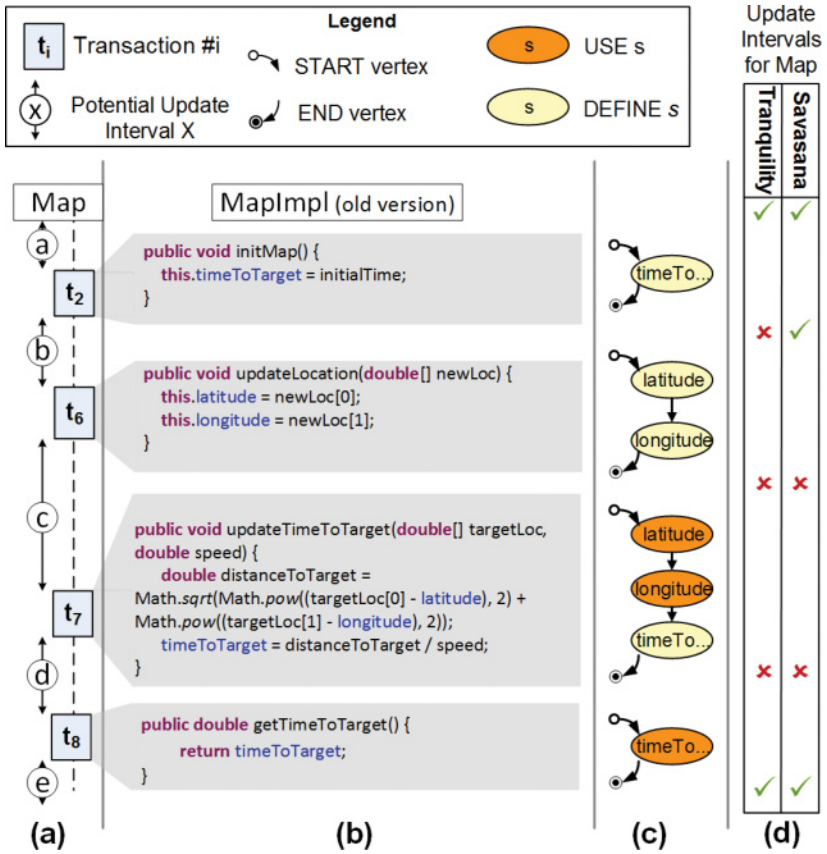

Fig. 4. A subset of the RE scenario: (a) Sequence of transactions defined on *Map* (i.e., DeT(*RE*, *Map*)); (b) the source code of *Map*'s interface corresponding to the transactions; and (c) Define Use Graph (DUG) of each transaction; (d) Safety of update intervals for Tranquility vs. Savasana.

Figure 4 intuitively illustrates Savasana's intra-component dependency analysis using the Resource Estimation scenario of EDS. Savasana extracts the *define* and *use* relationships among the state constructs comprising a component from its implementation. Under Savasana, *a component can be safely replaced if (1) it satisfies Tranquility or (2) it is in the middle of a transaction that it has already participated and is going to participate again, but the state constructs that are used in its future participation are redefined (reset) prior to use*. In the case of a Resource Estimation scenario in Figure 4, Savasana determines that update interval ⓑ is safe for the adaptation of the *Map* component, since the three state constructs (*latitude*, *longitude*, and *timeToTarget*) that are used in its future participation in the root transaction (i.e., t_1) are all defined again prior to use. Using more information allows Savasana to detect safe update intervals that otherwise would be missed. In the next section, we formally define Savasana and prove its ability to safely replace components at runtime.

4. SAVASANA

Savasana aims to identify the set of intervals in which the components comprising the system can be safely adapted. This section first formally defines the concepts used throughout the article and subsequently introduces Savasana's criteria for identifying the safe intervals.

4.1. Definitions

A transaction $t \in T$ is defined as a triple tuple $t = <s, e, c>$, where s and e, respectively, represent the start and end of t, while $c \in C$ is the component that hosts t. In different systems, s and e can be realized by different concepts (e.g., events, method calls, etc.). For each transaction in Figure 3, s is depicted by a solid arrow, and e is depicted by a dotted arrow.

To facilitate the understanding of formalism provided in this section, we define three functions applied to transactions $t \in T$, namely `DependentTransactions` or `DeT()`, `StateConstructs` or `StC()`, and `DefUseGraph` or `DuG()`.

For any given transaction $t \in T$, a component $c \in C$, and a positive integer number $i \in \mathbb{N}$, we define three variations of function `DeT()` as follows:

$$\texttt{DeT} : \begin{cases} T \rightarrow \mathcal{P}(T), & \textit{in function } \texttt{DeT(t)} \\ T \times C \rightarrow \mathcal{P}(T), & \textit{in function } \texttt{DeT(t,c)} \\ T \times C \times \mathbb{N} \rightarrow T, & \textit{in function } \texttt{DeT(t,c,i)} \end{cases}.$$

As shown in the function signature, the second and third parameters of `DeT()` are optional, and the presence of those parameters could change the output domain of the function. In the following, we describe the variations of `DeT()` through several examples.

For a given transaction $t \in T$, `DeT(`t`)` returns a sequence of transactions that t directly or indirectly depends on to complete execution. The returned results of `DeT(`t`)` are sorted transactions according to their start time. When t is an independent transaction (e.g., t_2, and t_5–t_8 in Figure 3), `DeT(`t`)` returns t as its only element; otherwise, when t is a dependent transaction (e.g., t_1, t_3, and t_4 in Figure 3), it returns more elements (i.e., $|\texttt{DeT(t)}| > 1$).

A *root transaction* t is a kind of transaction where there is no other transaction x in the system such that $t \in \texttt{DeT(x)}$. In other words, the occurrence of a root transaction is not tied to other transactions in the system. Root transactions correspond to the system's use cases (functional capabilities). For instance, t_1 in Figure 3 is a root transaction, initiated in response to a request by the user. We let $R \subseteq T$ to be the set of all root transactions.

For a given root transaction $r \in R$ and a component $c \in C$, we use function `DeT(`r, c`)` to return the sequence of transactions involving component c in the root transaction r. For instance, Figure 4(a) depicts the sequence of transactions defined on the *Map* component in the context of the *RE* scenario (i.e., `DeT(`RE, Map`)` $= <t_2, t_6, t_7, t_8>$).

Moreover, we use the third parameter i of function `DeT(`t, c, i`)` to retrieve the ith element of the transaction sequence that t depends on to complete execution and involving component c. For instance, in the example shown in Figure 4(a), `DeT(`RE, Map`,2)` $= t_6$.

For a given transaction $t \in T$ with the hosting component $c \in C$, and program's variables and data structures $s \in S$, we define function `StateConstructs` or `StC()` as follows:

$$\texttt{StC} : T \times C \rightarrow \mathcal{P}(S).$$

`StC()` returns c's state constructs, such as variables and data structures. For example, `timeToTarget`, `latitude`, and `longitude` are some of the state constructs of the Map component shown in Listing 1. For a component c, only a subset of its state constructs may be involved during a transaction t. Thus, we use `StC(`t, c`)` to distinguish the subset of state constructs of c that are used and defined during transaction t.

Finally, for a given transaction $t \in T$ with the hosting component $c \in C$, and program's *control flow graph (CFG)*, function DuG() is defined as follows:

$$\texttt{DuG} : T \times C \rightarrow \mathcal{P}(CFG).$$

$\texttt{DuG}(t, c)$ returns the *def-use graph* for the component $c \in C$ that hosts t. Def-use graph is a subset of the CFG for a transaction t. This subset is defined by only keeping those vertexes of CFG that deal with the state of component c, either by defining or using a *state construct* $s \in \texttt{StC}(t, c)$, denoted as def_s and use_s, respectively. Figure 4(c) shows the internal behavior of the *Map* component in the *RE* scenario by depicting Define Use Graph (DUG) of all transactions in $\texttt{DeT}(RE, Map)$ sequence. Here, yellow (light) oval represents define and orange (dark) oval represents use of a state variable in *Map*. For example, in $\texttt{DuG}(t_7, Map)$ depicted as the third row from the top in Figure 4(c) (i.e., $\texttt{DeT}(RE, Map, 3) = t_7$), *latitude* and *longitude* are used, and then *timeToTarget* is defined. Note that there are some other variables with limited scope, which are excluded from $\texttt{DuG}(t_7, Map)$, as they do not affect the state of *Map*. They include method parameters (e.g., *targetLoc*, *speed*) and local variables (e.g., *distanceToTarget*).

4.2. Savasana Criteria

A safe adaptation does not leave the system in an inconsistent state. As mentioned in Section 3, Savasana addresses inconsistencies due to dynamic replacement of a component—a situation where the state of the new instance of a component differs from the state of the retired instance of component only as a result of adaptation, which, as described in Section 3, may lead to system failure. In this section, we define Savasana's criteria for consistent adaptation of software.

We start by formal definition of replacement inconsistency:

Definition 1 (*Replacement Inconsistency*). Suppose that, in the context of root transaction r, component $c \in C$ is replaced with component $c' \in C$ after transaction $t_i = \texttt{DeT}(r, c, i)$, and before transaction $t_j = \texttt{DeT}(r, c, j)$, where $i \leq j$. If $\exists use_{s'} \in \texttt{DuG}(t_j, c')$, then any deviation between state constructs $s \in \texttt{StC}(t_j, c)$ and the corresponding state constructs $s' \in \texttt{StC}(t_j, c')$ at the time of $use_{s'}$ is called replacement inconsistency.

According to this definition, the replaced component may create an inconsistency if a subset of its state constructs, when used in the pending transactions comprising a root transaction, are in a different state compared to the original component.

Savasana has two criteria in specifying the safe update intervals for a component: (1) *inter-component update criterion* and (2) *intra-component update criterion*.

Savasana relies on Tranquility [Vandewoude et al. 2007] for detecting inter-component update intervals. The only difference is that Savasana does not assume the availability of an inter-component dependency model and extracts it through static program analysis. The inter-component update criterion is defined as follows:

Definition 2 (*Inter-Component Update Interval*). A component is in an inter-component update interval (tranquil interval) in the context of a root transaction if "(1) it is not currently engaged in a transaction that it initiated, (2) it will not initiate new transactions, (3) it is not actively processing a request, and (4) none of its adjacent nodes are engaged in a transaction in which it has both already participated and might still participate in the future" Vandewoude et al. [2007].

LEMMA 1. *Replacing a component in an inter-component update interval does not cause inconsistency.*

PROOF. The validity of Lemma 1 is shown in prior work [Vandewoude et al. 2007]. □

In Figure 3, inter-component update interval implies that the *Map* cannot be adapted after t_2 and prior to t_8 (i.e., in ⓑ, ⓒ, and ⓓ).

The intuition behind the intra-component update criterion in Savasana is as follows: *It is safe to replace component c with c' inside root transaction r, if the state of c' is not used in the future of r, or it is redefined prior to its usage.*

The first part of the intuition suggests that it is safe to adapt if the values of each and every state construct $s \in \texttt{StC}(r, c')$ is not used in the future of r. In other words, no use_s happens after adaptation and during the remaining execution of root transaction. The second part of the intuition implies that it is also safe to adapt if the value of state construct s is redefined before its next usage in r. We know that any change in the state of s is modeled as def_s. Therefore, we can express the second part of the intuition as follows: If a use_s happens after adaptation, there should be a def_s that occurs after adaptation and precedes use_s. For instance, in Figure 4(c), intra-component update criterion holds for *latitude* at update interval ⓑ, as the only instance of $use_{latitude}$ that happens after ⓑ is at $t_7 = \texttt{DeT}(RE, Map, 3)$, but *latitude* is redefined prior to that at $t_6 = \texttt{DeT}(RE, Map, 2)$. The intra-component update criterion also holds for *timeToTarget* and *longitude* at ⓑ. Therefore, we can say ⓑ is an intra-component update interval, since no other state construct of the *Map* is used after ⓑ, without first being redefined.

The intra-component update criterion is defined formally as follows:

Definition 3 (*Intra-Component Update Interval*). In the context of root transaction $r \in R$, a component $c \in C$ is in an intra-component update interval with respect to its replacement component $c' \in C$ after transaction $t_i = \texttt{DeT}(r, c, i)$, if

$\forall j : (i < j, s \in \texttt{StC}(t_j, c'), use_s \in \texttt{DuG}(t_j, c'), \exists k : (i < k < j, def_s \in \texttt{DuG}(t_k, c')))$,
where $t_j = \texttt{DeT}(r, c', j)$ and $t_k = \texttt{DeT}(r, c', k)$.

THEOREM 1. *Replacing a component in an intra-component update interval does not cause inconsistency.*

PROOF. The proof of Theorem 1 is by contradiction: Let us assume updating component $c \in C$ with component $c' \in C$ under the update interval of Definition 3 causes inconsistency. Based on Definition 1, this means that there is at least one state construct s' in component c' that at the time of use has a different state from the corresponding construct s in component c. Following the guidelines of Definition 3 in Theorem 1, every state construct of c' that is *used* in future transactions (i.e., $\texttt{StC}(t_j, c')$) are redefined before usage. Therefore, we can redefine state construct s' to be equal to s. This results in a contradiction. and. hence, our assumption is false. □

We can now define Savasana's criterion for safe adaptation:

Definition 4 (*Savasana*). Component $c \in C$ is in a Savasana interval in the context of root transaction $r \in R$ if it is in either an inter-component update interval or an intra-component update interval.

COROLLARY 1. *Replacing a component in Savasana interval does not cause replacement inconsistency.*

PROOF. The proof of Corollary 1 trivially follows from Lemma 1 and Theorem 1. □

Consistency Model for a component $c' \in C$ is the set of all intervals in which it can safely replace an old component $c \in C$ (i.e., c satisfying the Savasana criteria). For instance, the Consistency Model of the *Map* component in the context of transaction shown in Figure 4 is the set {ⓐ, ⓑ, and ⓔ}, as those are the intervals in which the *Map* can be safely adapted.

The Savasana criteria, particularly the intra-component update interval, is conceptually close to the definition of the *live* variable in the compiler domain. A variable is defined as live at a program point if *"its current value may be read during the remaining execution of the program"* [Aho et al. 1986]. Intuitively, at a program statement a variable is live when satisfying one of these two conditions: (1) it is read in the current statement or (2) it is read in future statements unless it is written in current statement. The application of *Liveness Analysis* in compiler optimization is to eliminate non-live variables in the compiled target code to reduce the number of required registers.

Despite this conceptual similarity, Savasana criteria and Liveness analysis have some key differences. Liveness analysis is defined in terms of low-level program structures, while Savasana criteria are defined in terms of high-level software components. More specifically, if a class is considered as a component, Savasana only deals with the class fields, while Liveness analysis takes all variables into account, including non-fields such as local variables of methods. Moreover, while Savasana criteria are defined with respect to the combination of all state constructs of a component, Liveness analysis considers and evaluates each variable independently.

4.3. Non-Determinism Issues

The impact of non-determinism has been largely ignored in the prior literature. This is attributed to the fact that the black-box approaches do not have access to the information necessary for reasoning about such issues. Since Savasana analyzes the code, it can determine the parallel execution flows that could be executed non-deterministically.

For the sake of simplicity, our formalism assumes deterministic transactions. In reality, however, $\texttt{DeT}(r)$, defined in Section 4.1, is a regular expression of transactions. When a *dependee* transaction is encountered inside a conditional statement, we would mark it as an optional transaction. For instance, if we assume that t_2 in Figure 3 is optional, then the regular expression for Resource Estimation would be as follows: $\texttt{DeT}(RE) = <t_1, t_2?, t_3, t_4, t_5, t_6, t_7, t_8>$.

A given update interval can be a part of many parallel execution flows. In Savasana, we adopt a pessimistic approach and require the update interval to be safe in all of those execution flows to guarantee consistency. In other words, we enforce Savasana's criteria on all possible execution flows that result from combining unique paths from the START vertex to the END vertex of all DUGs in all instances of $\texttt{DeT}(r, c')$ regular expression. Note that there is no parallelism or non-determinism inside a given execution flow. Since in Savasana we only care about define and use state constructs rather than their actual values, we treat loops as conditional statements as well.

Figure 5 depicts the DUG for the new version of the *Map* component (i.e., recall Listing 1(c)), which behaves non-deterministically. Here, instead of showing the DUG for all the members of $\texttt{DeT}(RE, Map)$, we show only the part, which differs from Figure 4 (i.e., $\texttt{DeT}(RE, Map, 3) = t_7$). In this new version, since the code has changed, the resulting Consistency Model has also changed. The update interval of ⓑ is not safe for adaptation anymore. As you may recall from Figure 4, $use_{timeToTarget}$ happens at $\texttt{DeT}(RE, Map, 4) = t_8$, and, hence, all the execution flows should have at least one $def_{timeToTarget}$ that happens after ⓑ but before the beginning of t_8. However, we cannot find such a definition in the execution flows that take the path that jumps over $def_{timeToTarget}$ in t_7 (see Figure 5(c)).

Static program analysis is subject to over-approximate the behavior of a software system, and therefore our approach is subject to have false positives. However, by taking a conservative approach in our analysis, we eliminate false negatives to guarantee safe adaptation, as detailed in the next section. Applying a non-deterministic version of Savasana increases the false positives and makes the adaptation process more

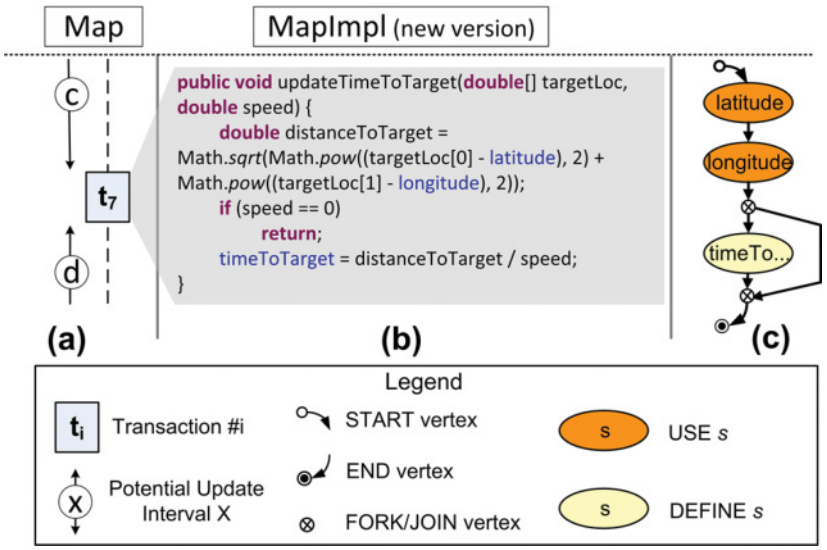

Fig. 5. DUG of transaction $\texttt{DeT}(RE, Map, 3) = t_7$, where the *Map*'s implementation of Listing 1(b) has been replaced by the *Map*'s implementation of Listing 1(c).

disruptive, that is, changes are not effected out of concern for situations that do not manifest at runtime. However, in its worst case, when the intra-component criterion is not satisfied for any component, Savasana performs as efficient as Tranquility.

5. IMPLEMENTATION

This section describes our realization of *Savasana* for Java programs running on top of the Spring framework.

5.1. Overview

As depicted in Figure 6, our implementation consists of two parts: *Code Analysis* that runs on the system's implementation logic and *Runtime Control* that manages the corresponding *running system*. Code Analysis runs once for each system (re)configuration and provides the required input for Runtime Control, which operates continuously during the execution of the software system.

Code Analysis comprises two activities: the *Dependency Extractor* and *Model Analyzer*. Dependency Extractor employs static analysis techniques to extract a *Dependency Model* of the components from either the system's source code or its executable code (e.g., Java bytecode). The Dependency Model consists of two parts: the inter-component and intra-component dependency models. The inter-component dependency model, analogously to the UML sequence diagram, represents the relationships among the interacting components in a transaction. The intra-component dependency model represents the relationships among the internal constructs (variables and data structures) constituting a software component's state, as well as the operations that manipulate the state. Model Analyzer uses both dependency models to determine the safe update intervals for all root transactions and represents this information in a *Consistency Model*.

Runtime Control also consists of two activities: *Adapter* and *Monitor*. Adapter receives a *request for adaptation* and waits until the *Adaptation Registry* corresponding to the target component indicates that it is safe to adapt. At that point, it swaps the old component with the new one. Monitor observes the transactions in the running system,

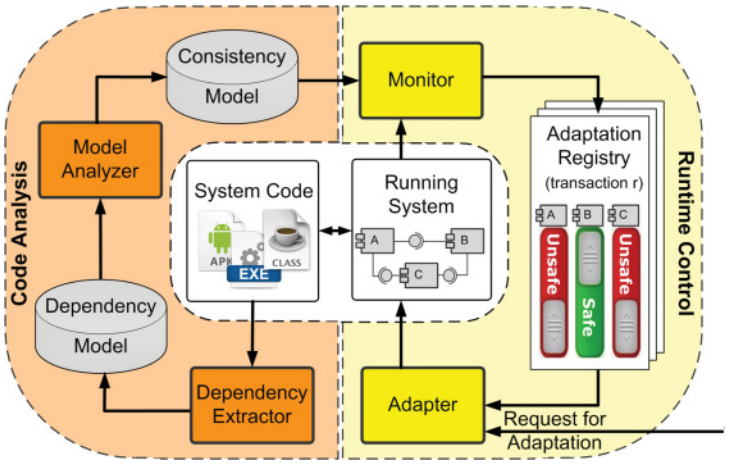

Fig. 6. Overview of our implementation.

matches them to the Consistency Model to determine which components are safe to adapt, and continuously updates the status of components in the Adaptation Registry. A component that is marked "safe" in the Adaptation Registry can be adapted without creating inconsistency problems.

In the following two sections, we describe the details of Code Analysis and Runtime Control.

5.2. Code Analysis

We realized Savasana's code analysis capabilities on top of Soot [Vallée-Rai et al. 1999]. Soot is a Java optimization framework that can also be used for static analysis of either source or executable Java code.

Code Analysis consists of two activities: Dependency Extractor and Model Analyzer. Both of these activities operate on CFG, an abstract representation of the code, where vertexes represent statements (including implicit statements such as START and END) and edges represent flow of control in the code. We denote the set of vertexes and edges in a CFG as V and E, respectively. The details of extracting CFG is beyond the scope of this article. We rely on existing work [Vallée-Rai et al. 1999] to build this representation. We use Java source code for the sake of readability in all of our examples. However, our implemantation can also be applied to executable Java code, since Soot can be used to extract the CFG from Java bytecode [Vallée-Rai et al. 1999].

Dependency Model is composed of two parts: inter- and intra-component dependency models. An inter-component dependency model represents the sequence of transactions comprising a root transaction in the system (we call it D_r, which is a sequence of transactions returned by function $\texttt{DeT}(r), r \in R$), while an intra-component dependency model represents the relationship between operations and data structures inside the component. In the following subsections, we describe how Savasana extracts these models in more detail.

5.2.1. Inter-Component Dependency Model. An inter-component dependency model is obtained by traversing CFG_t, denoting a subset of CFG relevant to transaction t. In essence, CFG_t represents an execution path in the overall CFG of one or more components. Figure 7 shows a hypothetical CFG_t for a subset of the RE scenario depicted in Figure 3. The bold vertexes and edges represent the execution path corresponding to

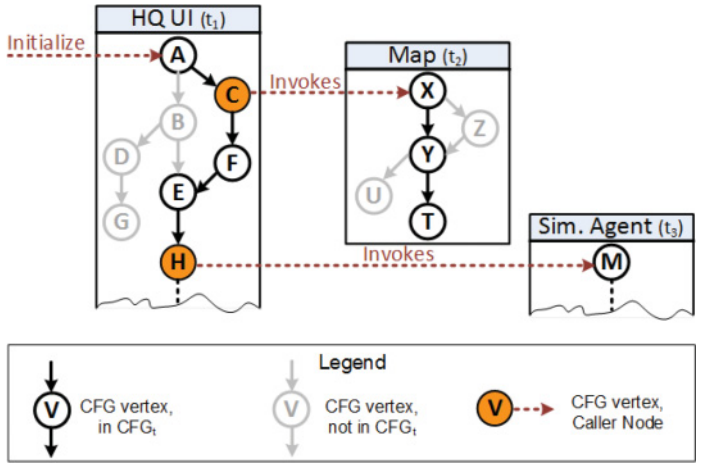

Fig. 7. Hypothetical CFG_t for a subset of the RE scenario of Figure 3, including t_2 and a part of t_1 and t_3.

components' participation in the RE scenario. Filled vertexes (C and H) show the points where a transaction starts another (dependent) transaction by invoking another component's interface. For instance, in Figure 7, the component HQ UI, which is running the root transaction t_1, invokes an interface of Map at vertex C and starts transaction t_2.

To extract the inter-component dependency model for any given root transaction $r \in R$, we start from the entry point (i.e., interface) of the host component of r. Recall from Section 5.1 that an underlying assumption in our research as well as prior work is that the interfaces responsible for initiating the root transactions are known. We traverse the CFG_t of r and its dependent transactions in a *depth first search* manner. This means that when we reach a caller node (filled vertexes in Figure 7), we leave the current transaction graph and start traversing the first node of the callee transaction. On the other hand, when we reach the end of a callee transaction, we backtrack to the next node in the caller graph. For instance, in Figure 7, $<\text{START}\rightarrow\text{A}\rightarrow\text{C}\rightarrow\text{X}\rightarrow\text{Y}\rightarrow\text{T}\rightarrow\text{F}\rightarrow\text{E}\rightarrow\text{H}\rightarrow\text{M}\rightarrow\cdots>$ is the sequence of traversed nodes. This process is repeated until we reach the end of CFG_t for root transaction.

During the CFG_t traversal, the transaction sequence D_r is constructed. At the beginning of the process, D_r is initialized by adding the root transaction. Subsequently, on reaching any invocation node, we append the callee transaction to D_r. For example, in the following sequence, the start and end of transactions t_1, t_2, and t_3 are identified and appended to D_r as follows: $<\text{START}\xrightarrow{t_1.s}A\rightarrow\text{C}\xrightarrow{t_2.s}\text{X}\rightarrow\text{Y}\rightarrow\text{T}\xrightarrow{t_2.e}\text{F}\rightarrow\text{E}\rightarrow\text{H}\xrightarrow{t_3.s}\text{M}\rightarrow\cdots>$. Note that Figure 7 shows a part of CFG_t for the RE scenario. The final extracted inter-component dependency model (i.e., D_r) following this process would be the same as that shown in Figure 3.

5.2.2. Intra-Component Dependency Model. For modeling the internal behavior of a component $c \in C$ in a given root transaction $r \in R$, we first extract the sequence of transactions involving c in the context of r, or $\texttt{DeT}(r, c)$ as defined in Section 4.1. Afterwards, we leverage Definition-Use Chain analysis [Harrold and Soffa 1994] to generate Define Use Graph ($\texttt{DuG}(t, c)$) of each transaction t. In our analysis, distinction between def_s or use_s of a state construct $s \in S$ is made based on whether it appears on the left-hand side (LHS) or right-hand side (RHS) of an assignment, respectively.

The inter- and intra-component dependency models extracted in the manner described above become inputs to the Model Analyzer component of our implementation

(see Figure 6). This component is a Java program that implements the Savasana criteria, as described in detail in Section 4. The output from Model Analyzer is the Consistency Model, representing the set of intervals that are safe for adaptation of each component under each scenario.

5.3. Runtime Control

We implemented Savasana's runtime control capabilities on top of the Spring framework [Johnson et al. 2005]. For implementing the Monitor component, we used Spring's extensive support for aspect-oriented programming, and for the Adapter component, we relied on Spring's dynamic (re)configuration facilities.

Runtime Control uses the models derived through Code Analysis to effect and manage the adaptation of software, while guaranteeing the consistency. As you may recall from Figure 6, Runtime Control has two activities: Monitor and Adapter.

Pseudocodes 1 and 2 summarize the activities performed by Monitor and Adapter, respectively. Monitor is called around (i.e., right before/after) the execution of each transaction to keep the state of system updated. On the other hand, Adapter runs continuously, via Adapter thread, and performs the adaptation process. The details of these functions are described in the remainder of this section.

PSEUDOCODE 1: Monitor (implemented using Spring's support for aspect-oriented programming)

```
   /* ConsistencyModel is the knowledge-base shown in Figure 6, which can be
      queried regarding to the (safe/unsafe) state of transactions.            */
   /* registry is the ''Adaptation Registry'' shown in Figure 6 and implemented
      as a map. Its key is the pair of < component_id, thread_id >, and its value is 'SAFE'
      or 'UNSAFE'.                                                             */
 1 Function Monitor.Run(transaction)
     // This function is called before/after the execution of transactions
 2   loggedTrace ← Monitor.loggedTrace;
 3   component ← transaction.hostComponent;
 4   thread ← transaction.thread;
 5   if ConsistencyModel.isSafe(transaction, loggedTrace) then
 6     registry.put(< component_id, thread_id > , 'SAFE');
 7   else
 8     registry.put(< component_id, thread_id > , 'UNSAFE');
 9   Monitor.LogTrace(transaction);
10 Function Monitor.ResetRegistry(component)
     // Reset the state of the component in the registry to 'UNSAFE'
11 Function Monitor.LogTrace(transaction)
     // Add the transaction to the system execution log (i.e. Monitor.loggedTrace)
```

5.3.1. Monitor. Monitor observes the execution of all transactions running in the system and detects if any component is in a safe update interval for replacement. To that end, for each transaction and its host component, Monitor retrieves the state of the transaction (line 2 of Pseudocode 1), which has been previously logged (line 9) and matches it against the Consistency Model of the component (lines 3–5).

Based on this analysis, Monitor records the status of components in the Adaptation Registry (lines 5–8). Safe adaptation status (which is depicted as a green switch in Figure 6) for a component in the context of a particular transaction means that the

PSEUDOCODE 2: Adapter (implemented using Spring's support for dynamic (re)configuration)

```
   /* toBeUpdatedComponents is a set that includes all components associated with
      an update request.                                                          */
   /* rootTransactions is a set that includes all root transactions running in
      the system.                                                                 */
   /* ConsistencyModel is the knowledge-base shown in Figure 6, which can be
      queried regarding the (safe/unsafe) state of transactions.                  */
   /* registry is the ''Adaptation Registry'' shown in Figure 6 and implemented
      as a map. Its key is the pair of < component_id, thread_id >, and its value is 'SAFE'
      or 'UNSAFE'.                                                                */
 1 Function Adapter.Run()
       // This function is called continuously via Adapter thread
 2     foreach component ∈ toBeUpdatedComponents do
 3         SafeToUpdate ← 'TRUE';
 4         foreach transaction ∈ rootTransactions do
 5             thread ← transaction.thread;
 6             if registry.get(< component_id, thread_id >) = 'SAFE' then
 7                 Adapter.Suspend(thread);
 8             else
 9                 SafeToUpdate ← 'FALSE';
10         if SafeToUpdate then
11             Adapter.Update(component);
12             Adapter.ResumeAll();
13             Monitor.ResetRegistry(component);
14 Function Adapter.Suspend(thread)
       // Suspend the thread
15 Function Adapter.ResumeAll()
       // Resume all suspended transactions
16 Function Adapter.Update(component)
       // Update the component
```

component satisfies the Savasana criteria at that point in time. Note that our implementation treats the concurrently running root transactions independently of each other.

5.3.2. Adapter. Adapter is responsible for two main tasks: (a) effecting changes in the running software through the `Adapter.Run` function and (b) suspending root transactions through the `Adapter.Suspend` function. These tasks are initiated when the Adapter receives a request for replacement of a component. After receiving the request, Adapter waits until the corresponding component registry in the Adaptation Registry switches to the safe mode for all active root transactions involving the component (i.e., when the `SafeToUpdate` flag defined in line 3 of Pseudocode 2 remains *True*) and then updates the component (lines 10–13).

When a component participates in multiple concurrently running root transactions, it is possible that the Adapter may have to wait for a long time before a safe interval is reached with respect to all root transactions if it is ever reached. To address this issue, the Adapter takes a proactive approach to steer the system toward a safe interval. Once the Monitor reports that a component is in a safe interval in the context of a particular root transaction, Adapter suspends that root transaction (`Adapter.Suspend` function called in line 7 of Pseudocode 2). Suspending a root transaction means temporarily

stopping the corresponding thread of execution. By gradually suspending the root transactions in which a component is involved, we are assured to eventually reach a safe interval with respect to all active root transactions, at which point the Adapter replaces the component with a new version of it. Finally, after adaptation is finished, Adapter resumes all the suspended root transactions (`Adapter.ResumeAll` function called in line 12).

Note that, in contrast to Tranquility, which takes a passive approach to adaptation and waits until a safe state is reached, Savasana takes an active approach and forces the safe state. This significantly decreases the wait time between receiving an adaptation request and fulfilling it. In fact, while Tranquility cannot provide any guarantees regarding reachability of safe state, Savasana guarantees the safe state will eventually be reached and, as shown in the next section, in a fraction of the time it takes to reach the safe state in Tranquility. In addition, Tranquility (as well as other existing approaches) locks the software components, such that they would not respond to any other request for services, making the approach susceptible to deadlock problems. But Savasana takes an active role without causing deadlocks, as it suspends the root transactions in the safe interval, instead of locking the components. Indeed, this unique ability comes from the knowledge obtained through extracting a detailed model of the components' internal behavior.

5.4. Assumptions and Limitations

As shown in Section 4.2, satisfying Savasana criteria guarantees safe and consistent updates of system components. Although from a conceptual standpoint Savasana is applicable to all kinds of component-based software, there are a few assumptions in the implementation described in this section that may limit its application.

Our implementation assumes the following:

(1) Components of the system are known, meaning that we know which code files belong to each component.
(2) Two versions of the component being updated provide the same interchangeable functionality accessible through the same interfaces.
(3) Component interfaces responsible for initiating root transactions are known; for example, in reference to Figure 3, we know the interface provided by *HQ UI* that is responsible for the initiation of the root transaction depicted there.
(4) Components interact and initiate new transactions synchronously by invoking each other's interfaces (e.g., public methods). These assumptions are consistent with those made in the prior literature [Kramer and Magee 1990; Vandewoude et al. 2007].
(5) Components of the system and the adaptation agent are running on the same machine. Supporting distributed settings requires a more advanced tool support to safely update components that are deployed remotely. Most notably, Runtime Control components would need to keep replicas of Adaptation Registry consistent across a set of distributed nodes.

The aforementioned assumptions limit the applicability of our current prototype implementation on top of the Spring framework. However, as alluded to earlier, they do not affect the conceptual contribution of Savasana as a sufficient criteria for safe adaptation of component-based software.

6. EVALUATION

We examine Savasana's behavior by measuring two properties: (1) *reachability*, the amount of time it takes for fulfilling the adaptation request, and (2) *disruption*, the amount of time that application threads are suspended for adaptation. There are two

factors that impact these properties: (1) the internal structure of root transactions (i.e., scenarios) and (2) the level of concurrency in the system.

In this section, we present the result of analyzing the sensitivity of Savasana to these factors on top of EDS [Malek et al. 2005; Malek 2007]. As you may recall from Section 2, EDS is intended for the deployment and management of personnel in emergency response scenarios. This software system consists of 13 component types, providing more than 50 interface types that are realized using over 5.6 KLOC. Additionally, the components utilize shared code and utility classes that bring the total size of the system to over 10.2 KLOC. Table II provides more detailed information about the components and interfaces of EDS. In Section 2, we described the system's architecture as well as its key use-cases.

In addition, we conduct an additional set of experiments on a suite of benchmark systems, where we control the various parameters (e.g., inter- and intra-component properties) that may affect Savasana's behavior. As Savasana is an improvement over Tranquility, we compare our results against Tranquility.

We have executed our experiments on a MacBook Pro laptop with a 2.66GHz Intel Core 2 Duo processor and 8GB 1067MHz DDR3 memory. We have handled uncontrollable factors in our experiments by repeating the experiments 33 times and reporting the results using their 95% confidence intervals, unless otherwise noted.

The interested reader can access our implementation and evaluation artifacts at Savasana's homepage.[3]

6.1. Effect of Scenarios

As you may recall from Section 4, the structure of a root transaction directly impacts the Consistency Model of components that participate in that transaction. Therefore, the structure of the root transaction impacts reachability, as Savasana uses Consistency Models to determine the safe update intervals. We evaluate the sensitivity of Savasana in three scenarios corresponding to EDS's three main functionalities, which, as you may recall from Section 2, are as follows: SA, DA, and RE. Note that the RE scenario depicted in Figure 3 is only for illustration purposes, as the original scenario is more complicated. Since we only care about the effect of scenarios in these experiments, we fix the level of concurrency to two users (concurrently running root transactions) in the system.

The root transactions corresponding to these scenarios differ from each other in two ways: (1) number of transactions and (2) number of safe update intervals for components. We selected *Weather*, *Repository*, and *Map* as the candidate components that are updated during the execution of root transactions corresponding to the SA, DA, and RE scenarios, respectively. For each scenario, we conducted the same experiment for both Tranquility and Savasana. During these experiments, we interrupted the normal execution of the running system by sending a request for adaptation of the target component and measured reachability and disruption.

The results of our experiments are provided in Table III. As we can see, reachability of Tranquility is proportional to the number of transactions involved in the scenario (i.e., $|D_r|$). As the number of transactions in the scenarios increases, the wait time for updating the component involved in these transactions increases as well. Tranquility waits until all transactions discontinue using the target component before adapting it. Therefore, as $|D_r|$ increases, the wait time for reaching the tranquil state increases. In other words, Tranquility is not able to make use of all possible safe update intervals, as it only considers two safe update intervals: before the first usage and after the last usage.

[3]http://www.ics.uci.edu/∼seal/projects/savasana/.

Table II. EDS Components and Their Provided/Required Interfaces

		EDS Components												
		Clock	Deployment Advisor	HQ UI	Map	Repository	Resource Manager	Resource Monitor	SAKB UI	Simulation Agent	Strategy Analysis KB	Strategy Analyzer	Weather	Weather Analyzer
Interfaces (R=Required , P=Provided)	AddResources						P	R						
	AdviseDeployment		P	R										
	AdviseDeploymentResp		R	P										
	AnalyzeStrategy			R								P		
	AnalyzeStrategyInvalidResp			P								R		
	AnalyzeStrategyValidResp			P								R		
	CallUIUpdate			P									R	
	DeleteResources						P	R						
	DeleteRule										P			
	DepleteResources						P	R						
	DontSaveFight			R						P				
	DontSaveFightResp			P						R				
	FetchResources							P						
	FetchResourcesResp							R						
	GetAllData					P								
	GetAllDataResp				P	R								
	GetAttributes				P		R							
	GetAttributesResp				R		P							
	GetData				R	P								
	GetDataResp				P	R								
	GetInitialRules										R			
	GetInitialRulesResp					R					P			
	GetMap			R	P			P						
	GetMapRegions				P		R							
	GetMapRegionsResp				R		P							
	GetMapResp			P	R									
	GetResources						P	R						
	GetResourcesResp						R	P						
	GetRules		R							R	P	R		
	GetRulesResp		P							P	R	P		
	GetWeather												P	R
	GetWeatherLevel													P
	GetWeatherLevelResp													R
	GetWeatherList				R								P	
	GetWeatherListResp				P								R	
	GetWeatherResp				P								R	P
	LogState										R			
	MoveResources			R			P	R						
	SetAttributes			R	P	P	R							
	SetAttributesResp				P,R	R								
	SetMap				P	P	R			R				
	ShowSAKBUI			R					P					
	SimulateFight			R						P				
	SimulateFightResp			P						R				
	SimulateFightRespFinal			P						R				
	StoreRule										P			
	SupplyResources						P	R						
	Tick	R						P		P	P		P	
	UpdateResources			R				P						
	UpdateRule								R		P			
	UpdateUI			P	R									
Lines of Code		45	270	1644	397	536	645	322	237	299	312	309	430	183

Reachability of Savasana is also affected by $|D_r|$ but not as significantly as Tranquility. The reason is that the reachability of Savasana is also affected by the number of safe update intervals that occur during the execution of root transactions. Savasana allows adapting a component when all root transactions in the system are in the Savasana interval (recall Definition 4) with respect to the new version of the component. More safe update intervals means less wait time for adaptation and, in turn, faster reachability.

Table III. Impact of Three EDS Scenarios on Reachability and Disruption

		EDS Scenario		
		Strategy Analysis	Resource Deployment	Resource Estimation
Number of D_r's Transactions		5	6	8
Number of Safe Update Intervals		4	3	5
Reachability (sec)	Tranquility	24.68 ± 8.4	32.78 ± 13.3	66.32 ± 14.7
	Savasana	6.53 ± 2.31	7.42 ± 1.34	7.92 ± 1.62
Disruption (sec)	Tranquility	0.07 ± 0.02	0.05 ± 0.01	0.07 ± 0.02
	Savasana	2.26 ± 1.05	2.23 ± 0.72	4.01 ± 1.32

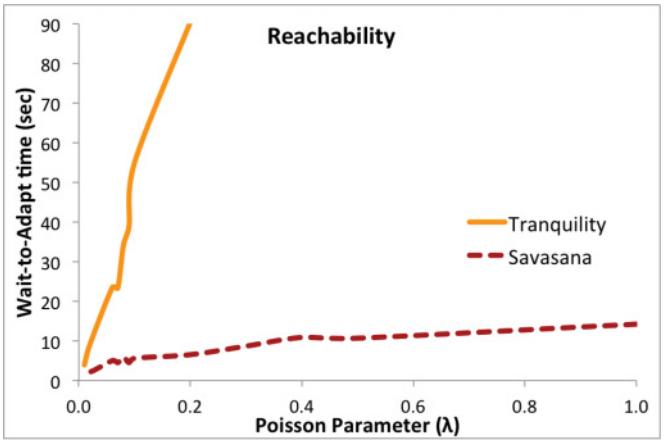

Fig. 8. Reachability of Tranquility vs. Savasana.

Unlike reachability, according to the results of Table III, there is no clear relationship between disruption and the properties of scenarios for Tranquility or for Savasana. As we will show later, disruption is instead dependent on the level of concurrency parameter, which was fixed in this part of experiment. However, in all scenarios, disruption caused by Tranquility is less than that by Savasana. This is because Tranquility is a passive approach that does not suspend the threads, while Savasana is an active approach that suspends the threads in their safe intervals.

6.2. Effect of Concurrency

Monitor component tracks the active root transactions and keeps the status of components up to date in the Adaptation Registry. More concurrency means more root transactions in the system and, in turn, more deliberation in the activities inside Runtime Control. We designed an experiment to measure the sensitivity of system properties (i.e., reachability and disruption) to concurrency. In this experiment, we simulated 400 EDS users (which resembles a wide rescue operation in EDS) initiating concurrent root transactions according to a Poisson distribution [Bertsekas and Tsitsiklis 2008]. We controlled the level of concurrency by changing the value of the Poisson parameter (i.e., λ) from 0.01 to 1. Since in this experiment we only cared about the effect of concurrency and we wanted the results to be comparable, we only invoked the RE functionality.

Figure 8 plots the reachability of safe update interval for Savasana and Tranquility in this experiment. As we can see, Savasana always reaches a safe update interval much faster than Tranquility. Moreover, by taking a look at the slope of the lines, we can see that the growth rate of Savasana is much slower than that of Tranquility. In fact, in this experiment, Tranquility was not able to reach a safe update interval for

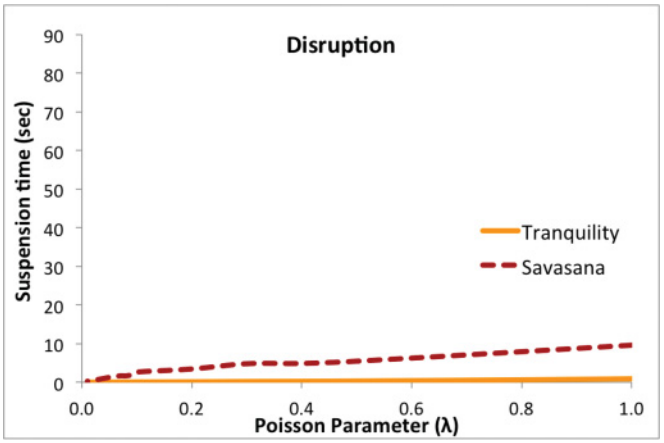

Fig. 9. Disruption of Tranquility vs. Savasana.

$\lambda > 0.3$, which accounts for six active users in the system. This large difference is due to the fact that Savasana takes an active role in adaptation and gradually enforces Savasana criteria, while Tranquility takes a passive role and waits for the Tranquility criteria to manifest itself. As the level of concurrency increases, the chance of reaching a safe update interval drops, and, hence, the tranquil interval becomes unreachable.

There is, however, a slight side effect in taking an active role in adaptation. Figure 9 shows the disruption caused by Savasana and Tranquility in the same set of experiments. Since Tranquility is a passive approach and does not suspend the threads, it has a negligible disruption caused by buffering the events intended for the new component during the transition. Therefore, as the level of concurrency increases, Tranquility's disruption slightly increases. On the other hand, Savasana gradually suspends all of the required root transactions in a safe update interval. Therefore, as the number of concurrent root transactions for the same set of components increases, the chance of more root transactions being suspended increases and the normal operation of the system is further disrupted. However, as we see in Figure 9, this impact is sublinear (e.g., with 27 concurrent root transactions in the system, the total suspension time of root transactions is less than 9s on average).

By looking at Figures 8 and 9 side by side, we conclude that Savasana achieves a better overall tradeoff in comparison to Tranquility. It creates a little more disruption in the normal operation of the system and, in turn, attains a large gain in the time it takes to reach a safe update interval.

6.3. Performance and Timing

We have evaluated the efficiency of both Runtime Control and Code Analysis phases of Savasana. We exercised two versions of EDS under the same scenarios, with and without Savasana, and did not trigger any adaptation in either case, to be able to measure the impact of Savasana's monitoring and analysis components on the system's performance. We conducted these experiments 33 times to be able to report the results with a 95% confidence. As we can see in Table IV, the overhead of Runtime Control is negligible in comparison to the normal activity of the system.

We also measured the execution time of the Code Analysis phase, as provided in Table IV. Unlike Runtime Control, which runs continuously, Code Analysis runs once per each system (re)configuration. Therefore, the execution time of Code Analysis is not as crucial as Runtime Control, as the Consistency Models for a newly adapted

Table IV. Overhead of Runtime Control and Code Analysis Time

	EDS Scenario		
	Strategy Analysis	Resource Deployment	Resource Estimation
Execution Time without Savasana (sec)	14.16 ± 1.03	12.45 ± 1.01	18.23 ± 0.98
Runtime Control Overhead (%)	$0.07\% \pm 0.03\%$	$0.15\% \pm 0.03\%$	$0.10\% \pm 0.05\%$
Code Analysis Time (sec)	2.17	2.09	2.38

software are not needed until the next adaptation decision, which in most cases should leave plenty of time for the analysis to complete. If a software system goes through rapid adaptations, then the Consistency Models for different possible configurations of the software would have to be derived ahead of time. Since there is only negligible variability in the execution time of Code Analysis, we do not report it with a confidence interval.

6.4. Sensitivity Analysis through Benchmarks

To control the characteristics of the subjects (i.e., systems under test), we use a set of benchmark systems. These systems are handcrafted to help us investigate the efficacy of applying Savasana for adaptation of software systems with varying characteristics. Each benchmark system is designed to include *one* adaptable component c in a root transaction r. We define a set of variability dimensions (VD_1–VD_6) to control the characteristics of each benchmark system. We measure the impact of each dimension on Savasana. These variability dimensions are categorized as either inter-component or intra-component.

Inter-component dimensions control external properties related to a component's interactions:

—**VD_1:** Number of dependent transactions in the root transaction r (`|DeT(r)|`).
—**VD_2:** Number of dependent transactions in the root transaction r and involving the component c (`|DeT(r,c)|`).

Intra-component dimensions control the internal structure of a component:

—**VD_3:** Total number of state constructs belonging to c that are affected by transaction r (`|StC(r,c)|`).
—**VD_4:** Number of state constructs defined in c in the context of transaction r, normalized by total number of state constructs of c ($|def_s|$/`|StC(r,c)|`, where $s \in$ `StC(r,c)`).
—**VD_5:** Number of state constructs used in c in the context of transaction r, normalized by total number of state constructs of c ($|use_s|$/`|StC(r,c)|` where $s \in$ `StC(r,c)`).
—**VD_6:** Average *cyclomatic complexity* [McCabe 1976] of a component.[4]

For a given component, VD_3–VD_6 represent the internal structure of that component with respect to its state construct. VD_3 simply counts the total number of state constructs, VD_4 and VD_5 provide the metrics for the definition and usage of those state constructs, and, finally, VD_6 indicates how they are defined or used.

To investigate the effect of aforementioned dimensions on the efficiency of Savasana, five benchmark systems are devised to represent different values for each dimension. Table V summarizes the values of each dimension for the benchmark systems. For each benchmark, we repeated the same experiment that interrupts the normal execution of the running system by sending a request for adaptation of the target component and then measured reachability of Savasana.

Our experiment results show that Savasana's reachability is less sensitive to inter-component dimensions compared to intra-component dimensions. To observe

[4]Cyclomatic complexity calculates number of linearly independent paths through a program.

Table V. Impact of System Variability Dimensions on Reachability of Savasana

	Number of Components	Inter-Component		Intra-Component				Reachability (sec)
		VD_1	VD_2	VD_3	VD_4	VD_5	VD_6	
system 1	5	10	6	2	3.00	1.00	1.16	2.80
system 2	5	5	3	2	3.00	1.00	1.16	2.58
system 3	5	5	3	6	1.00	1.83	1.16	8.28
system 4	5	13	6	2	3.50	3.00	1.20	6.23
system 5	5	13	6	2	3.50	3.00	3.20	17.43

this difference, consider the results of systems 1 and 2 in Table V. The reachability times of these two systems are quite close (2.80s vs. 2.58s), despite the considerable difference in their inter-component dimensions (10 and 6 vs. 5 and 3). On the other hand, consider systems 2 and 3, where the inter-component VDs are the same (5 and 3), but the intra-component dimensions (VD_3–VD_6) vary, which leads to significantly different reachability results. The reason is that Savasana takes internal structures of components into account to identify safe-to-update intervals.

To observe the effect of system's internal structure on Savasana's efficiency more thoroughly, once again consider the results of systems 2 and 3 shown in Table V. According to the experiment results, Savasana is more efficient in applying adaptation for system 2, because of the internal structure of system's components, namely the intra-component dimensions VD_3–VD_5. System 3 has six state constructs, with low value for VD_4 (state construct definition rate) and relatively high value for VD_5 (state construct usage rate), while system 2 has only two state constructs with higher value for VD_4 and lower value for VD_5. As a result, the possibility of redefining state constructs prior to their usage is higher in system 2, which means more safe update intervals and shorter wait-for-update (reachability) times, when using Savasana.

Finally, to investigate the effect of non-determinism, we designed system 5, which is a replica of system 4 with added cyclomatic complexity. In this benchmark system, the def/use of state constructs are gated by `IF` conditions, thereby increasing the possible execution paths. Recall that if only one of those added execution paths violates the safe adaptation criteria for an update interval, Savasana conservatively considers that interval unsafe for all execution paths. As a result, in a system with high cyclomatic complexity, the chance of finding safe update intervals is reduced, and, consequently, the wait time for updating the component is increased, which can be observed by comparing the results from systems 4 and 5.

7. RELATED WORK

In Section 3, we described the most closely related approaches, namely Quiescence [Kramer and Magee 1990], Tranquility [Vandewoude et al. 2007], and Version-Consistency [Ma et al. 2011], including their relationship to this work. Here we focus on the other related literature.

Gupta and Jalote [1993] provided a restricted condition to ensure the validity of an update by allowing the old version of program to be replaced by the new one when idle. To make the approach more applicable, Gupta et al. [1996] provided a more general solution that a program can be updated when the state of the old version is reachable by the new version of the updated program. These approaches and Savasana both define the valid update based on the concept of states. However, their approach operates on a very low level definition of state that consists of elements such as position on stack or program counter, while our research targets the consistency of component-based adaptation.

Another approach that addresses the problem of consistent update in the domain of program analysis is the research by Neamtiu et al. [2008], which introduced the concept of Version-Consistency at the code level. In contrast to our work that a safe update is defined in the context of system components, they used a contextual effect system to provide a correctness condition for dynamic software update. In their approach, they assume there is a unique identifier for root transactions to guarantee a dependent transaction is served by either the old or new version of a component. This assumption limits the applicability of the approach only to the systems that provide such a unique identifier. This work was the basis for the subsequent research [Ma et al. 2011] at the component level, which is discussed in Section 3.

Among the other research addressing the problem of dynamic update at the *code level*, Hicks et al. [2001] and Hicks and Nettles [2005] proposed a general-purpose dynamic updating system that provides the patches containing the updated code together with the code needed to transition from the old version to the new. More recently, this approach is tailored for specific programming languages, such as C [Hayden et al. 2012] and Java [Pina and Hicks 2013]. In addition to applying software updates at different levels (code level vs. component level), this thrust of research focuses on *how* to implement the updates dynamically, while the goal of our approach, Savasana, is to find *when* is a safe time to update. Dynamic patch approach tackles the timing problem by requiring the program to be coded in a specific way from the outset, and, thus, wait-for-update time could be negligible. However, since the approach places certain restrictions on the structure of the program, it cannot be easily applied to legacy systems.

As a step toward specification-driven dynamic update of systems, Ghezzi et al. [2012] and Manna et al. [2013] presented a technique for determining correct updates with respect to the changes in the system specification. Unlike our work that considers the system's code for deriving safe update, their approach uses the system specification or environment properties as the basis to identify correct updates. However, both approaches use a controller to automatically apply the updates as soon as possible.

In our prior work [Canavera et al. 2012; Yuan et al. 2014; Esfahani et al. 2016], we used log of events collected from a system to construct its inter-component dependency model. To that end, we applied data mining techniques to infer a set of probabilistic rules representing the dynamic component dependencies among the software components of a system. Then we used these rules to predict safety of adaptation at any given interval. Savasana differs from this approach, as it uses static analysis rather than dynamic analysis. Moreover, Savasana employs both inter- and intra-component dependency models for its analysis.

Software architecture has been shown to provide an appropriate level of abstraction and generality to deal with the complexity of dynamically adapting software systems [Kramer and Magee 2007; Oreizy et al. 1998]. Gomaa and Hussein [2004] developed the notion of reconfiguration pattern, which is a repeatable sequence of steps for placing a software component in the Quiescence state. In our prior work [Esfahani and Malek 2010, 2012], we extended this concept to provide safe adaptation support on top of a middleware platform. Ramirez and Cheng [2010] also developed a set of adaptation-oriented design patterns for various adaptation tasks such as monitoring, decision making, and reconfiguration.

The other related research area to our work is static program analysis. While static program analysis originated from the compiler community for source code optimization, in the past decade it has found application in several software engineering activities [Binkley 2007], including security vulnerability detection [Pérez et al. 2011], fault localization [Agrawal et al. 1995], debugging [Gupta et al. 1997], and verification [Blanchet et al. 2003]. To the best of our knowledge, this article is the

first to apply program analysis techniques in the design and construction of adaptive systems, particularity to ensure their consistency.

8. CONCLUSION

We presented Savasana, the first white-box approach for reasoning about consistency of component-based adaptation. Savasana employs program analysis to automatically recover models that accurately reflect the dependencies in the software. It uses these models to identify more opportunities for consistent adaptation of software than what is possible with prior techniques. As a result, Savasana is able to enact the changes in the software faster than the previously published approach, namely Tranquility. We have formally proved that Savasana's criteria for adaptation is sufficient to avoid inconsistencies caused by replacing a component and empirically evaluated our implementation of Savasana on top of the Spring framework using a system with many users and under various conditions.

In our future research, we aim to extend our current implementation of Savasana to distributed settings, where the software components are executing on different computers that may be geographically apart. A distributed deployment of Savasana would need to overcome timeliness concerns due to communication delay and ambiguities that may arise due to lack of a shared clock (e.g., not knowing the exact ordering of running transactions in the system). There are several plausible architectures to address challenges posed by distribution, including either a centralized solution, in which a centralized instance of Savasana orchestrates the adaptation under certain assumptions (e.g., maximum network latency), or a decentralized solution, in which multiple instances of Savasana coordinate their decisions to effect changes in the system.

REFERENCES

Hiralal Agrawal, Joseph R. Horgan, Saul London, and W. Eric Wong. 1995. Fault localization using execution slices and dataflow tests. In *Proceedings of the Sixth International Symposium on Software Reliability Engineering (ISSRE'95)*. 143–151.

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers, Principles, Techniques*. Addison Wesley.

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

David Binkley. 2007. Source code analysis: A road map. In *Proceedings of the International Conference on Software Engineering (ISCE'07) and the Workshop on the Future of Software Engineering (FOSE'07)*. 104–119.

Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2003. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*. 196–207.

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 *Proceedings of the 20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (SIGSOFT'12)/(FSE'12)*. 18.

Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Di Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. 2009. Software engineering for self-adaptive systems: A research roadmap. In *Proceedings of the Software Engineering for Self-Adaptive Systems [Outcome of a Dagstuhl Seminar]*. 1–26.

Naeem Esfahani and Sam Malek. 2010. On the role of architectural styles in improving the adaptation support of middleware platforms. In *Proceedings of the Software Architecture, 4th European Conference (ECSA'10)*. 433–440.

Naeem Esfahani and Sam Malek. 2012. Utilizing architectural styles to enhance the adaptation support of middleware platforms. *Inf. Softw. Technol.* 54, 7 (2012), 786–801.

Naeem Esfahani, Eric Yuan, Kyle R. Canavera, and Sam Malek. 2016. Inferring software component interaction dependencies for adaptation support. *Trans. Auton. Adapt. Syst.* 10, 4 (2016), 26.

Carlo Ghezzi, Joel Greenyer, and Valerio Panzica La Manna. 2012. Synthesizing dynamically updating controllers from changes in scenario-based specifications. In *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'12)*. 145–154.

Hassan Gomaa and Mohamed Hussein. 2004. Software reconfiguration patterns for dynamic evolution of software architectures. In *Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture (WICSA'04)*. 79–88.

Deepak Gupta and Pankaj Jalote. 1993. On-line software version change using state transfer between processes. *Softw. Pract. Exper.* 23, 9 (1993), 949–964.

Deepak Gupta, Pankaj Jalote, and Gautam Barua. 1996. A formal framework for on-line software version change. *IEEE Trans. Software Eng.* 22, 2 (1996), 120–131.

Rajiv Gupta, Mary Lou Soffa, and John Howard. 1997. Hybrid slicing: Integrating dynamic information with static analysis. *ACM Trans. Softw. Eng. Methodol.* 6, 4 (1997), 370–397.

Mary Jean Harrold and Mary Lou Soffa. 1994. Efficient computation of interprocedural definition-use chains. *ACM Trans. Program. Lang. Syst.* 16, 2 (1994), 175–204.

Christopher M. Hayden, Edward K. Smith, Michail Denchev, Michael Hicks, and Jeffrey S. Foster. 2012. Kitsune: Efficient, general-purpose dynamic software updating for C. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'12)*. 249–264.

Michael W. Hicks, Jonathan T. Moore, and Scott Nettles. 2001. Dynamic software updating. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'01)*. 13–23.

Michael W. Hicks and Scott Nettles. 2005. Dynamic software updating. *ACM Trans. Program. Lang. Syst.* 27, 6 (2005), 1049–1096.

Rod Johnson, Juergen Hoeller, Alef Arendsen, Thomas Risberg, and Colin Sampaleanu. 2005. *Professional Java Development with the Spring Framework*. John Wiley & Sons.

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

Jeff Kramer and Jeff Magee. 2007. Self-managed systems: An architectural challenge. In *Proceedings of the International Conference on Software Engineering (ISCE'07) and the Workshop on the Future of Software Engineering (FOSE'07)*. 259–268.

Rogerio de Lemos, Holger Giese, Hausi A. Muller, Mary Shaw, Jesper Andersson, Luciano Baresi, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cikic, Ron Desmarais, Schahram Dustdar, Gregor Engels, Kurt Geihs, Karl M. Goeschka, Alessandra Gorla, Vincenzo Grassi, Poala Inverardi, Gabor Karsai, Jeff Kramer, Marin Litoiu, Antonia Lopes, Jeff Magee, Sam Malek, Serge Mankovskii, Raffaela Mirandola, John Mylopoulos, Oscar Nierstrasz, Mauro Pezze, Christian Prehofer, Wilhelm Schafer, Wilhelm Schlichting, Bradley Schmerl, Dennis B. Smith, Joao P. Sousa, Gabriel Tamura, Ladan Tahvildari, Norha M. Villegas, Thomas Vogel, Danny Weyns, Kenny Wong, and Jochen Wuttke. 2011. Software engineering for self-adpaptive systems: A second research roadmap. In *Software Engineering for Self-Adaptive Systems II*. Dagstuhl, Germany, 1–32.

Xiaoxing Ma, Luciano Baresi, Carlo Ghezzi, Valerio Panzica La Manna, and Jian Lu. 2011. Version-consistent dynamic reconfiguration of component-based distributed systems. In *Proceedings of the SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13)*. 245–255.

Sam Malek. 2007. *A User-Centric Approach for Improving a Distributed Software System's Deployment Architecture*. Ph.D. Dissertation. University of Southern California.

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

Valerio Panzica La Manna, Joel Greenyer, Carlo Ghezzi, and Christian Brenner. 2013. Formalizing correctness criteria of dynamic updates derived from specification changes. In *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'13)*. 63–72.

Thomas J. McCabe. 1976. A complexity measure. *IEEE Trans. Software Eng.* 2, 4 (1976), 308–320.

Iulian Neamtiu, Michael Hicks, Jeffrey S. Foster, and Polyvios Pratikakis. 2008. Contextual effects for version-consistent dynamic software updating and safe concurrent programming. In *ACM SIGPLAN Notices*, Vol. 43. ACM, 37–49.

Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. 1998. Architecture-based runtime software evolution. In *Proceedings of the 1998 International Conference on Software Engineering (ICSE'98)*. 177–186.

Pablo Martín Pérez, Joanna Filipiak, and José María Sierra. 2011. LAPSE+ static analysis security software: Vulnerabilities detection in java EE applications. In *Future Information Technology*. Springer, 148–156.

Luís Pina and Michael Hicks. 2013. Rubah: Efficient, general-purpose dynamic software updating for java. In *Proceedings of the 5th Workshop on Hot Topics in Software Upgrades (HotSWUp'13)*.

Andres J. Ramirez and Betty H. C. Cheng. 2010. Design patterns for developing dynamically adaptive systems. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'10)*. 49–58.

Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*. 13.

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

Eric Yuan, Naeem Esfahani, and Sam Malek. 2014. Automated mining of software component interactions for self-adaptation. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'14)*. 27–36.

Received March 2015; revised December 2016; accepted February 2017