The Journal of Systems and Software 86 (2013) 854–867

Contents lists available at SciVerse ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

AFChecker: Effective model checking for context-aware adaptive applications

Yepang Liu[a], Chang Xu[b,c,*], S.C. Cheung[a]

[a] Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Kowloon, Hong Kong, China
[b] State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China
[c] Department of Computer Science and Technology, Nanjing University, Nanjing, Jiangsu, China

ARTICLE INFO

Article history:
Received 12 July 2012
Received in revised form 16 October 2012
Accepted 17 November 2012
Available online 14 December 2012

Keywords:
Adaptation fault
Deterministic constraint
Probabilistic constraint
False positive
Fault ranking

ABSTRACT

Context-aware adaptive applications continually sense and adapt to their changing environments. A large body of such applications relies on user-configured adaptation rules to customize their behavior. We call them rule-based context-aware applications (or RBAs for short). Due to the complexity required for adequately modeling environmental dynamics, adaptation faults are common in these RBAs. One promising approach to detecting such faults is to build a state transition model for an RBA, and exhaustively explore the model's state space. However, it can suffer from numerous false positives. For example, 78.6% of 784 reported faults for one popular RBA – PhoneAdapter, turn out to be false in a real deployment. In this paper, we address this false positive problem by inferring a domain model and an environment model for an RBA. The two models capture the hidden features inside user-configured adaptation rules as well as the RBA's running environment. We formulate these features as deterministic constraints and probabilistic constraints to prune false positives and effectively prioritize remaining faults. Our experiments on two real RBAs report that this approach successfully removes 46.5% of false positives and ranks 86.2% of true positives to the top of the fault list.

1. Introduction

Recent advances in mobile technologies foster the increasing deployment of pervasive computing infrastructures (Weiser, 1999). Context-aware adaptive applications for handheld devices with built-in sensors are becoming increasingly popular. These applications, typically driven by user-configured rules (Locale, 2012; SweetDreams, 2012; Tasker, 2012), follow an event-condition-action computing paradigm (Dittrich et al., 1995) to continually sense and adapt to their changing environments. For example, Tasker (Tasker, 2012), a Google contest awardee (Google ADC 2 Winners, 2009), can help its users switch their Android phones to silent mode when they arrive at their offices, and switch the phones back to ring mode after they return home. To achieve this, the users need to configure the following two rules:

- *Rule 1*: Enable silent mode when a Bluetooth device "OfficePC" is discovered nearby (i.e., an office situation).
- *Rule 2*: Enable ring mode when the phone's GPS module reports its current location as "Home" (i.e., a home situation).

While such adaptive applications help integrate the physical and cyber worlds, they are vulnerable to abnormal adaptations caused by rule misconfigurations (Sama et al., 2008, 2010a) Consider a situation for the preceding Tasker application: a user takes his office PC home to continue his work. Tasker will then discover that "OfficePC at home" satisfies both of the above rules. If the two rules have the same priority, they would be non-deterministically selected for execution, which may result in an unexpected adaptation. Such rule misconfigurations are common since users are typically not experts, and cannot carefully perform rule validation each time the rule configurations are changed. This problem may become more serious when many rules are configured and they implicitly correlate to each other. In fact, Tasker's developers already confirmed the frequent occurrence of such abnormal adaptations and pointed out that these adaptations would significantly reduce an RBA's usability (Tasker Limitations, 2012).

To detect such adaptation faults from misconfigured rules, a state-of-the-art technique (Sama et al., 2010a) extracts an adaptation finite state machine (called A-FSM) from an RBA's adaptation rules, and exhaustively searches this A-FSM's state space to find potential abnormal adaptations. For each detected fault, this model checking technique generates a report (see Fig. 1 for an example)

* Corresponding author at: State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China. Tel.: +86 25 89680919; fax: +86 25 83593283.
E-mail addresses: andrewust@cse.ust.hk (Y. Liu), changxu@nju.edu.cn (C. Xu), scc@cse.ust.hk (S.C. Cheung).

0164-1212/$ – see front matter © 2012 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jss.2012.11.055

Fault Type	Non-deterministic Adaptation Fault
Triggering Situation	
GPS.Location = "Home" && *Bluetooth.DiscoveredDevices*.contains("OfficePC")	
Details	Rule 1 and Rule 2 can be simultaneously satisfied in the above situation.

Fig. 1. An example fault report.

for user inspection. While an exhaustive search guarantees the completeness (i.e., no false negatives), it may produce many false positives. Our experiments reveal that 78.6% of 784 faults reported for a benchmark RBA – PhoneAdapter[1] (PhoneAdapter, 2012) are actually false (i.e., they will never occur in reality). Without effective filtering or prioritization mechanisms, this sheer volume of false positives would certainly hinder fault inspection and fixing. In this paper, we address the false positive problem of the existing model checking technique without sacrificing its completeness property.

We observed that an RBA's adaptive behavior is determined by two important factors: (1) its internal adaptation logic, and (2) its external environmental dynamics. Overlooking any of them might incur imprecision in the adaptation fault detection. Therefore, we propose to properly model the two factors for achieving effective adaptation fault detection. Specifically, we model an RBA's internal adaptation logic as a domain model, and its external environmental dynamics as an environment model. The derivation of both models has not been systematically studied in the existing literature. In this work, we focus on how to automatically derive these two models and formulate them into certain constraints to mitigate the false positive problem. We note that deriving the domain and environment models is non-trivial because of the following confounding issues:

1. Users tend to implicitly encode their domain knowledge in an RBA's adaptation rules. For example, consider a rule "turn off the phone's ringer when GPS is switched on, and it reports the current location as 'Broadway Theater"'. This rule implicitly assumes that the GPS service must be switched on before any of its usage. Although it is common sense to human beings, automated discovery of such knowledge is essentially non-trivial. Despite the challenges, a useful domain model should capture such knowledge. This is because such knowledge can help decide the impossibility of certain situations (e.g., GPS is switched off, but it reports a valid location), and then filters out the faults that are assumed to occur under these situations.
2. An RBA's running environment is intrinsically probabilistic, and different users' behavioral patterns can differ greatly. Given a set of RBA adaptation rules, some users may suffer from adaptation faults, while others may not. Environment models thus vary with different users and are difficult to infer statically. For example, without an in-depth analysis of historical environmental data, existing techniques can hardly discover the likelihood of certain situations (e.g., how likely it is to find a user's home PC in his office on Saturdays). A useful environment model, however, should capture the likelihood or unlikelihood of such a situation, and use it to rank a fault report associated with this situation accordingly. Thus users can focus on more likely faults.

To tackle these challenges, we propose a hybrid approach, which combines static analysis and pattern-based data mining, to extracting domain and environment models. Our approach relies on the existing static model checking technique (Sama et al., 2010a) for adaptation fault detection. We systematically process the generated fault reports using both deterministic and probabilistic constraints formulated from our derived models. Deterministic constraints are used to filter out fault reports that are guaranteed to be false positives. Probabilistic constraints are used to prioritize the fault reports that remain after pruning false positives using deterministic constraints. The quality of this probability-based ranking may fluctuate because spurious constraints can be accidentally associated with high probabilities. To address this issue, we designed a ranking refinement mechanism based on information theories. After the ranking, users may choose to validate the top ranked faults during an inspection round. To further minimize manual inspection efforts, we leverage their validation feedback to dynamically re-rank those uninspected faults. This allows more true positives to be promoted to the top of the fault list while more false positives are relegated to the bottom in the next inspection round. By doing so, our approach complements the state-of-the-art technique by improving the precision of its fault detection. The contributions of this paper include:

- We propose a novel approach, which combines static analysis and pattern-based data mining, to automatically deriving domain and environment models for RBAs to improve the effectiveness of the existing adaptation fault detection technique.
- We show how to leverage the constraints formulated from the derived models to prune false positives and prioritize fault reports in a sound way. We design a two-phase ranking mechanism as well as a feedback-directed dynamic ranking strategy to enhance fault ranking quality.
- We implement a tool called AFChecker, and evaluated it using two typical RBAs: PhoneAdapter (PhoneAdapter, 2012) and Tasker (Tasker, 2012). The evaluation results show that our approach can significantly enhance the effectiveness of the existing adaptation fault detection technique.

The rest of this paper is organized as follows. Section 2 reviews the background of our work. Section 3 presents our model derivation algorithm and dynamic fault ranking strategy. Section 4 describes the implementation details of our AFChecker and discusses our experimental results on two representative RBAs. Section 5 reviews related work, and Section 6 concludes the paper.

2. Background

In this section, we first review the basics of RBAs, and formally define the key concepts of our work. We then introduce the finite-state model of RBAs and use a small yet realistic example to illustrate how to perform model checking to detect adaptation faults.

2.1. Rule-based context-aware applications

RBAs are well supported by existing pervasive infrastructures (Capra et al., 2003; Gu et al., 2004b; Julien and Roman, 2006; Omnidroid, 2012). Due to their capability to perform proper actions under specific environmental changes, RBAs are becoming increasingly popular among users who wish to automate some recurring tasks (i.e., put a phone offline after midnight). To achieve such automation, RBAs commonly let users configure adaptation rules through a GUI. An adaptation rule typically contains five fields: current state, predicate, new state, actions, and priority level (Sama et al., 2010b). The semantics of a rule is that if the environmental changes satisfy the rule's predicate, the application will transit from its current state to a new one, along with the specified actions

[1] PhoneAdapter, a successor of ContextPhone (Context Phone, 2005), is a benchmark application used in the evaluation of the related study (Sama et al., 2010a). We re-implemented it as a real Android application and made it open-source for research purposes (PhoneAdapter, 2012).

performed. To facilitate user configuration, predicates should be simple yet expressive enough. We observe that rule predicates are typically expressed with the following syntax (SweetDreams, 2012; Tasker, 2012; Locale, 2012; PhoneAdapter, 2012).

$$\text{Predicate} \rightarrow \text{Clause} \mid \text{Clause} \vee \text{Predicate}$$
$$\text{Clause} \rightarrow \text{Atom} \mid \neg\text{Atom} \mid (\text{Atom} \hat{} \text{Clause})$$
$$\text{Atom} \rightarrow \text{p(x)}$$

A rule predicate is a disjunction of clauses, each of which is a conjunction of propositional atoms (i.e., disjunctive normal form). A propositional atom $p(x)$ is a function that produces a truth value (i.e., true or false) by evaluating its *context variable* x, which stores environmental attributes. For example, the following propositional atoms evaluate whether the user is driving fast or slow.

$$slowDriving(GPS.speed) = true \text{ if } 20km/h \leq GPS.speed < 70km/h$$
$$fastDriving(GPS.speed) = true \text{ if } 70km/h \leq GPS.speed \leq 350km/h$$

Propositional atoms tend to be correlated. In this paper, we study two types of correlations between propositional atoms: *internal correlations* and *external correlations*. Internal correlations capture an RBA's internal adaptation logics, and constitute our domain model. External correlations capture the hidden features of an RBA's running environment, and constitute our environment model. We formally define the two types of correlations in the following.

Definition 1. Two propositional atoms $p(x)$ and $p'(y)$ are internally correlated if x and y refer to the same context variable.

For example, *fastDriving* and *slowDriving* are internally correlated as their Boolean outcome depends on two different ranges of the same context variable *GPS.speed*. They are logically inconsistent as they can never both be true. Another example is the semantic entailment correlation between LOC_{Office} and GPS_{On}. They are considered to be internally correlated because their context variables *GPS.Status* and *GPS.Location* refer to the same higher level context variable *GPS*.

Definition 2. Two propositional atoms $p(x)$ and $p'(y)$ are externally correlated if x and y refer to different context variables, but their values are correlated due to environmental dynamics.

For example, LOC_{Office} and $WiFi_{Home}$ (defined in Table 1) are externally correlated. For most users, the context variable *WiFi.AccessPoint* cannot have the value "HomeWiFi" when the *GPS.Location* equals "Office", and vice versa. As a result, LOC_{Office} and $WiFi_{Home}$ are logically inconsistent. Due to the probabilistic nature of a physical environment, external correlations may vary with each individual user. For instance, the logical inconsistency between LOC_{Office} and $WiFi_{Home}$ may not exist for users who live very close to their offices. Instead, for those users, LOC_{Office} and $WiFi_{Home}$ could be externally correlated in the way that they often simultaneously hold.[2]

2.2. Finite state modeling of RBAs

The event-condition-action computing paradigm makes it natural to model the adaptive behavior of RBAs using finite-state machines. In our work, we follow the typical A-FSM modeling in the literature (Sama et al., 2010a). An A-FSM is a tuple $<S, R, A, T>$. S is a finite set of states. R is a finite set of rule predicates. A is a finite set of actions. $T \subseteq S \times R \times S \times A$ is a quaternary relation representing state transitions when rule predicates are satisfied. The A-FSM model can be statically derived from an RBA's adaptation rules.

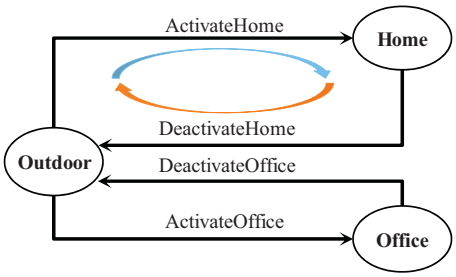

Fig. 2. A-FSM example.

Fig. 2 pictorially illustrates the A-FSM model for an example application whose adaptation rules are given in Table 1. Nodes and edges in Fig. 2 represent states and state transitions, respectively. The corresponding adaptation rules are specified in the text above each edge. This A-FSM models the application's adaptive behavior. For instance, when the application (currently in the "*Outdoor*" state) detects that the mobile device is connected to the network "HomeWiFi", it will change its state to "*Home*" and sets its user's favorite song as the ring tone. Although this simple A-FSM only contains three states, adaptation fault detection requires all possible value combinations of the concerned propositional atoms at each state visited by the A-FSM during a feasible execution to be analyzed. This makes adaptation fault detection a non-trivial task.

2.3. Adaptation fault detection

Sama et al. identified a set of adaptation fault patterns, each of which describes a common type of abnormal adaptation of RBAs (e.g., the non-deterministic adaptation in Section 1) (Sama et al., 2010a). They provide static model checking algorithms for detecting each pattern of faults. The algorithms derive an A-FSM from an RBA's adaptation rules and exhaustively explore its feasible behaviors to search for abnormal adaptations.

Let us use a running example to illustrate the checking process. We involve only non-deterministic and unstable adaptation faults in our example. A full pattern list can be found in the related study (Sama et al., 2010a). The truth value vectors of the five propositional atoms (GPS_{On}, LOC_{Home}, LOC_{Office}, $WiFi_{Home}$, $WiFi_{Office}$) in Table 1 drives the execution of the A-FSM in Fig. 2 because whether to trigger state transitions depends on the Boolean outcome of rule predicates, which are determined by the truth value of concerned propositional atoms. We start from checking the "*Outdoor*" state. The algorithm requires enumerating all possible truth value vectors of the propositional atoms (2^5 possibilities in total). The checker executes the A-FSM using each truth value vector as a stimulus under the "*Outdoor*" state, and searches for abnormal adaptations. Consider a truth value vector with only GPS_{on}, LOC_{Office} and $WiFi_{Home}$ being true. The checker finds that the A-FSM can transit from the "*Outdoor*" state to both "*Home*" and "*Office*" states non-deterministically. What is worse, if the A-FSM chooses to transit to the "*Home*" state, it may transit back to the "*Outdoor*" state immediately as the predicate of the rule "DeactivateHome" is also satisfied. When it encounters the same non-deterministic choice again, by sticking with the old choice, it may repeat the previous meaningless adaptation cycle as marked in Fig. 2. In this case, the checker will report that a non-deterministic adaptation would occur at the "*Outdoor*" state under the triggering situation described by the truth value vector example, and this non-determinstic adaptation may further result in unstable adaptations. After that, the checker proceeds to check "*Home*" and "*Office*" states.

[2] In this case, it is not suggested for users to configure adaptation rules similar to those in Table 1 as neither GPS location nor WiFi access point are good indicators of their contextual situation. Instead, they may use Bluetooth devices to better differentiate the home and office situations (see Rule 1 in Section 1 for an example).

Table 1
Adaptation rules for the example application.

Rule name	Current state	Predicate	New state	Actions	Priority level
ActivateHome	Outdoor	$WiFi_{Home} \vee (GPS_{on} \wedge LOC_{Home})$	Home	Set favorite song as ring tone	1
DeactivateHome	Home	$\mid GPS_{on} \wedge \neg LOC_{Home}$	Outdoor	Enable vibration and loud ring volume	2
ActivateOffice	Outdoor	$WiFi_{office} \vee (GPS_{on} \wedge LOC_{office})$	Office	Enable silent mode	1
DeactivateOffice	Office	$GPS_{on} \wedge \neg LOC_{office}$	Outdoor	Enable vibration and loud ring volume	1
Propositional atom		Explanation			
GPS_{on}		True if and only if GPS is enabled			
$LOC_{Home}(LOC_{office})$		True if and only if GPS reports current location as "Home" ("Office")			
$WiFi_{Home}(WiFi_{office})$		True if and only if the phone is connected to the access point "HomeWiFi" ("OfficeWiFi")			

Such an exhaustive search guarantees the absence of false negatives in fault detection. However, it can report a long list of faults containing many false positives. *We consider a reported fault as a false positive if its triggering situation can never happen in the application's specific running environment*. For instance, the fault detected in the running example is a false positive for those users who do not live near their company.

3. Approach

In this section, we introduce our approach in detail. We first present how to automatically derive domain and environment models, and concretely formulate them as deterministic and probabilistic constraints. We then show how to utilize these constraints together with a dynamic fault ranking technique to systematically process the fault reports generated by existing static model checkers for achieving effective adaptation fault detection.

3.1. Technique overview

RBAs allow users to customize their application behavior by configuring adaptation rules. The rules can be reconfigured when users have new requirements (e.g., when switching their working environments). The state-of-the-art static model checker (Sama et al., 2008, 2010a) can be applied to analyze whether the configured rules would introduce adaptation faults. The static checker exhaustively explores an A-FSM's state space and would not miss any potential fault. However, such completeness is not meaningful as a large proportion of reported faults could be spurious. Our approach mitigates this false positive problem by leveraging the domain and environment models derived for an RBA. The domain model consists of the internal correlations between propositional atoms, which capture an RBA's internal adaptation logic. The environment model comprises the external correlations between propositional atoms, which capture the hidden features of an RBA's running environment.

Some internal correlations (e.g., the logical inconsistency between *fastDriving* and *slowDriving*) are statically derivable from an RBA's adaptation rules using constraint solving techniques. We formulate these correlations as deterministic constraints, which can be used to filter out false positives from a list of faults[3] as we discussed in Section 1. However, internal correlations may involve sophisticated semantics, making their inference beyond the deduction capability of general constraint solving techniques. For example, no existing constraint solvers can infer the semantic entailment relationship between LOC_{Office} and GPS_{On}. Similarly, external correlations vary with users and RBAs' running environments, and cannot be statically inferred. We therefore take a probabilistic path to infer these two types of correlations, which will be collectively referred to as "likely correlations" hereafter. More specifically, we perform pattern-based data mining over environmental information collected during an RBA's runtime to infer likely correlations. They are then formulated as probabilistic constraints with different confidence values to help rank the faults that cannot be classified as false positives by deterministic constraints.

Fig. 3 gives a clear overview of the workflow of our model derivation and fault report processing approach. By pruning false positives using deterministic constraints and ranking any remaining faults using probabilistic constraints, true positives can be promoted to the top of the fault list. When faults are presented to users, they typically inspect a few top-ranked ones to understand potential abnormal adaptations and consider solutions. Each fault report contains actionable information regarding the fault type and the situation that triggers it. This facilitates users to validate a fault by judging the feasibility of its triggering situation. The validation made by users is then used as feedback to dynamically adjust the confidence values of probabilistic constraints and re-rank uninspected faults so that true positives can be ranked more favorably. By doing so, the users' fault inspection effort is further minimized.

In the remainder of this section, we give details about our automated correlation (formulated as constraints) inference algorithm and dynamic fault ranking strategy. To facilitate correlation inference, our approach first statically analyzes an RBA's adaptation rules to derive the set of propositional atoms, denoted as P.

3.2. Inference of deterministic constraints

Internal correlations model an RBA's adaptation logic, and constitute our domain model. Our approach analyzes propositional atoms in pairs to derive internal correlations. Specifically, we detect *inconsistent truth value vectors* of a given propositional atom pair $(p(x), p'(y))$. An inconsistent truth value vector is an infeasible truth value combination of the concerned propositional atoms. If such truth value vectors are discovered, they can be formulated as deterministic constraints, which should always be satisfied, to prune false positives. For example, knowing that $p(x)$ and $p'(y)$ are logically inconsistent (cannot both be true) helps formulate the following constraint.

$$p(x) \rightarrow \neg p'(y) \text{ (or equivalently } \neg p(x) \vee \neg p'(y))$$

As mentioned in Section 2.3, we consider a fault's triggering situation as a truth value vector of the concerned propositional atoms. *We define that a fault F violates a constraint c if and only if c is evaluated to be false using F's triggering situation* (meaning that this situation is internally inconsistent and therefore infeasible in practice). If such a violation is detected, the corresponding fault can be classified as a false positive. By doing so, we only remove faults that can be proved to be false positives, and thus guarantee that our approach processes fault reports in a sound way.

In order to discover the aforementioned constraint, we rely on the CHOCO constraint solver (Choco, 2012) to deduce that $\neg p(x) \vee \neg p'(y)$ is a tautology (In other words, it is infeasible for $p(x)$ and $p'(y)$ to hold simultaneously). Fig. 4 presents a straightforward

[3] Deterministic constraints can also be used to reduce the search space of the A-FSM model. See the related study (Sama et al., 2010a) for details.

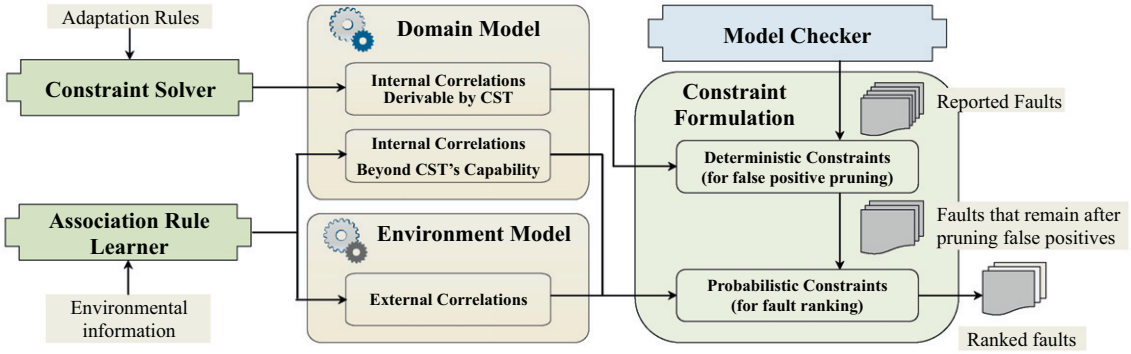

Fig. 3. Model derivation and fault reports processing workflow: "CST" stands for "Constraint Solving Techniques".

Algorithm: Deterministic Constraint Inference

Input: P: a set of propositional atoms
Output: C: a set of deterministic constraints

```
1: for each unordered pair (p(x) , p'(y)), where p(x) , p'(y)  P and p(x) ≠ p'(y)
2:     if x and y refer to the same context variable   // internal correlation
3:         if ¬(p(x) ∧ p'(y)) is a tautology
4:             add it to C   // truth value vector (true, true) is inconsistent
5:         end if
6:         if ¬(p(x) ∧ ¬p'(y)) is a tautology
7:             add it to C   // truth value vector(true, false) is inconsistent
8:         end if
9:         if ¬(¬p(x) ∧ p'(y)) is a tautology
10:            add it to C   // truth value vector (false, true) is inconsistent
11:        end if
12:        if ¬(¬p(x) ∧ ¬p'(y)) is a tautology
13:            add it to C   // truth value vector (false, false) is inconsistent
14:        end if
15:    end if
16: end for
```

Fig. 4. Deterministic constraint inference algorithm.

algorithm for inferring different types of deterministic constraints. It enumerates all unordered pairs of distinct propositional atoms. For each of such pairs $(p(x), p'(y))$, if x and y refers to the same context variable (i.e., $p(x)$ and $p'(y)$ are internally correlated), then Choco tries to discover infeasible truth value combinations of the two atoms. If they are discovered, the algorithm formulates them as deterministic constraints.

Algorithm discussion: The algorithm performs pairwise analyses. In the worst case, the time complexity is $O(|P|^2 \times T)$, where T is the timeout limit for a constraint solver to finish a proof. We show later in our experiments that this complexity is affordable as the number and size of the rules are typically limited for ordinary users.

From the algorithm, we can also see that our inferred deterministic constraints are binary relations. We could certainly explore deterministic constraints of a more complex form (e.g., ternary relations). However, that simply shifts the burden to the constraint solvers. We later demonstrate that binary relations are a good tradeoff between efficiency[4] (i.e., how much resource is needed to infer them) and performance (i.e., how good they are at pruning false positives).

3.3. Inference of probabilistic constraints

Despite the advances in constraint solving techniques, existing tools like Choco (Choco, 2012) are still limited in deducing semantic internal correlations. Besides, external correlations cannot be inferred statically due to environmental dynamics (recall the definition of external correlation in Section 2.1). Therefore, we statistically analyze the dynamically collected environmental information to infer the following two types of likely correlations:

- *Type 1*. Internal correlations whose deduction is beyond the capability of existing constraint solvers;
- *Type 2*. External correlations that are inherently probabilistic.

3.3.1. Projecting environmental information

Environmental information is a collection of data sensed by an RBA at runtime (e.g., location, joined network etc.). It captures the features of an RBA's running environment, and therefore is crucial for constructing the environment model. As noise inevitably exists in these sensed data, environmental information may contain

[4] We are concerned with the efficiency of deriving constraints because the ideal platform to deploy our technique is a mobile device which typically has constrained resources.

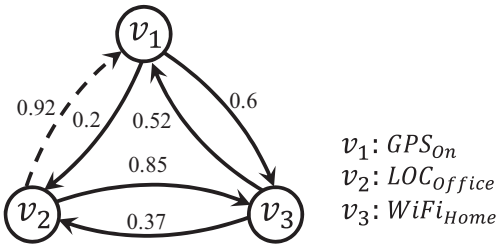

Fig. 5. An example constraint graph.

inconsistencies (Xu and Cheung, 2005). For instance, it is possible for a mobile device's GPS module to report its current location as "Gym" while this device is connected to a wireless network with the identifier "LibraryWifi". To enhance the quality of environmental information, our approach relies on *Cabot* (Xu et al., 2004), a context management middleware, to resolve such inconsistencies with our best effort.

After inconsistency resolution, the environmental information is a collection of entries recording the latest values of environmental attributes (e.g., *Location* = "Home", *Network* = "HomeWifi", etc.). Such raw data is not convenient for statistical analysis. As we are interested in inferring likely correlations between propositional atoms, our approach converts the environmental information into a dataset of truth value vectors by projecting it onto the propositional atom set (i.e., the set P). Specifically, we use each entry in the environmental information to evaluate the propositional atoms in P, and store the Boolean outcomes as our truth value vectors.

3.3.2. Organizing probabilistic constraints

Likely correlations are formulated as probabilistic constraints in our approach. For example, LOC_{Office} and GPS_{On} are likely correlated: if LOC_{Office} is evaluated to be true, then GPS_{On} is likely to be evaluated to be true. By statistically analyzing the environmental information, we can infer the correlation's probability (quite high). Then this correlation can be used as a constraint with the confidence value set to this correlation's probability. From now on, we use the terms likely correlation and probabilistic constraint interchangeably with the same meaning. In our approach, we organize all such probabilistic constraints in a weighted directed graph, called *constraint graph*. It is a directed graph because the correlation between two propositional atoms is not necessarily bi-directional (e.g., the correlation between LOC_{Office} and GPS_{On}). Formally, we define a *constraint graph* as a tuple $G = <V, E, W>$:

- V is a set of vertices. Each propositional atom $p(x) \in P$ is mapped to two vertices in V. One vertex represents the positive truth value assignment of $p(x)$, denoted $p(x)_\top$. The other vertex represents the negative truth value assignment of $p(x)$, denoted $p(x)_\bot$.
- E is a set of ordered pairs of vertices, called edges. An edge (v_i, v_j) is associated with a likely correlation between v_i's and v_j's corresponding propositional atoms. For example, the edge for the aforementioned likely correlation starts from the vertex $LOC_{Office\top}$, and ends at the vertex $GPS_{On\top}$.
- W is a matrix of numerical values representing the weight of each edge. An edge's weight indicates the probability of the correlation associated with this edge (i.e., also the confidence value of the corresponding constraint).

Fig. 5 presents an example constraint graph. The propositional atoms used in the graph have been defined earlier in Table 1. The dashed edge represents the aforementioned probabilistic constraint: if LOC_{Office} holds, then GPS_{On} also holds. The confidence value of this constraint is 0.92 (highly probable). In the following subsection, we show how the confidence value of a probabilistic constraint can be computed.

Table 2
Example dataset of truth value vectors.

Truth value vector	$WiFi_{Office}$	$WiFi_{Home}$	GPS_{On}	LOC_{Office}	LOC_{Home}
t_{v1}	True	False	False	False	False
t_{v2}	False	False	True	True	False
t_{v3}	True	False	True	True	False
t_{v4}	False	False	False	False	True
t_{v5}	False	False	False	True	False

3.3.3. Computing constraint confidence values

In the data mining area, association rule learning is a popular and well-studied technique for discovering interesting relations between variables in a large dataset (Alpaydin, 2010). In our approach, probabilistic constraints model the correlations between propositional atoms. Therefore, association rule learning can be naturally adapted to compute the confidence values of our probabilistic constraints. In our problem, P is a set of propositional atoms, and the association rules are in the following form:

$$(p(x) = a) \Rightarrow (p'(y) = b), \text{where } p(x), p'(y) \in P, \text{and } a, b \in \{true, false\}$$

This indicates that if $p(x)$ has the truth value a, then $p'(y)$ is likely to have the truth value b. This form of association rule corresponds to the probabilistic constraints in our work. Therefore, our objective is to learn the confidence of such association rules via mining a dataset of truth value vectors, which is obtained by projecting environmental information. Each vector in the dataset specifies the truth value assignments to the propositional atoms in P.

We adapt a widely used support-confidence framework (Agrawal et al., 1993) to compute the confidence values of probabilistic constraints. We explain the concepts of *support* and *confidence* below, and use an example dataset in Table 2 to illustrate how to compute them.

- The ***support*** $supp\,(p(x) = a)$ is defined on the dataset of truth value vectors, and gives the proportion of those truth value vectors that contain the occurrence of $p(x) = a$. For example, from Table 2, we can compute $supp(LOC_{Office} = \text{true}) = 3/5 = 0.6$.
- The ***confidence*** of an association rule $(p(x) = a) \Rightarrow (p'(y) = b)$ is defined as the probability of observing this rule's consequent $p'(y) = b$ in the dataset of truth value vectors under the condition of observing this rule's antecedent $p(x) = a$. Eq. (1) gives the formula for computing the rule's confidence.

$$\begin{aligned} &conf((p(x) = a) \Rightarrow (p'(y) = b)) \\ &= pr(p'(y) = b | p(x) = a) \\ &= \frac{supp(p(x) = a) \&\& (p'(y) = b)}{supp\,(p(x) = a)} \end{aligned} \quad (1)$$

For example, from Table 2, we can compute $conf((LOC_{Office} = \text{true}) \Rightarrow (GPS_{On} = \text{true})) = 0.4/0.6 = 0.67$. This indicates that the probabilistic constraint "if LOC_{Office} holds, then GPS_{On} also holds" has a confidence value of 0.67. This example is only for illustration purposes. In practice, due to inevitable noise, such an inferred probabilistic constraint needs the support of at least tens of truth value vectors in order to be considered statistically significant.

3.4. Dynamic fault ranking

Fault ranking complements static model checking techniques that may suffer from false positives. A good ranking scheme is able to rank the most likely true positives to the top of a fault

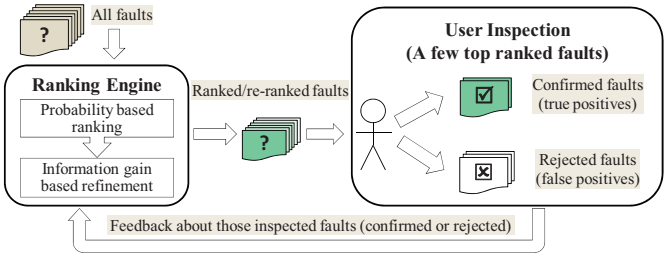

Fig. 6. An iteration in the dynamic fault ranking process.

list, while relegating highly possible false positives to the bottom. Generally speaking, fault ranking can be categorized into two types: *static fault ranking* and *dynamic fault ranking*. Static fault ranking associates each fault with a fixed probability of being a true positive (or probability for short) (Kremenek and Engler, 2003). Based on these fixed probabilities, all faults are ranked and presented to users for inspection. During inspection, the ranking order of these faults remains unchanged. On the contrary, dynamic fault ranking allows each uninspected fault's probability to be dynamically tuned (Kremenek et al., 2004) according to users' feedback about the inspected faults (confirming their validity or rejecting them as false positives). This allows more true positives to be re-ranked to the top of the fault list, and at the same time more false positives to be re-ranked to the bottom.

In this section, we present our dynamic fault ranking technique, which requires little manual effort. Our dynamic ranking process works in an iterative way as illustrated in Fig. 6. It starts when a list of faults is generated and ends when all faults are inspected or users choose to terminate it. In each iteration, our ranking engine ranks all faults before presenting them to the users. Users typically choose the top ranked faults for inspection. They can confirm the validity of these faults or reject them as false positives. Since top ranked faults are few in number, users can afford to validate them manually. Our ranking technique leverages the validation results (i.e., these inspected faults are confirmed or rejected) as feedback to re-rank remaining uninspected faults. This allows our technique to further minimize user's effort in fault inspection by favorably re-ranking the faults that are more likely to be real among the uninspected faults. In the following, we introduce our ranking scheme in Section 3.4.1, and discuss how to leverage the validity feedback of the inspected faults to re-rank the remaining uninspected faults in Section 3.4.2.

3.4.1. Ranking scheme

The selection of a ranking scheme can largely affect the quality of fault ranking. As illustrated in Fig. 6, we choose a ranking scheme that works in two phases: (1) it first ranks all faults based on their *probabilities*; (2) it then leverages *information gain* to refine the ranking order, in particular for those top ranked faults when their probabilities are very close to each other. Before introducing these two ranking metrics (i.e., probability and information gain), we clarify two characteristics of our approach below:

- *Characteristic 1*. A probabilistic constraint describes a likely correlation between two propositional atoms. A fault's triggering situation is a truth value vector for its concerned propositional atoms. Therefore, a fault *F relates to* a probabilistic constraint *c* only when constraint *c*'s propositional atoms are involved in fault *F*'s triggering situation.
- *Characteristic 2*. Our earlier constraint graph offers a set of probabilistic constraints labeled with their confidence values. A fault may violate some of these constraints (recall the constraint violation definition in Section 3.2).

With these two characteristics, we use a weighted average function (Eq. (2)) to measure how strong a fault *F* conflicts with a set *LC* of probabilistic constraints, denoted *ConflictStrength*(*F*, *LC*). This function puts more emphasis on those constraints with high confidence values. In Eq. (2), *conf*(*c*) represents the confidence that a probabilistic constraint *c* is a real constraint (defined earlier in Eq. (1)). Functions *rel*(*F*, *c*) and *vio*(*F*, *c*) test whether a fault *F* relates to or violates a probabilistic constraint *c*, respectively (see the above two characteristics). Note that *vio*(*F*, *c*) implies *rel*(*F*, *c*) as a fault can only violate its related constraints.

$$ConflictStrength(F, LC) = \frac{\sum_{c_i \in LC}(conf(c_i) \cdot vio(F, c_i))}{\sum_{c_i \in LC}(conf(c_i) \cdot rel(F, c_i))}, where$$
$$rel(F, c_i) = \begin{cases} 1, & if\ F\ relates\ to\ c_i \\ 0, & otherwise \end{cases}, vio(F, c_i) = \begin{cases} 1, & if\ Fviolates c_i \\ 0, & otherwise \end{cases} \quad (2)$$

We then define a fault *F*'s probability of being a true positive (or probability for short; our first ranking metric) with respect to a set *LC* of probabilistic constraints *PBT*(*F*, *LC*) in Eq. (3). All faults are first ranked in the decreasing order of their probabilities.

$$PBT(F, LC) = 1 - ConflictStrength(F, LC) \quad (3)$$

However, we often observe that the probabilities of the top ranked faults are very close to each other with a difference of less than 0.01. This suggests that using the ranking metric of probability alone is inadequate. As such, we further refine the ranking order for those top ranked faults before presenting them to users. Our insight is that inspecting two different faults would contribute different levels of useful information that can help better re-rank other uninspected faults. Here, "useful" can be interpreted based on the following two observations:

- *Observation 1*. If a fault *F* can later be confirmed by users as a true positive, the probabilistic constraints that *F* has violated become invalid (because a true fault should not violate any valid constraint). Their confidence values can thus be set to 0.
- *Observation 2*. If a fault *F* can later be rejected by users as a false positive, the confidence values of the probabilistic constraints that *F* has violated should increase (because these constraints have helped identify false positives). The choice of the increment value depends on the quality of the environmental information. If the environmental information is less subject to noise, a larger increment value can be used. Otherwise, the value should be chosen conservatively (in our experiment, we set the value to 0.05).

With these two observations, we adopt an idea that was inspired by Kremenek et al.'s work (Kremenek et al., 2004) for ranking refinement: A fault whose inspection can offer a larger amount of useful information should be chosen as the next candidate for inspection. We elaborate this idea below.

In information theories, *entropy* measures the uncertainty of a random variable. The *mutual information* $I(X; Y)$ between two random variables *X* and *Y* quantifies the amount of information *X* can tell about *Y* in terms of how much *Y*'s uncertainty can be reduced due to the knowledge of *X* (Cover and Thomas, 2006). The following Eq. (4) formally defines such mutual information. $H(Y)$ in Eq. (4) is variable *Y*'s entropy (i.e., uncertainty) before knowing variable *X*'s value. Conditional entropy $H(Y|X)$ represents the remaining uncertainty of variable *Y* after knowing variable *X*'s value.

$$I(X; Y) = H(Y) - H(Y|X) \quad (4)$$

We can generalize this metric to quantify how much information we can get about the remaining faults $\boldsymbol{U}$, if a fault *F* is the next

candidate to be inspected. The generalized metric *information gain* is our second ranking metric. We denote it as $I(F; \boldsymbol{U})$.

$$\begin{aligned} I(F;\boldsymbol{U}) &= \sum_{U_i \in \boldsymbol{U}} I(F;U_i) \\ &= \sum_{U_i \in \boldsymbol{U}} (H(U_i) - H(U_i|F)) \\ &= \sum_{U_i \in \boldsymbol{U}} \left(\sum_{f \in \{true,false\}} \sum pr(F=f)(H(U_i) - H(U_i|F=f)) \right) \end{aligned} \quad (5)$$

Eq. (5) is derived from Eq. (4) to measure information gain. For each remaining fault $U_i \in \boldsymbol{U}$, $I(F; U_i)$ represents the amount of information that a fault F can tell about the fault U_i. Specifically, $I(F; U_i)$ quantifies how much U_i's uncertainty can be reduced if fault F's validity is known (i.e., $H(U_i) - H(U_i|F)$). From our two observations, one can see that whether F is confirmed (F = true) or rejected (F = false) later, fault U_i's probability would be affected (so would fault U_i's uncertainty). This is because the confidence values of the probabilistic constraints fault F violated would be adjusted. Then the information gain $I(F; \boldsymbol{U})$ of fault F can simply be defined as the summation of all such $I(F; U_i)$, where $U_i \in \boldsymbol{U}$. We omit more derivation details here. Interested readers can refer to Cover et al.'s work (Cover and Thomas, 2006) for details about entropy computation.

In brief, our ranking scheme works as follows: when two ranking metrics are used together, faults are first ranked by their probabilities. If the probabilities of those top ranked faults are very close to each other, then the faults that carry relatively larger information gain values would be promoted for breaking such tie cases.

3.4.2. Leveraging feedbacks

We have explained how to rank a list of faults according to their probabilities and information gains. These ranked faults are presented to users for inspection. Users can freely validate a few top ranked faults. Their validation feedbacks (i.e., the faults are confirmed or rejected) would be used to improve the ranking of remaining faults.

To understand how it works, one should note that the probability of a fault relies on the confidence values of the probabilistic constraints the fault relates to (see Eq. (3)). Our preceding two observations in Section 3.4.1 indicate that the user's feedback would help adjust the confidence values of such probabilistic constraints. Therefore, the probabilities of the remaining (uninspected) faults would change accordingly. This leads to a re-ranking of these faults, thus completing one iteration in our dynamic fault ranking process. More iterations can be conducted as required. In practice, confirming and fixing a fault may greatly reduce the number of remaining faults as we will show in Section 4.

4. Evaluation

In this section, we present the empirical evaluation of our approach. First, we describe the implementation of AFChecker in Section 4.1. Next, we present our evaluation, including the research questions, the experimental setup and the evaluation results in Sections 4.2–4.6. Finally, we discuss threats to validity in Section 4.7.

4.1. Implementation

We implemented our approach as a publically available Java library named AFChecker.[5] AFChecker takes as input a set of adaptation rules, and outputs a list of detected adaptation faults. It has three major components:

- *Model checker*. The central component of AFChecker is a model checker based on the state-of-the-art technique (Sama et al., 2010a). The model checker derives a state transition model from a set of user-configured adaptation rules and verifies the model to detect five common types of adaptation faults: (1) non-deterministic adaptations, (2) dead rule predicates, (3) dead states (meaning that no rules can be satisfied in these states), (4) adaptation races, and (5) unreachable states (meaning that the states cannot be transitively reached from other states).
- *Constraint inference engine*. The constraint inference engine of AFChecker infers both deterministic and probabilistic constraints. AFChecker relies on the Choco constraint solver (Choco, 2012) to derive deterministic constraints by analyzing the propositional atoms in the user-configured adaptation rules. AFChecker also embeds an association rule learner to infer probabilistic constraints by analyzing the environmental information.
- *Fault report processor*. AFChecker's fault report processor processes the fault reports generated by its underlying model checker. It prunes a subset of fault reports using deterministic constraints, and prioritizes the remaining reports using probabilistic constraints. In our implementation, AFChecker can interact with users in two modes: (1) non-interactive mode, and (2) interactive mode. In the non-interactive mode, the ranking of fault reports is static and AFChecker simply generates a list of prioritized fault reports for users' inspection. In the interactive mode, the ranking of fault reports is dynamic. AFChecker iteratively presents a few highly possible faults for users' inspection and asks users to confirm their validity. Based on users' feedbacks, AFChecker dynamically adjusts the probability of the uninspected faults and favorably re-ranks them. By such interaction, users can quickly realize the misconfigurations in their configured adaptation rules.

4.2. Research questions

We evaluate AFChecker using controlled experiments, which are designed to study the following three research questions (RQ1–3):

- *RQ1*: Are our derived deterministic constraints effective for pruning false positives in adaptation fault detection?
- *RQ2*: Are our inferred probabilistic constraints effective for ranking reported faults? How does collected environmental information affect our fault ranking quality?
- *RQ3*: Can our dynamic ranking strategy effectively improve the quality of our fault ranking?

Among these research questions, RQ1 and RQ2 are designed for evaluating the practical usefulness of our extracted domain model and environment model. RQ3 is for evaluating the effectiveness of our dynamic ranking strategy.

4.3. Experimental setup and design

We selected two popular RBAs, PhoneAdapter (2012) and Tasker (2012) as our experimental subjects. For PhoneAdapter, we used the same adaptation rules as in the related study (Sama et al., 2010a). For Tasker, one of the top 10 awardees from the Google Android Developer Challenge 2 (ADC2) (Google ADC 2 Winners, 2009), we used the adaptation rules (listed in Table 3) that have been discussed in Tasker's forum with slight modifications for syntactic reasons (Rule Discussion, 2012a,b,c,d,e). The use of propositional atoms in these rules is self-explanatory (refer to Table 1 for

[5] We make AFChecker and its documentation publically available at http://www.cse.ust.hk/~andrewust/afchecker/index.html

Table 3
Adaptation rules for Tasker.

Rule name	Current state	Predicate	New state	Actions	Priority
ActivateHome	Outdoor	$WiFi_{Home} \vee (GPS_{on} \wedge LOC_{Home})$	Home	Set user's favorite song as ring tone	Normal
ActivateOffice	Outdoor	$WiFi_{office} \vee (GPS_{on} \wedge LOC_{office})$	Office	Enable silent mode with vibration on	High
ActivateOutdoor	Office, Home	$GPS_{on} \wedge \neg LOC_{office} \wedge \neg LOC_{Home}$	Outdoor	Enable vibration and loud ring volume	Normal
ActivateMeeting	Office	$Time_{Meeting\ Start}$	Meeting	Start recording audio	High
DeactivateMeeting	Meeting	$Time_{Meeting\ End}$	Office	Stop recording audio	High
ActivateRunning	Outdoor	$GPS_{on} \wedge LOC_{park} \wedge Speed_{Fast}$	Running	Start counting steps (use step counter applications)	Low
DeactivateRunning	Running	$GPS_{on} \wedge LOC_{park} \wedge Speed_{Slow}$	Outdoor	Stop counting steps	Low
ActivateSleeping	Home	$Time_{Midnight}$	Sleeping	Enable airplanemode	High
DeactivateSleeping	Sleeping	$Time_{Morning}$	Home	Disable airplane mode and start alarm	High
ActivatePowerSaving	Outdoor	$Battery_{Low}$	PowerSaving	Disable network connection	Normal
DeactivatePowerSaving	PowerSaving	$\neg Battery_{Low}$	Outdoor	Enable network connection	Normal

examples). We ran these two RBAs on an HTC Desire S510e smartphone for two weeks, and collected the phone's raw sensory data every 5 min. From them, we obtained a dataset with 1517 pieces of environmental information, which were converted into truth value vectors for experimental purposes. Our following experiments with AFChecker ran on a Macbook Pro with a 2.4 GHz dual-core CPU and 4 GB RAM. For AFChecker's association rule learning algorithm, we set the support and confidence thresholds to 0.25 and 0.5, respectively. We also set the timeout limit for constraint solving to 200 ms. All experiments for PhoneAdapter finished within 800 ms and cost less than 26 MB memory. All experiments for Tasker finished within 1100 ms and cost less than 34 MB memory. As the number and size of the rules are limited for ordinary users, such analysis overhead makes it possible for RBA developers to integrate our AFChecker into their applications.[6]

As mentioned in Section 3.1, AFChecker processes the faults reported by its underlying model checker to mitigate the false positive problem. Before evaluating its effectiveness, we need to manually validate all reported faults (i.e., determine the ground truth). Since there were many faults reported, we first analyzed all adaptation rules of both RBAs to manually infer their constraints with best effort, and then used the inferred constraints to identify false positives in all reported faults. We note that inferring such constraints requires profound analysis and exhaustive search. Even for RBAs with only ten rules, users may need to analyze hundreds of candidates to recognize constraints. Hence, it is impractical to infer every constraint manually. As such, we carefully validated each fault that does not violate any inferred constraint to confirm or reject it. With such ground truth, we then designed three sets of experiments to answer our aforementioned research questions RQ1–3.

Experiment set 1. To answer research question RQ1, we ran AFChecker to derive deterministic constraints, and then used them to classify reported faults. We conducted a study on the number of false positives identified by these constraints. The results are presented in Section 4.4.

Experiment set 2. To answer research question RQ2, we ran AFChecker to infer probabilistic constraints, and then used them for fault ranking. Note that fault ranking is conducted after removing false positives using deterministic constraints. We gradually increased the size of the dataset used for learning the confidence values of probabilistic constraints, and studied how the quality of our fault ranking would be affected accordingly. We used Discounted Cumulative Gain (DCG) to measure the overall fault ranking quality (Järvelin and Kekäläinen, 2002). DCG is a metric often used in information retrieval for evaluating the effectiveness of web search engine algorithms, and also suitable for measuring the quality of a general ranking algorithm. Let us briefly review DCG's basic idea. A list contains many entries for ranking and each entry originally has a relevance value. A good algorithm is supposed to highly rank the entries with high relevance values. After ranking, one would measure the *gain* of each entry, which is proportional to this entry's relevance value, and inversely proportional to its position in the ranked list. Then the gain accumulated from the first entry to the last entry in the ranked list (called the DCG value) can be used to measure the quality of the whole ranking. We formally define the DCG metric in the following Eq. (6). rel_i represents the relevance value of the entry at position i in a ranked list, and p is the size of this list.

$$DCG_p = rel_1 + \sum_{i=2}^{p} \frac{rel_i}{\log_2 i} \tag{6}$$

In our case, an entry's relevance can be set to 1 if this entry is a true positive and 0 for a false positive. A perfect ranking with all true positives ranked before any false positive would yield an ideal DCG value. We use this value as the baseline and normalize it to 1.00. The results presented in Section 4.5 would be normalized accordingly for comparison. In such a setting, *a higher DCG value would indicate a better overall ranking quality*.

In practice, users are likely only interested in top ranked faults, therefore we also measured the percentage of true positives in the selected top ranked faults. Measurements were made based on the top 26.5% and 10% faults for PhoneAdapter, and top 21.7% and 10% faults for Tasker, respectively. We explain why we selected these percentages in Section 4.5.

Experiment set 3. To answer research question RQ3, we applied our dynamic fault ranking strategy and automated its iterative process. When a list of faults was ranked, its top ranked fault was selected for inspection. With the support of our earlier determined ground truth, this fault was confirmed or rejected. Such feedback was then used to re-rank the remaining faults. This process terminated after all faults were inspected, and we thus generated a whole ranking for all faults according to their inspection order (i.e., being inspected earlier means a higher ranking). After that, we computed the DCG value for this whole ranking, and measured the percentages of true positives in the selected top ranked faults for ranking quality comparisons. In some scenarios, users may choose to immediately fix confirmed faults in an inspection. We, therefore, conducted an extra experiment to study how the number of remaining faults would change if one fixes the top three confirmed faults reported by our dynamic fault ranking technique. We report these results in Section 4.6.

4.4. Pruning false positives by deterministic constraints

Fig. 7 presents our experimental results for answering research question RQ1. It reports about two RBAs: (1) the numbers and percentages of true and false positives in all reported faults, (2) the numbers and percentages of those faults that can be

[6] We provide a tutorial about how to use AFChecker at http://www.cse.ust.hk/~andrewust/afchecker/tutorial.html

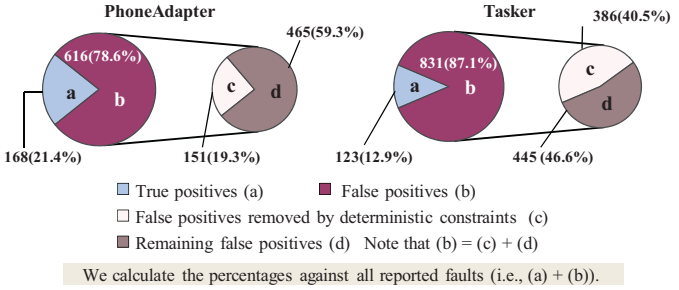

Fig. 7. Effectiveness of using deterministic constraints to prune false positives.

categorized as false positives using deterministic constraints inferred by our AFChecker, and (3) the numbers and percentages of remaining false positives after applying deterministic constraints. For PhoneAdapter, there are a total of 784 faults reported (non-deterministic and unstable adaptation faults). The deterministic constraints derived by AFChecker helped prune 151 false positives from these 784 faults. For Tasker, the total number of faults is 954, among which 386 were removed as false positives using the deterministic constraints[7] derived by our AFChecker. These removed false positives account for 19.3% and 40.5% of all reported faults for PhoneAdapter and Tasker, respectively. This shows that deterministic constraints are useful for pruning a lot of false positives. However, since there are many reported faults, the remaining false positives still cause a serious problem. As illustrated in Fig. 7, after applying deterministic constraints, 465 false positives in PhoneAdapter still remain, accounting for 73.5% of all remaining faults (465/(465 + 168) = 73.5%). For Tasker, the situation is even worse, only 21.7% of the remaining faults are true positives (123/(123 + 445) = 21.7%). In other words, users would encounter only one or two real faults out of any randomly inspected ten faults. Therefore, deterministic constraints are useful but inadequate for the removal of false positives.

4.5. Ranking faults by probabilistic constraints

Figs. 8–10 present our experimental results for ranking the remaining faults by probabilistic constraints. The results are for answering research questions RQ2–3. We discuss RQ2 in this section and RQ3 in Section 4.6. As mentioned earlier, we use the DCG metric to measure the overall ranking quality. Fig. 8 compares the DCG values for ranking results produced by: (1) our fault ranking scheme with the dynamic ranking *disabled* (named static ranking), (2) a random ranking algorithm, and (3) an ideal algorithm for reference. We also report the percentages of true positives of the top ranked faults for our two experimental subjects in Figs. 9 and 10. We chose to study the percentage of true positives in the top 26.5% faults (see Fig. 9) for PhoneAdapter because the total of true positives occupy exactly 26.5% of the remaining faults (168/(168 + 465) = 26.5%). For Tasker, this percentage becomes 21.7% (123/(123 + 445) = 21.7%). Using these two percentages, the ideal ranking algorithm can achieve exactly 100% for the measurement (larger percentages do not make sense as there are no more true positives). We also studied the percentage of true positives in the top 10% faults (see Fig. 10) for the two subjects. This is because we consider that it should be acceptable for users to inspect up to 10% of faults (no more than 50 faults). We garner several findings from Figs. 8 to 10:

- First, as the dataset grows in size, the DCG values for all ranked faults and the percentages of true positives in the top ranked faults generally keep increasing. This indicates that more false positives were relegated to the bottom of the fault list. As a larger dataset can provide more information for our association rule learner, the confidence values of the inferred probabilistic constraints become more reliable and stable as the dataset increases in size. As a result, these constraints helped associate false positives with lower probabilities.
- Second, AFChecker's inferred probabilistic constraints have greatly improved our fault ranking quality. For instance, Fig. 8 shows that for a random ranking, its DCG values are around 0.50. However, for our static ranking (using probabilistic constraints), its DCG values can rise to 0.83. This result is close to the ideal ranking (1.00). Figs. 9 and 10 report similar results. For example, with more environmental information (i.e., a larger dataset), the percentage of true positives in top ranked faults can reach 71.5% (for top 21.7% ranked faults) and 84.2% (for top 10% ranked faults) for Tasker (see Figs. 9(b) and 10(b)). That is almost four times better than a random ranking (15.4% and 17.5%, respectively).
- Third, as the size of the dataset grows, the quality of our fault ranking increases quickly at the beginning, and then becomes relatively stable later. This is understandable. At the beginning, with more environmental information (i.e., a larger dataset), the confidence values of the inferred probabilistic constraints become more and more reliable. This helps rank false positives low (i.e., increasing the DCG value). When the size of the dataset grows further, the confidence values of these constraints have become relatively stable (i.e., no large change). This leads to only slight variance in the ranking quality (i.e., an almost fixed DCG value).

We note that collecting a huge set of environmental information may be impractical for two reasons. First, frequently requesting updates from the sensors of a smartphone is very energy-consuming. Second, a user's physical environment tends to change. Profiling environmental information over a very long period may receive contradicting data, which can conversely lower the quality of our fault ranking. According to our experience, collecting environmental information for a few days can already improve the fault ranking quality substantially. For example, Fig. 10(a) shows that a dataset of size 400 (corresponding to about 4–5 days) can already improve the percentage of true positives in the top 10% faults to 66.7% (as compared to 22.2% by the random ranking). Therefore, we suggest that if a user has already used his application for a few days, then it would be very convenient to quickly find whether his newly configured rules would cause any adaptation faults by using our AFChecker. More importantly, all reported faults are automatically ranked with good quality, by presenting the most likely true positives at the top.

4.6. Dynamic ranking of reported faults

Figs. 8–10 also present our experimental results for evaluating whether our dynamic ranking strategy can effectively improve the fault ranking quality for PhoneAdpater and Tasker.

We observe that both the DCG values and the percentages of true positives in the top ranked faults are clearly improved after applying our dynamic ranking strategy (i.e., dynamic ranking), as compared to our static ranking. For instance, in Fig. 9(b), the percentage of true positives in the top 21.7% faults for Tasker reaches 86.2% after enabling dynamic ranking. As a comparison, this percentage never exceeds 72.0% in the static ranking. We note that a probabilistic technique may mistakenly associate a high confidence value with a spurious constraint due to noise in the dataset. Our dynamic ranking can effectively leverage a user's manual inspection feedback to remove such spurious constraints (by setting their

[7] These deterministic constraints include: (1) Locations are mutually exclusive (only one of LOC_{Home}, LOC_{Office}, LOC_{Park} can hold at a time point); (2) The logical inconsistencies between: $Time_{MeetingStart}$ and $Time_{MeetingEnd}$, $Time_{Midnight}$ and $Time_{Morning}$, $Speed_{Fast}$ and $Speed_{Slow}$.

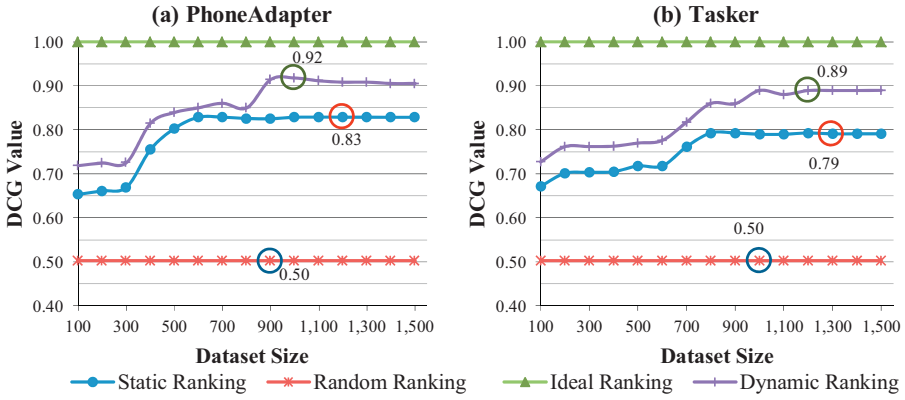

Fig. 8. Overall fault ranking quality.

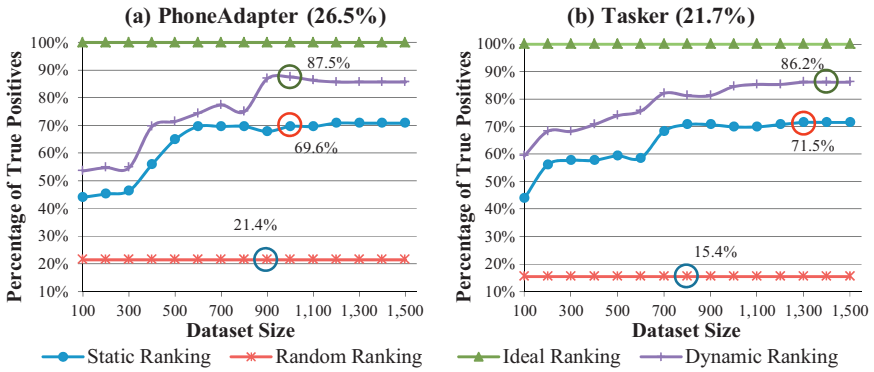

Fig. 9. Percentage of true positives in top 26.5% and 21.7% faults for PhoneAdapter and Tasker, respectively.

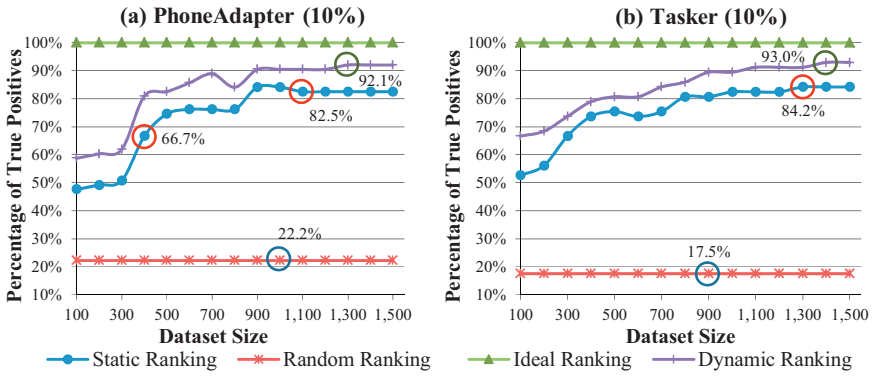

Fig. 10. Percentage of true positives in top 10% faults for PhoneAdapter and Tasker.

confidence values to 0 according to our Observation 1 in Section 3.4.1). On the other hand, if a fault is rejected by the user, the confidence values of the constraints this fault violates can instead be increased, as they contribute to discovering false positives (our Observation 2 in Section 3.4.1). In both cases, user inspection can provide useful information (we used our ground truth to play the role of user inspection in the experiments). Therefore, our dynamic ranking can effectively improve the fault ranking quality. The results in the three figures consistently confirm our expectations.

Finally, we additionally studied whether fixing confirmed faults would change the number of remaining faults. We observe a significant decrease in the number of remaining faults in Fig. 11, where the top three confirmed faults were fixed in turn. For example, the number of remaining faults is reduced from 633 to 352 after fixing the top three confirmed faults for PhoneAdapter (a scale factor of approximately (633–352)/3 = 94 times). As our dynamic fault ranking strategy faithfully follows a feedback-based paradigm (i.e., rank faults, then inspect top faults, then fix confirmed faults, and then re-rank remaining faults again), this result suggests the practical usefulness of our dynamic ranking strategy. Such a large scale factor shows that the faults are closely correlated and fixing some of them can immediately remove many others. Therefore, dynamic fault ranking would be useful for practical adaptation fault detection and fixing. Users can use Sama et al's model checking algorithm (Sama et al., 2010a) together with our AFChecker to collectively find such faults in practice.

4.7. Threats to validity

The validity of our experimental results may be subject to some threats. One is the representativeness of the adaptation

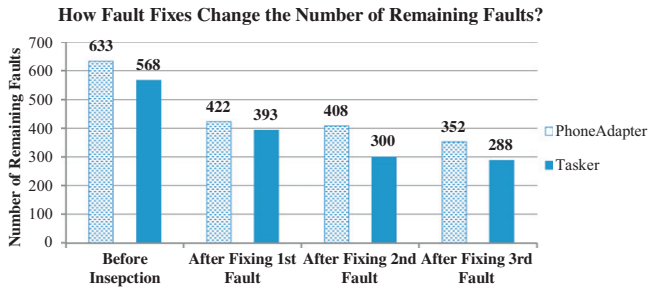

Fig. 11. How fault fixes change the number of remaining faults.

rules used in Tasker. These rules were selected from Tasker's online forum, and adapted slightly for our experiments (syntactic reasons) (Rule Discussion, 2012a,b,c,d,e). We confirmed in the forum that users often configure similar rules for handling their daily events. Table 3 lists these rules to enable interested readers to repeat our experiments. Another potential threat is the determination of the ground truth (i.e., deciding true or false positives in the reported faults). We understand that a manual process like this is essentially error-prone. To play it safe, we asked two additional researchers to independently inspect all the faults. We cross-validated their inspection results for consistency. Finally, although two experimental subjects are not substantial, we conducted a comprehensive study based on them to evaluate our AFChecker. We expect to conduct more experiments to further evaluate AFChecker, and we will make it our future work.

5. Related work

Our work closely relates to three research areas, namely, quality assurance for context-aware applications, program invariant inference, and fault ranking. We discuss some representative pieces of work from recent years.

Quality assurance for context-aware applications. Context-aware applications have received increasing attention in recent years. Many powerful middlewares, such as CARISMA (Capra et al., 2003), SOCAM (Gu et al., 2004a), and EgoSpaces (Julien and Roman, 2006) support the construction of these applications. However, their quality assurance faces unique challenges that need special treatment (Cheng et al., 2009). First, these applications are continually driven by their perceived contexts. Due to the inevitable noise, context inconsistencies are common and should be detected in a timely fashion so as to avoid abnormal application behavior. Our previous work proposed techniques to efficiently detect context inconsistencies when perceived contexts violate consistency constraints, and resolve these inconsistencies automatically for guarding an application's quality (Xu and Cheung, 2005; Xu et al., 2010). Second, when enhanced with context consistency management services, an application may mix up the original contexts and their resolved counterparts, leading to unexpected data flow changes. Tse et al. (2004) first investigated this issue and reported its significance. Lu et al. (2006, 2008) later measured the impact of such changes, and proposed new testing adequacy criteria for addressing the dynamics caused by these data flow changes. Wang et al. (2007) studied similar data flow switching at context-aware program points, and presented new test generation techniques to cover such switching. These pieces of work share similar observations with our work by noticing the importance of the complex interactions between an application and its running environment. Third, even if an application's contexts are managed to be consistent, this application may still suffer from challenging faults. These faults manifest themselves when certain context-aware services are composed (Cubo et al., 2009), or result in an application crash or freezing at runtime (Xu et al., 2012). The former type of fault is statically identifiable from an application's adaptation model. To search for such faults, Sama et al. (2010a) proposed to use an adaptation finite-state machine (A-FSM) to model rule-based context-aware applications (also the target of this work). Their techniques extensively search the A-FSM model for potential faults (including many false positives as illustrated in this paper) by enumerative or symbolic algorithms. For convenience, they assumed the availability of global constraints for pruning those false positives in reported faults. However, we found that adequately and precisely inferring such constraints requires non-trivial efforts. Therefore, in this paper we presented a hybrid approach to extracting a domain and environment model represented by deterministic and probabilistic constraints. These constraints can be used to effectively improve the fault analysis quality (e.g., pruning false positives and ranking true positives with higher scores). The latter types of faults only manifest themselves into errors at runtime, and thus are beyond the detection capability of Sama et al.'s technique. Our latest work (Xu et al., 2012) systematically studied common types of such faults by inspecting various failure reports. For their detection, we proposed a technique that is able to precisely model a context-aware application's adaptation semantics and relate runtime errors to responsible faults.

Program invariant inference. Our constraint inference relates to program invariant inference techniques. Daikon (Ernst et al., 1999) is a popular dynamic invariant inference tool. It relies on the execution traces of a large test suite, and then infers likely invariants from these traces. However, these inferred invariants are typically not so useful for our problem, due to its limited built-in program invariant patterns. To address this problem, DySy, proposed by Csallner et al. (2008), combines concrete executions of test cases with their corresponding symbolic executions. Through this, DySy is able to produce abstract conditions about program variables that are satisfied in concrete executions. DySy can then significantly increase the relevance of its inferred program invariants, as well as reducing the number of test cases required for obtaining satisfactory invariants. Both the invariants inferred by Daikon or DySy seem to be correlations similar to those studied in this paper. However, they differ in the sense that our external correlations capture the hidden features of an application's physical environment, rather than those relationships internal to this application.

Fault ranking. Finally, our work presented a dynamic fault ranking strategy to prioritize reported faults. This is based on the observation that existing fault detection techniques, in particular static analysis, usually produce a long list of suspicious faults including numerous false positives (Kremenek and Engler, 2003). Fault ranking is thus necessary for helping users decide which faults to inspect first according to their probabilities of being true positives (or severity). To conduct effective fault ranking, Kremenek et al. (2004) observed that both true and false positives can cluster together due to code locality. They proposed exploiting this clustering behavior to improve fault ranking quality. For example, their proposed z-ranking technique (Kremenek and Engler, 2003) uses statistical analysis to cluster different groups of false positives, resulting in a static ranking of all faults. Later, they went on to explore a dynamic ranking strategy, where user's feedback is used to tune the original ranks for faults. This inspired our dynamic ranking strategy. However, our work differs from their work in one fundamental aspect. Their work requires a model (Bayesian network) to accurately capture the clustering behavior of reported faults while our work does not rely on the observation that faults tend to cluster.

6. Conclusion and future work

In this paper, we presented a hybrid approach toward effective adaptation fault detection for rule-based context-aware applications. We aim to support users to effectively inspect reported faults by removing most false positives and prioritizing the remaining ones. We achieved this by extracting a domain model and an environment model from user-configured adaptation rules and an application's running environment. We also adopted a dynamic ranking strategy to incorporate users' inspection feedback to further improve the fault ranking quality, and thus minimize users' efforts in fault inspection.

We implemented our work into a publically available tool, named AFChecker, and evaluated it through controlled experiments based on two popular applications. The results confirmed AFChecker's usefulness in improving user's fault detection and inspection experiences.

Our evaluation has been restricted to two selected applications. We plan to apply AFChecker to more applications and further validate its effectiveness. We also plan to extend our work to more types of context-aware applications, for example, those whose adaptation rules are not necessarily expressed by the syntax in Section 2.1. Such applications would be more flexible by allowing users to freely specify rules with quantifiers or domain-specific functions. Effective model checking techniques are required to address new challenges and assist fault detection tasks for these applications. Additionally, our work currently only considers those constraints built on binary relationships. Although, our evaluation results suggest that binary relations are effective enough in processing fault reports, we need to further investigate whether mining certain types of more complex constraints would better help detect faults. Finally, the ideal platform to deploy our work is a mobile device. Therefore, we plan to optimize our techniques to make them more cost-effective in terms of CPU usage and memory consumption, and we are already heading in this direction.

Acknowledgements

This research was partially funded by Research Grants Council (General Research Fund 612210, 612309) of Hong Kong, and by National Basic Research Program (973 program 2009CB320702), National High-Tech Research & Development Program (863 program 2011AA010103), and National Natural Science Foundation (61100038, 61021062) of China. Chang Xu was also partially supported by Program for New Century Excellent Talents in University, China (NCET-10-0486).

We thank Shauna Dalton for her careful proof-reading. We also thank anonymous reviewers for their valuable comments and suggestions on earlier versions of this paper.

References

Agrawal, R., Imieliński, T., Swami, A., 1993. Mining association rules between sets of items in large databases. In: Proceedings of ACM SIGMOD International Conference on Management of Data, pp. 207–216.

Alpaydin, E., 2010. Introduction to Machine Learning, 2nd ed. The MIT Press, Cambridge, Massachusetts/London, England.

Capra, L., Emmerich, W., Mascolo, C., 2003. CARISMA: context-aware reflective middleware system for mobile applications. IEEE Transaction on Software Engineering 29 (10), 929–945.

Cheng, B.H.C., Lemos, R.D., Giese, H., Inverardi, P., Magee, J., 2009. Software engineering for self-adaptive systems: a research roadmap, Software Engineering for Self-Adaptive Systems. Lecture Notes in Computer Science 5525, 1–26, http://www.springer.com/computer/swe/book/978-3-642-02160-2

Choco Solver, 2012. http://www.emn.fr/z-info/choco-solver/

Context Phone, 2005. http://www.cs.helsinki.fi/group/context/

Cover, T.M., Thomas, J.A., 2006. Elements of Information Theory, 2nd ed. John Wiley and Sons, Hoboken, New Jersey.

Csallner, C., Tillmann, N., Smaragdakis, Y., 2008. DySy: dynamic symbolic execution for invariant inference. In: Proceedings of the 30th International Conference on Software Engineering, pp. 281–290.

Cubo, J., Sama, M., Raimondi, F., Rosenblum, D., 2009. A model to design and verify context-aware adaptive service composition. In: Proceedings of International Conference on Services Computing, pp. 184–191.

Dittrich, K.R., Gatziu, S., Geppert, A., 1995. The active database management system manifesto: a rulebase of ADBMS features. Lecture Notes in Computer Science 985, 3–20.

Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D., 1999. Dynamically discovering likely program invariants to support program evolution. In: Proceedings of International Conference on Software Engineering, pp. 213–224.

Google ADC 2 Winners, 2009. http://code.google.com/android/adc/gallery_winners.html

Gu, T., Pung, H.K., Zhang, D.Q., 2004a. A middleware for building context-aware mobile services. IEEE Vehicular Technology Conference 5, 2656–2660.

Gu, T., Pung, H.K., Zhang, D.Q., 2004b. Toward an OSGi-based infrastructure for context-aware applications. IEEE Pervasive Computing 3 (4 (October)), 66–74.

Järvelin, K., Kekäläinen, J., 2002. Cumulated gain-based evaluation of IR techniques. ACM Transaction on Information Systems 20 (4 (October)), 422–446.

Julien, C., Roman, G.C., 2006. EgoSpaces: facilitating rapid development of context-aware mobile applications. IEEE Transaction Software Engineering 32 (5 (May)), 281–298.

Kremenek, T., Engler, D., 2003. Z-ranking: using statistical analysis to counter the impact of static analysis approximations. In: Proceedings of the 10th International Conference on Static Analysis, pp. 295–315.

Kremenek, T., Ashcraft, K., Yang, J., Engler, D., 2004. Correlation exploitation in error ranking. In: Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 83–93.

Locale, 2012. http://www.twofortyfouram.com/

Lu, H., Chan, W., Tse, T., 2006. Testing context-aware middleware-centric programs: a data flow approach and a RFID-based experimentation. In: Proceedings of the 14th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Portland, OR, USA, November, 2006, pp. 242–252.

Lu, H., Chan, W., Tse, T., 2008. Testing pervasive software in the presence of context inconsistency resolution services. In: Proceedings of the 30th International Conference on Software Engineering, Leipzig, Germany, pp. 61–70, May 2008.

Omnidroid, 2012. http://code.google.com/p/omnidroid/

PhoneAdapter, 2012. http://www.cse.ust.hk/~andrewust/phoneadapter.html

Rule Discussion, 2012a. http://tasker.wikidot. com/muteinmeetings

Rule Discussion, 2012b. http://tasker.dinglisch. net/tour.html

Rule Discussion, 2012c. http://tasker.wikidot.com/sleepmode

Rule Discussion, 2012d. http://groups. google.com/group/tasker/browse_thread/thread/6fa6398ec69f086c

Rule Discussion, 2012e. http://groups. google.com/group/tasker/browse_thread/thread/4ca7f88bf8bb0ae1

Sama, M., Rosenblum, D.S., Wang, Z., Elbaum, S., 2008. Model-based fault detection in context-aware adaptive applications. In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 261–271.

Sama, M., Elbaum, S., Raimondi, F., Rosenblum, D.S., Wang, Z., 2010a. Context-aware adaptive applications: fault patterns and their automated identification. IEEE Transactions on Software Engineering 36 (5 (September/October)), 644–661.

Sama, M., Rosenblum, D.S., Wang, Z., Elbaum, S., 2010b. Multi-layer faults in the architectures of mobile, context-aware adaptive applications. The Journal of Systems and Software 83 (2010), 906–914.

SweetDreams, 2012. https://market.android.com/details?id=com.inizz

Tasker, 2012. http://tasker.dinglisch.net

Tasker Limitations, 2012. http://tasker.dinglisch.net/bugs.html

Tse, T.H., Yau, S.S., Chan, W.K., Lu, H., Chen, T.Y., 2004. Testing context-sensitive middleware-based software applications. In: Proceedings of the 28th Annual International Computer Software and Applications Conference, vol. 1, pp. 458–466.

Wang, Z., Elbaum, S., Rosenblum, D.S., 2007. Automated generation of context-aware tests. In: Proceedings of the 29th International Conference on Software Engineering, Minneapolis, MN, USA, May, 2007, pp. 406–415.

Weiser, M., 1999. The computer for the 21st century. SIGMOBILE Mobile Computing Communications Review, 3–11.

Xu, C., Cheung, S.C., 2005. Inconsistency detection and resolution for context-aware middleware support. In: Proceedings of the Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Lisbon, Portugal, September, 2005, pp. 336–345.

Xu, C., Cheung, S.C., Lo, C., Leung, K.C., 2004. Cabot: on the ontology for the middleware support of context-aware pervasive applications. In: IFIP Workshop on Building Intelligent Sensor Networks.

Xu, C., Cheung, S.C., Chan, W.K., Ye, C., 2010. Partial constraint checking for context consistency in pervasive computing. ACM Transactions on Software Engineering and Methodology 19 (3 (January)), 1–61, Article 9.

Xu, C., Cheung, S.C., Ma, X., Cao, C., Lu, J., 2012. ADAM: identifying defects in context-aware adaptation. The Journal of Systems and Software 85 (12), 2812–2828 (Dec).

Yepang Liu is a Ph.D. student with the Department of Computer Science and Engineering of the Hong Kong University of Science and Technology (HKUST). He received his BSc degree in computer science and technology from Nanjing University. His research interests include software engineering, software analysis, and mobile computing.

Chang Xu is an associate professor with the State Key Laboratory for Novel Software Technology and Department of Computer Science and Technology of Nanjing University, China. He received his Ph.D. degree in computer science and engineering from the Hong Kong University of Science and Technology (HKUST). His research interests include software engineering, software testing and analysis, and pervasive computing.

S.C. Cheung received his BEng degree in Electrical Engineering from the University of Hong Kong, and his Ph.D. degree in Computing from the Imperial College London. After graduation, he joined the Hong Kong University of Science and Technology (HKUST) where he is a full professor and the director of RFID Center. He was an editorial board member of the IEEE Transactions on Software Engineering (TSE, 2006–9). He is the General Chair of the 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2014). His research interests include program analysis, testing and debugging, mobile computing, cloud computing, cyber-physical systems, end-user programming, and software analytics.