The Journal of Systems and Software 120 (2016) 70–88

Contents lists available at ScienceDirect

The Journal of Systems and Software

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

New Trends and Ideas

SIT: Sampling-based interactive testing for self-adaptive apps

Yi Qin[a,b], Chang Xu[a,b,*], Ping Yu[a,b], Jian Lu[a,b]

[a] State Key Laboratory for Novel Software Technology, Nanjing University, China
[b] Department of Computer Science and Technology, Nanjing University, China

ARTICLE INFO

Article history:
Received 7 September 2015
Revised 4 June 2016
Accepted 3 July 2016
Available online 7 July 2016

Keywords:
Self-adaptive application testing
Interactive application model
Sample-based testing

ABSTRACT

Self-adaptive applications ("apps" for short) are useful but error-prone. This stems from developers' inadequate consideration of environmental dynamics and uncertainty. Two features of self-adaptive apps, *infinite reaction loop* and *uncertain interaction*, bring additional challenges to software testing and make existing approaches ineffective. In this article, we propose a novel approach SIT (Sample-based Interactive Testing) to testing self-adaptive apps effectively and in a light-weight way. Our key insight is that a self-adaptive app's input space can be systematically split, adaptively explored, and mapped to the testing of the app's different behavior. This is achieved by our approach's two components, an interactive app model and a test generation technique. The former captures characteristics of interactions between an app and its environment, and the latter uses adaptive sampling to explore an app's input space and test its behavior. We experimentally evaluated our approach with real-world self-adaptive apps. The experimental results reported that our SIT improved the bug detection by 22.4–42.2%, but with a smaller time cost. Besides, SIT is also scalable with our tailored optimization techniques.

1. Introduction

Self-adaptive applications ("apps" for short) are gaining increasing attention. Their programs contain adaptation logic, which decide ways of delivering services based on varying environmental conditions (McKinley et al., 2004). Such apps typically run on embedded systems or smartphone platforms, using their sensors to collect environmental conditions. Different from traditional programs, a self-adaptive app involves a closed interaction loop that connects the app to its running environment, in which the app senses environmental changes, makes decisions based on its adaptation logic and performs adaptation to cope with the sensed changes. Examples of such self-adaptive apps include robot- car (Yang et al., 2014), which controls an intelligent automatic car to explore an unknown area, phone-adaptor (Sama et al., 2008)[1], which adapts the working mode of a smartphone according to its sensed changes in its user contexts, and Locale,[2] which provides location-based services (e.g., reminding book-returning near library or meeting schedules in office) automatically.

While self-adaptive apps offer flexible functionalities, they have to address complexities incurred by environmental dynamics and uncertainty. Empirical evidence shows that building self-adaptive apps is challenging and they are easily error-prone (Kulkarni and Tripathi, 2010; Sama et al., 2010; Xu et al., 2012). Regarding this, we identify two major causes:

Infinite reaction loop

A self-adaptive app executes in an infinite reaction loop with its environment (Brun et al., 2009). This execution model differs from those of many traditional apps, which take finite values (as input) and return results (as output). A self-adaptive app's reaction loop incurs an infinite series of input/output pairs, and this forms a large state space, in which the app's behavior can hardly be adequately tested. Besides, due to physical constraints, this series of input/output pairs has inherent correlations on their values. For example, a specific output (say, action of moving a robot-car forward) changes an environment's status, which then affects the concerned app's next input (say, shortening the distance between the car and its facing obstacle). As such, testing self-adaptive apps is non-trivial in that it has to take environment into consideration. In this article, we use *iteration* to denote one pass in executing an app's reaction loop (i.e., sensing-decision-adaptation).

* Corresponding author at: Department of Computer Science and Technology, Nanjing University, China.
E-mail addresses: qy.ics@smail.nju.edu.cn (Y. Qin), changxu@nju.edu.cn (C. Xu), yuping@nju.edu.cn (P. Yu), lj@nju.edu.cn (J. Lu).

[1] http://sccpu2.cse.ust.hk/afchecker/phoneadapter.html.
[2] http://www.twofortyfouram.com.

http://dx.doi.org/10.1016/j.jss.2016.07.002
0164-1212/

Uncertain interaction

A self-adaptive app also interacts with its environment in an uncertain way (Cheng, 2009). This refers to both *internal uncertainty* (difficulty of deciding the impact of adaptation on an app's goal realization) and *external uncertainty* (difficulty of deciding the error in measuring environmental conditions) (Esfahani et al., 2011). While the former concerns more on adaptation algorithms, we in this article focus on the latter, which affects how a self-adaptive app understands its environment and takes normal or abnormal behavior.

These two causes together decide the differences between the bugs in self-adaptive apps and those in traditional programs. While a traditional program's bugs depend mostly on the program itself (e.g., developers' mistakes in writing programs), a self-adaptive app's bugs would rely on three factors, namely, the app (program) itself, its running environment, and the uncertainty that affects app-environment interactions. First, the app may itself have implementation defects in dealing with app-environment interactions. Second, the app's running environment follows certain physical constraints and this may trigger program bugs in a way independent of the app itself. Finally, uncertainty in app-environment interactions can also cause the app's execution to deviate from its supposed path, leading to unexpected failure.

The uniqueness of bugs in self-adaptive app contributes to the challenges in effectively testing self-adaptive apps. An app can fail after accumulating multiple executions of its reaction loop with its environment. These executions are different iterations of the same program loop but with different input/output pairs. Testing such an app has to connect all these iterations, and this forms a large space, in which failure can be easily missed. Uncertainty further worsens the testing practice by making the input/output pairs deviate from their ideal values. For example, a robot-car can have its sensed distance contain unpredicted error, and its turning-direction action can also be imprecise due to uncontrollable rub with the ground. Such uncertainty can cause new problems to testing self-adaptive apps. For example, if the car senses its environment once per second and it runs for only one minute, then its app's execution can contain up to 2^{60} possibilities, even if uncertainty causes only two values to each input. Searching for a bug in such a large space is difficult. Thus the testing has to be extremely efficient to be useful.

Existing work proposed various random testing techniques to address the large space problem for self-adaptive apps, e.g., by considering metamorphic relations (Tse et al., 2004), context-switching points (Wang et al., 2007) and adverse environmental conditions (Ramirez et al., 2011). While they improve the testing efficiency, a systematic exploration of an app's space is not guaranteed. Moreover, the uncertainty issue is also overlooked. Our later evaluation reports that random testing could miss 39.2–64.2% bugs in self-adaptive apps.

Some other work focuses on guided testing for systematic exploration of an app's space. Dynamic symbolic execution (DSE) or concolic testing (Godefroid et al., 2005; Sen et al., 2005) and their variants are typical examples. When applied for testing self-adaptive apps, these techniques are also inadequate due to their low-efficiency caused by constraint solving. Moreover, physical constraints from environment (named *environmental constraints*) are usually, but such constraints are usually missing, and this causes solving-based techniques unable to explore an app's space completely. Our later evaluation reports that DSE could miss 25–40.5% bugs in self-adaptive apps. Besides, DSE itself is very time-consuming and it could explore very limited reaction interactions given the same time budget.

Self-adaptive app resembles mobile app in that both of them relate the environment. However, our work is essentially testing self-adaptive apps that interact with their environments, which can act in an uncertain way, rather than testing mobile apps without considering their environments and uncertainty explicitly (although such apps can also be our subjects in some situations as shown in our evaluation). Testing self-adaptive apps differs from testing mobile apps. For example, the former needs to consider an app's environment for a complete interaction-and-adaptation loop and this loop can go for multiple iterations, while the latter focuses mostly on an app itself. In other words, when considering sensing inputs from multiple iterations between an app and its environment, testing a mobile app seldom considers the inherent constraints of the sequences of sensing inputs of the app. This might cause problems to testing self-adaptive apps that interact with their environments, if one uses the same testing approach. For example, our later evaluation reports that bugs detected for self-adaptive apps by a mobile app testing approach (Liang et al., 2014) can contain up to 78.5% false positives.

In this article, we propose a novel approach SIT (Sample-based Interactive Testing) for testing self-adaptive apps. It consists of an interactive app model and a test generation technique. The model captures the characteristics of interactions between an app and its environment, and the technique uses adaptive sampling to systematically explore an app's space. They together contribute to our SIT approach's effectiveness. On one hand, our interactive app model considers the impact of environmental constraints and uncertainty on an app's input/output pairs, and enables a systematic exploration of its space. This exploration is guided, as compared to random testing. On the other hand, our test generation technique is light-weight by only sampling those inputs required by the exploration. Besides, the sampling does not rely on constraint solving, as compared to DSE. Therefore, it is highly efficient.

We experimentally evaluated our SIT approach with real-world self-adaptive apps and compared it with existing work. The experimental results consistently showed our SIT's effectiveness and efficiency. In particular, SIT improved the bug detection by 22.4–42.2% but with a smaller time cost. In summary, we make the following contributions in this article:

- Proposed an interactive app model for understanding interactions between an app and its running environment;
- Proposed a test generation technique for systematically sampling an app's space in a light-weight way;
- Conducted experiments to evaluate our approach with real-world self-adaptive apps.

The remainder of this article is organized as follows. Section 2 introduces our interactive app model for understanding how an app interacts with its environment, and presents a motivating example. Section 3 elaborates on our sampling-based test generation based on our interactive app model. Section 4 evaluates our SIT approach and compares it with existing work. Section 5 discusses related work, and finally Section 6 concludes this article and discusses future work.

2. Preliminaries

In this section, we introduce our interactive app model and present our motivating example based on this model.

2.1. Interactive app model

We propose an interactive app model (IAM) to explain how a self-adaptive app interacts with its environment under uncertainty. The model concerns not only the app itself, but also its interacting environment, as contrast to traditional app models, which typically concern apps themselves only.

Given a self-adaptive app, we define its IAM using a tuple (P, E, $inter_P$, $inter_E$, U, C). We use P to represent the app, and E to represent the environment where the app executes. We assume the availability of P's source code (i.e., white-box), but do not know how E works (i.e., black-box). Still, we can observe P's behavior in E (i.e., P's output) and P's obtained sensory data from E (i.e., P's input) through the testing interfaces of P and E, denoted by $inter_P$ and $inter_E$, respectively. Testing interfaces describe P's and E's variables whose values can be observed and manipulated as requested by a certain testing approach. We use U to represent an uncertainty specification, which describes the uncertainty affecting the interactions between P and E. Note that a complete description of uncertainty is typically infeasible, but we assume the availability of a partial specification (e.g., knowing a specific sensor variable's error range, which can be obtained from hardware specification or field tests). We use C to represent P's and E's initial configuration (i.e., default startup parameters for P and initial environmental layout for E). In the following, we elaborate on these elements.

App (P) and its testing interface $inter_P$

Given a self-adaptive app P, we define its testing interface using a tuple (I_P, O_P, G_P):

- I_P represents the app's input parameters. It is formed by a vector $\langle i_{P1}, i_{P2}, \ldots, i_{Pn}\rangle$, and each one i_{Pj} represents a specific sensor variable, which takes sensory data as its value from environment.
- O_P represents the app's output parameters. It is also formed by a vector $\langle o_{P1}, o_{P2}, \ldots, o_{Pm}\rangle$, which together explain what change is to be made to environment.
- G_P represents a set of variables defined in P that are shared across different iterations in P's execution. These variables are considered as *global variables* of P. Their values specify the app's current state.

Take our Robot-car app P for example. Its input parameters I_P is a vector containing three variables: ⟨*disF, disL, disR*⟩, which represent the car's sensed distances to its front, left-hand and right-hand obstacles, respectively. Its output parameters O_P is a vector of two variables: ⟨*actionType, actionPara*⟩, which represent the type of the action the car is to take (e.g.,"moving" or "turning") and its associated data (e.g., 3 cm for "moving" or +90 degrees for "turning"). The app's global variables G_P contains two variables, *task* and *past*, which represent the car's current task and recent input data (containing last five inputs for decision making), respectively.

Environment (E) and its testing interface inter_E

Our interactive app model also considers the environment E in which a self-adaptive app P executes. Conceptually, we consider the environment as a black-box program, which takes the app P's output as its input (i.e., applying the change the output is to make) and returns its output as P's input (i.e., returning sensory data to P). We further define E's testing interface $inter_E$ using a tuple (I_E, O_E, G_E):

- I_E represents the environment's input parameters. It is a vector $\langle i_{E1}, i_{E2}, \ldots, i_{Em}\rangle$, which corresponds to the app P's output $\langle o_{P1}, o_{P2}, \ldots, o_{Pm}\rangle$.
- O_E represents the environment's output parameters. It is also a vector $\langle o_{E1}, o_{E2}, \ldots, o_{En}\rangle$, which corresponds to the app P's input $\langle i_{P1}, i_{P2}, \ldots, i_{Pn}\rangle$.
- G_E represents a set of variables that describe environment E's status (e.g., environmental layout and object relationships). We assume these variables to be observable and resettable, thus facilitating our testing of the concerned app P.

For our Robot-car app P, its environment E's input parameters I_E contains two variables: *actionType* and *actionPara*, which are exactly P's output parameters (i.e., type of the action the car is to take and its associated data). E's output parameters O_E contains three variables: *disF, disL* and *disR*, which similarly correspond to P's input parameters (i.e., car's sensed distances to its front, left-hand and right-hand obstacles). G_E contains variables like *envCarLoc, envCarDir, envObjPro*$_1$, ..., *envObjPro*$_n$. They describe the car's current location, direction, and properties of obstacles (e.g., layout and boundaries) in the environment.

Uncertainty specification (U)

We define the uncertainty specification U as a set of functions, each of which maps a given environment's output parameter o_E to its corresponding app's input parameters as well as the associated error range (i_P, *lower, upper*). We assume the error range to be continuous within its lower and upper bounds. This simplification treatment applies to many real-world cases. Our model includes uncertainty since a self-adaptive app P interacts with its environment E under uncertainty in practice. If one does not consider uncertainty, P's input I_P would trivially equal to environment E's output O_E, i.e., $I_P = O_E$, on their values. In practice, $I_P \neq O_E$ (on values) due to uncertainty. Their differences are caused by unreliable environmental sensing (e.g., a sensed value deviates from its supposed value) or flawed physical actions (e.g., an action is taken without exactly achieving its supposed effect) (Ramirez et al., 2012). The aforementioned definition of U models such differences.

For our Robot-car app, its uncertainty specification U is defined as the following mappings:

$$O_E.disF \rightarrow (I_P.disF, -0.1, 0.1);$$
$$O_E.disL \rightarrow (I_P.disL, -0.1, 0.1);$$
$$O_E.disR \rightarrow (I_P.disR, -0.1, 0.1).$$

Configuration (C)

We use C to represent the initial configuration for app P and environment E. C contains initial values for variables in G_P and G_E, respectively. For our Robot-car app, C makes initial assignments to app P's global variables, and initializes environment E's layout and properties of its contained obstacles.

As a whole, our interactive app model IAM = (P, E, $inter_P$, $inter_E$, U, C) works in an iterative way, as illustrated in Fig. 1. It starts with app P and environment E initialized by configuration C through their testing interfaces(Label 1). App P's input I_P comes from environment E's output O_E. Then P executes based on I_P, updates its G_P, and returns output O_P by its testing interface $inter_P$ (Label 2). E takes O_P as its input I_E, "executes" by applying I_E's effect to update its G_E, and returns output O_E by its testing interface $inter_E$ (Label 3). This forms an iterative reaction loop. Since uncertainty specification U affects the interactions between $inter_P$ and $inter_E$, we conceptually represent this affection by $I_P = U(O_E)$, making I_P and O_E no longer simply identical (Label 4).

Our SIT approach assumes that the app under test should have source code (some existing work also has this assumption, e.g., Fredericks et al., 2013; Sama et al., 2008), but this assumption does not also go for the environment under test (i.e., the environment can simply be a black box). Besides, the source code of a self-adaptive app can be used to explain how a failure is triggered for this app, when it runs under a specific environment according to SIT's generated execution traces, as we explain later.

We also have two assumptions for the used environment in our SIT approach. First, we assume that the environment should be observable. This implies that one can monitor the values of its parameters for understanding its status during an app's execution. Some existing work (Ramirez et al., 2011; Fredericks et al., 2013)

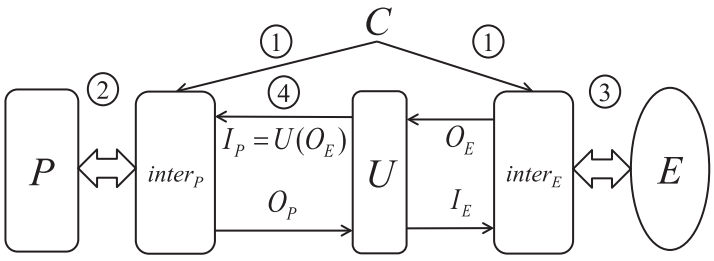


Fig. 1. IAM's iterative reaction loop.

on testing self-adaptive apps has similar assumptions. Besides, this assumption may also be supported by existing work (Perera et al., 2014; Lee et al., 2012). With this support, one is able to evaluate O_E and I_P, as required in our SIT approach. Second, we optionally assume that the environment can also be manipulated, i.e., the values of its parameters can be set or reset. This facilitates executing our SIT approach. If it is not supported, our approach still works but has reduced efficiency as the app under test needs to restart from the beginning each time.

Then the elements in an IAM model can be prepared as follows: (1) app P is directly available from the user, who plans to test P; (2) environent E is also available from the user, which could be an environment simulator, emulator or a real one (e.g., we used third-party simulators or emulators for experimental subjects in our later evalutation); (3) testing interface $inter_P$ is realized by instrumenting app P (concerned input and output parameters should be identified manually in advance), and testing interface $inter_E$ should be already ready if E is a simulator or emulator, or needs manual implementation if E is a real environment (the implementation is not complex as it is only for observing E's status but not controlling E); (4) uncertainty specification U is prepared manually from hardware specification or field tests (we checked sensor specifications for our experimental subjects and conducted trials for confirming error ranges for concerned sensors); (5) initial configuration C should be prepared manually, and some of its items need manual settings (e.g., the space boundary for the simulated environment for Robot-car in our experimental subjects), while others can be generated randomly (e.g., obstacles and their layout in the environment for running the car).

2.2. Motivating example

Let us consider an app that controls a robot-car to explore an unknown area based on its sensed distances to nearby obstacles. The car is required to keep some distance from obstacle in any direction for safety. Fig. 2 shows a code snippet of the app's program. The code snippet describes the app's adaptation logic when the car is too far away to its right-hand obstacle. Line 163 calculates a turning angle based on past sensory data. Line 168 makes the car take an "approaching the obstacle" decision by controlling the car to drive by leaning to its right-hand obstacle. When the app finds that the car is already keeping a safe distance from the obstacle, Line 171 turns the car again to make it drive in parallel to the obstacle and continue its exploration.

The shown code snippet can control the car to explore an unknown area correctly under ideal settings, i.e., when app-environment interactions do not suffer any uncertainty. However, it may have problems at the presence of uncertainty. In the scenario illustrated in Fig. 3, when the car drives to Position "A", the app plans to control the car to turn left by an angle in order not to crash into the obstacle wall, i.e., following the first dashed line D1. Line 171 calculates the car's turning angle for turning the car away from the obstacle, i.e., following the first dashed line D1. It uses a simple trigonometric calculation based on past sensory data (`past[4]` and `past[0]`). When considering uncertainty, all distance data contain error and the car may also not drive in an expected direction precisely. In some situations, uncertainty may just cause a smaller turning angle, and after turning the angle the car actually drives along the second dashed line D2. This direction, although not ideal, may not necessarily drive the car to crash into obstacle immediately. However, in an extreme situation, i.e., the difference between `past[4]` and `past[0]` is too large due to sensing error, the car might actually drive along the solid line D3, which causes the car to get too close to the wall or even crash into the wall, thus leading to a failure (physical crashing or safety assertion violated).

This example discloses the following testing challenges:

Large state space

The app takes sensory data as input from its environment, and their value combinations can be numerous. Besides, uncertainty blurs these values in a random way, and this further expands the app's state space. Software testing has to efficiently explore this large space to find potential execution traces leading to any failure.

Long execution trace

The app may iterate quite a few times before it executes to a failure, since a bug may manifest only after quite a few iterations. Software testing has to explore as many iterations as possible within its time budget, since a bug may manifest only after quite a few iterations.

The two challenges prevent existing work from effectively testing self-adaptive apps. For random testing, it treats an app as a black-box and tries to cover its space as much as possible. Consider our Robot-car app that has three input parameters. Assuming that each parameter has 100 different values, their combinations can be up to 100^3 even in a single iteration, not to mention when combined with uncertainty. Our later evaluation shows that random testing can hardly find execution traces leading to failures like the one in Fig. 3. For guided testing like DSE, it can explore all paths in one iteration systematically by solving constraints from these paths in turn. However, long execution trace makes DSE very ineffective. For our example, the failure manifests after Position "A". This accumulation requirement makes DSE have to take great effort to explore multiple iterations so as to reach that failure point. This can quickly drain limited time budget, not to mention that long execution trace can easily make constraint solving fail (i.e., timeout without any result). Our later evaluation shows that DSE can only explore first several iterations, missing most bugs like the one in Fig. 3.

The two challenges exhibit distinct requirements on testing self-adaptive apps. Large space requires systematic space exploration in order not to miss failure-inducing traces, and long execution trace calls for efficient space exploration in order not to drain time budget quickly. Random testing and DSE fail to satisfy at least one requirement, while our proposed SIT approach can meet both. Its interactive app model enables systematic space

```
 1.  public void main() // Main function
 2.  {
 3.     RobotCar P = new RobotCar();
        …
53.     while (true)
54.     {
55.        input = P.obtainData(); // Sense environment
           …
162.       if (P.task == ALONG_WALL && input.disR < 10){
163.          float angle = -(Math.abtan(((past[0].getR() -
                  past[4].getR()) / 5) / Math.PI * 180) +15);
164.          act = new Action ("turning-direction", angle);
165.          P.task = ADJUST;
166.          P.updatePast(input);
167.       } else if (P.task == ADJUST && !input.disR > 16){
168.          act = new Action ("moving", 2);
169.          P.updatePast(input);
170.       } else if (P.task == ADJUST && input.disR > 16){
171.          float angle = Math.atan(((past[0].getR() -
                  past[4].getR()) / 5) / Math.PI * 180);
172.          act = new Action ("turning-direction", angle);
173.          P.task = ALONG_WALL;
174.          P.updatePast(input);
175.       } else if …
              …
403.          P.output(act); // Make adaptation (take action)
404.       }
        …
512.  }
      …
```

Fig. 2. A code snippet for an example self-adaptive app.

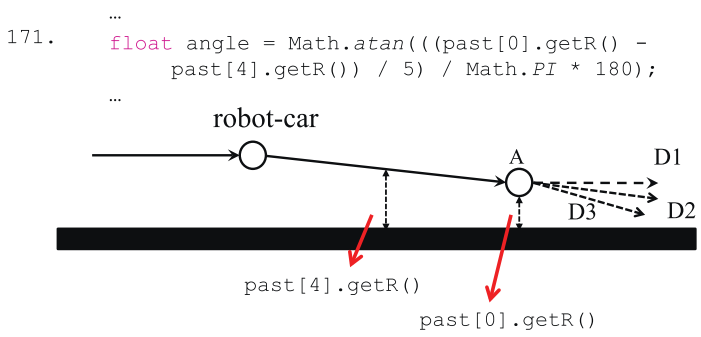


Fig. 3. A bug manifested by uncertainty in the Robot-car app's program.

exploration, and its test generation allows light-weight sampling-based testing. They together contribute to effective testing of self-adaptive apps, as we explain in the following.

3. Sampling-based interactive testing

In this section, we present our SIT approach for testing self-adaptive apps suffering uncertainty.

3.1. Overview

Our SIT contains an interactive app model IAM, as introduced in Section 2, and a test generation technique based on this model. Given a self-adaptive app P's IAM model, SIT would return a set of sequences of value assignments to $inter_P$'s input parameters I_P. Each sequence specifies a series of inputs to P, which runs with the inputs and eventually fails in its execution. The i-th value assignment in a sequence assigns values to P's input in its i-th iteration. To facilitate our discussions, we introduce two concepts, input space and input space tree, below.

Input space (IS)

An input space IS specifies ranges of possible values for input parameters to P, or more precisely, $IS = r_1 \times r_2 \times \ldots \times r_n$, where r_i specifies a range of possible values for P's i-th input parameter according to uncertainty specification U. We represent an input space as a vector of intervals $\langle [i_{P1}low, i_{P1}up], [i_{P2}low, i_{P2}up], \ldots, [i_{Pn}low, i_{Pn}up] \rangle$, in which the i-th interval specifies the range of values for P's i-th input parameter.

Input space is due to uncertainty, which changes input values for app P from deterministic values into non-deterministic values in a range. Considering that our targeted self-adaptive apps use sensors to collect environmental conditions, their input parameters take real numbers as values. Since each input parameter's value can vary in its error range, the whole input space becomes a continuous real-number space. For our Robot-car example, if the ideal

distance from the car to its front/left/right obstacle is 5, 10 and 7 respectively, then the app's input space at this moment is ⟨[4.9, 5.1], [9.9, 10.1], [6.9, 7.1]⟩, assuming uncertainty to be $[-0.1, +0.1]$ for all input parameters.

Input space tree (IST)

An input space tree gives the hierarchical structure for a set of input spaces, which provide input values for different iterations in a self-adaptive app's execution. In such a tree, if an input space is the parent node of another input space, it implies that the latter is derived from an iteration whose execution input is from the former. The root of an input space tree is always the initial input space, which is determined by the initial configuration C in an app's IAM model.

We use such a tree structure to model the relationships between input spaces for different iterations in an app's execution. This follows the intuition that an app's input values in one iteration can be affected by its input values in its past iterations. An app's input space tree would keep spanning when it takes more iterations in its execution. This is because in each iteration the app can possibly take different input values from the input space particular for this iteration. Thus, the input space tree models how our SIT approach explores an app's space. For our Robot-car example, a possible child node for the preceding input ⟨[4.9, 5.1], [9.9, 10.1], [6.9, 7.1]⟩ can be a new input space ⟨[3.9, 4.1], [9.9, 10.1], [6.9, 7.1]⟩, if the car drives one unit of distance ahead in the current iteration.

To find failure-inducing sequences of inputs, our SIT systematically explores the input space tree for an app in a breadth-first search (BFS) manner, i.e., SIT does not proceed for the input space of P's i-th iteration until it has explored all input spaces of P's $(i-1)$-th iteration. When app P starts, its testing interface $inter_P$'s initial values for input parameters I_P come from its first environmental sensing, i.e., O_E's value from E's testing interface $inter_E$. Due to uncertainty, even if O_E's value is deterministic, I_P's value can vary in its corresponding input space. We use U to derive the space and it is the root node of P's input space tree, which is to be first explored by SIT.

To explore an input space, SIT does not try all possible values in this space. Instead, it only samples some of them and make these sampled ones representative in terms of exercising an app's different behavior (sampling details discussed later in Section 3.3). For the first iteration, SIT explores an app's initial input space to check whether any specific input i_P in this space can lead to a failure. If yes, a failure-inducing sequence (containing only one input in this case) is found. Otherwise, for each sampled input i_P, since it does not lead to any failure, the app's execution would return a corresponding output o_P. SIT feeds each such output o_P to environment E as its input i_E and observes E's output o_E. Similarly, due to uncertainty, this output does not equal to app P's input in the next iteration. Instead, it corresponds to a new input space. The number of new input spaces in the next iteration equals to that of sampled inputs in the current iteration. Then a new round of the input space exploration starts, except that this time SIT has to explore more than one input space, and its constructed failure-inducing sequences can contain more than one input (for multiple iterations). This process repeats until a failure is encountered, if one aims to detect the first failure, or all time budget drains out (e.g., timeout), if one aims to detect as many failures as possible.

3.2. SIT Framework

Different from traditional programs, self-adaptive apps typically execute the same reaction loop for multiple iterations. Our intuition is to divide the task of testing a self-adaptive app's execution into that of testing its multiple iterations. We use an *abstract trace* with multiple *trace segments* to record an IAM's execution.

Abstract trace (AT) and trace segment (SEG)

We use *abstract trace* to represent an IAM's execution. An abstract trace is a sequence of trace segments $\langle at^1, at^2, \ldots, at^n \rangle$, where each trace segment records the execution information for one iteration of the concerned IAM. For ease of presentation, we use at^0 to represent the IAM's state before its first iteration. A *trace segment* is a tuple at^i: (i_P, o_P, G_P, G_E, is), in which $at^i.i_P$ and $at^i.o_P$ represent app P's input and output, respectively, in its i-th iteration. $at^i.G_P$ and $at^i.G_E$ represent values of P's and E's global variables after finishing the i-th iteration, respectively. $at^i.is$ represents app P's input space for its next iteration, which is from P's interaction with environment E in this iteration.

As mentioned earlier, we use G_P and G_E to represent the state of an IAM during its execution. In practice, all of app P's global variables are put into G_P, and we instrument P to record G_P during the IAM's execution. For G_E, its contained global variables depend on a given environment's observability. If it is an environmental simulator or emulator, we use its controlling interfaces to decide those parameters related to environmental layout and object relationships, and add them into G_E. If it is a real environment, we need to specify a set of elements that can monitor and quantify app P's operation conditions in environment. Then G_E includes those variables that can monitor these specified elements. Such monitoring can be accomplished by passively monitoring app P's sensing input (e.g., for surrounding environmental layout) or actively monitoring the environment via additional infrastructures (e.g., for precise location of a driving car). This process may need manual support, as also suggested by existing work (Fredericks et al., 2014).

Algorithm 1 gives our SIT approach's framework. It tries to detect as many failures as possible and uses BFS for space exploration. Set *FAIL* collects failure-inducing sequences, each of which is a series of inputs to app P and each input specifies values to

Algorithm 1 SIT framework

Input:
IAM $M := (P, E, inter_P, inter_E, U, C)$.
Output:
A set of failure-inducing sequences *FAIL*.

```
1: FAIL := ∅;
2: let is^0 be the initial input space of P;
3: IS := {is^0};   // All input spaces to explore
4: at^0 := ⟨ ( , , G_P^0, G_E^0, is^0) ⟩;   // G_P^0 and G_E^0 are from C
5: AT := {at^0};   // All abstract traces collected so far
6: while time budge allows do
7:   while IS ≠ ∅ do   // Execute app by sampling
8:     is := removeFrom(IS);   // BFS exploration
9:     let at be from AT such that at[LAST].is = is;
10:    (SEG_fail, SEG_succ) := sampling(M, is, at[LAST].G_P);
11:    FAIL := FAIL ⋃ appendInput(at, SEG_fail);
12:    AT := AT \ {at} ⋃ appendSeg(at, SEG_succ);
13:  end while
14:  AT' := ∅;   // Prepare for new abstract traces
15:  while AT ≠ ∅ do   // Interact with uncertain environment
16:    at := removeFrom(AT);
17:    (is, G_E) := interact(M, at[LAST].o_P, at[SEC_LAST].G_E);
18:    at[LAST].is := is;
19:    at[LAST].G_E := G_E;
20:    AT' := AT' ∪ {at};
21:    IS := IS ∪ {is};
22:  end while
23:  AT := AT';   // New abstract traces ready
24: end while
25: return FAIL;
```

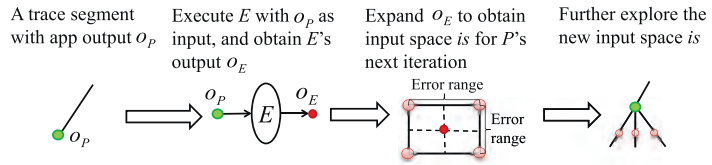


Fig. 4. IAM-based app/environment interaction.

P's input parameters in one iteration. is^0 is the initial input space for exploring (Line 2), as explained earlier. at^0 is the state before the first iteration, in which G_P^0 and G_E^0 are initial values of variables in P and E from initial configuration C (Line 4). After initialization (Lines 1–5), SIT executes app P with generated tests in two steps (Lines 6–24). First, SIT uses sampling to explore each available input space in the current iteration (Lines 7–13). The exploration results in two sets, SEG_{fail} and SEG_{succ}, which collect failing and passing executions, respectively, for the current iteration (Line 10). SEG_{fail} goes to *FAIL* for failure-inducing sequences since sampled inputs have caused failure (Line 11). SEG_{succ} extends the current abstract traces with new trace segments since app P successfully executes to its output (Line 12). Second, SIT makes app P interact with its environment E for each of the current abstract traces (BFS has multiple executions for each iteration; Lines 14–23). For app P's output from each abstract trace, the interaction between P and environment E leads to P's new input (recall $O_P \Rightarrow I_E \Rightarrow O_E \Rightarrow I_P$). Due to uncertainty in the interaction, each value i_P grows into a space *is* as P's input in its next iteration (Line 17). When values of the last trace segments in the current abstract traces are all ready, this process goes back to the beginning to start the next round of iteration. It repeats until all time budget drains out. If one aims to detect the first failure only, the algorithm can terminate once the first failure is detected at Line 11.

In an ideal environment where uncertainty does not exist, we have $I_P = O_E$ and $I_E = O_P$, on values, which make the method `interact` at Lines 17 quite simple. When uncertainty exists, we have $I_P = O_E \pm \triangle_E$ and $I_E = O_P \pm \triangle_P$ (conceptually). Here, we use $\triangle$ to denote the error introduced by the uncertainty specification. As such, a specific app's input i_P should consider this error in practice. We represent this by growing i_P into an input space, each dimension of which extends from one point to a range as specified by the uncertainty specification U defined in the IAM. This treatment applies to many real-world cases. Even if an actual error range may not necessarily be continuous, this treatment avoids missing potential failure-inducing inputs. Fig. 4 illustrates this interaction process. Another issue concerning method `interact` is that how one derives O_E based on O_P. If E can be manipulated, we directly set E to at[SEC_LAST].G_E, the environment's state by the end of the last iteration, and execute E on O_P to derive O_E. If E cannot be manipulated, we have to reset E according to the initial configuration C, and execute E from the beginning using the abstract trace.

3.3. Sampling-based test generation

Here we explain more about how tests are generated. Our SIT explores app P's input space *is* through systematic sampling of *is*, rather than trying all possible values in *is*. Its goal is to try those sampled inputs only such that they lead to different executions in P. This "difference" can be easily judged by their execution traces. Fig. 5 illustrates this sampling process. SIT first samples all vertexes at boundaries of input space *is*, and obtains corresponding execution traces by taking these vertexes as inputs (each vertex/input leads to one execution trace). Based on the similarity of the sampled execution traces, SIT decides whether to split the current space for further exploration. Each splitting brings multiple smaller input spaces, whose number relies on the dimension of the original input space. This splitting and exploration process repeats until an input space is sufficiently small or its vertexes do not lead to different executions for app P. Among all sampled executions, failing ones go to SEC_{fail} and passing ones go to SEC_{succ} for further processing as in Algorithm 1.

More precisely, SIT measures the similarity of execution traces from an input space's all boundary vertexes and decides whether to further split this input space for new exploration. It first derives the set of input values from the current input space's all boundary vertexes. Then it executes P with these input values for one iteration in turn, and records their execution traces, which are basically taken branches in the executions. After that, the similarity among these execution traces can be measured based on the overlapping of their taken branches. We assign each branch a unique *id* and calculate a hash value based on the *id*s of all taken branches in an execution trace. By this hash value, one can quickly distinguish different execution traces. Besides, similarity can also be calculated for a set of hash values from the aforementioned executions. For example, given an input space with four boundary vertexes (i.e., a two-dimensional space), these vertexes correspond to four inputs. We feed the four inputs to P, observe its execution traces, and obtain four corresponding hash values. The similarity among these four hash values decides whether one needs to further split this input space. Currently, we apply a simple strategy, i.e., considering the hash values the same or different only. We use a threshold for this strategy, which is set to one. It trivially implies that the input space should be further split if any hash value differs from the others.

When we decide to further split an input space, it is split in half at each dimension, i.e., selecting a midpoint between two vertexes of each dimension. Thus we obtain 2^n smaller input spaces from the original one, where n is the space's dimension. This seems to grow exponentially, but many of the input spaces do not have to be explored. We discuss their optimizations later in Section 3.4. Besides, we stop splitting an input space if it has been sufficiently small. We name this "sufficiently small" criterion *space-splitting threshold*. The setting of this threshold value needs to balance our SIT approach's testing coverage and efficiency. A smaller threshold value enables one to sample an input space more precisely to find failing executions for an app. However, it also incurs more time cost in exploring each input space, and results in less explored input spaces that SIT can explore within a given time budget. On the other hand, a larger threshold value enables one to sample an input space more efficiently, and thus SIT can explore more input spaces within the same time budget. However, one may miss few failing executions. In this work, we set the threshold to be 1/16 of error ranges from the uncertainty specification U to balance SIT's effectiveness and efficiency. We also investigate the impact of different threshold values on SIT's effectiveness and efficiency in the later evaluation.

Our sampling-based test generation was inspired by existing work on white-box sampling (Bao et al., 2012). The idea works for testing of self-adaptive apps due to the following two observations. First, a self-adaptive app P's input parameters take values from sensors, which typically report real numbers in continuous ranges. This enables us to split an input space and obtain meaningful samples as P's inputs. Second, an app's adaptation logic typically relies

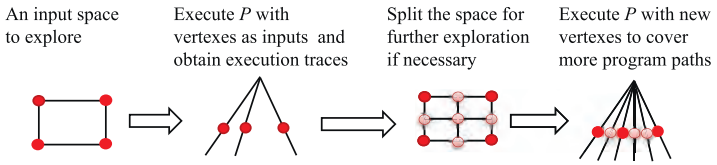


Fig. 5. Sampling-based test generation.

on a value range for its input to cope with uncertainty in its environmental interaction, rather than on specific value points. For example, PhoneAdaptor changes its profile when its collected GPS position falls in ranges like "home" or "office", rather than a specific position at home or in office[2]. These two observations concern a self-adaptive app's input and output, respectively, and our SIT exploits them to systematically split an input space and test an app's different behavior.

Algorithm 2 gives our sampling-based test generation. At the beginning, it explores only one input space (Line 3), but later it may explore more if the space is further split. In each exploration, the algorithm investigates one input space, and checks whether its vertexes (as app *P*'s inputs) lead to any failure, which is recorded (Line 11). All execution traces in one exploration are calculated for their hash values (Line 15), which, after a similarity and space size check (Line 17), decide whether to further split the current input space (Line 18).

3.4. Optimizations

As mentioned, sampling for an input space can lead to this space being split, when the samples from the space drive the app being tested to behave differently. Although we control the limit of splitting an input space into smaller ones, the growth of their number can be quick. Therefore, we introduce optimizations in our SIT approach to alleviating this issue.

Algorithm 2 Sampling-based test generation

Input:
IAM $M := (P, E, inter_P, inter_E, U, C)$, input space *is*, and G_P.
Output:
Two sets of trace segments SEC_{succ} and SEC_{fail}.

```
1:  SEC_succ := ∅;
2:  SEC_fail := ∅;
3:  IS := {is^0};   // All input spaces to explore
4:  while IS ≠ ∅ do
5:    is := removeFrom(IS);
6:    H := ∅;   // Storing hash values for execution traces
7:    for each vertex i_P of is do
8:      set(P, G_P);   // Set P's global variables to G_P
9:      (seg, branches) := execute(P, i_P);   // Execute P with input i_P
10:     if P fails then
11:         SEC_fail := SEC_fail ⋃ {seg};
12:     else
13:         SEC_succ := SEC_succ ⋃ {seg};
14:     end if
15:     H := H ⋃ {hash(branches)};
16:   end for
17:   if !similar(H) && !tooSmaller(is) then
18:     IS := IS ⋃ split(is);
19:   end if
20: end while
21: return (SEC_fail, SEC_succ);
```

In our sampling-based test generation, vertexes for testing in one iteration and input spaces for exploring across different iterations can both be redundant. We optimize them by pruning.

Fig. 6(a) illustrates the first type of redundancy, *vertex retest redundancy*. It occurs when an input space is split into smaller ones in one iteration. In Fig. 6(a), input space is_0 is split into four smaller ones, is_1, ..., is_4. For is_0, SIT has tested its four vertexes, v_1, v_2, v_3, v_4. When exploring is_1, SIT tests its four vertexes, v_1, v_5, v_6, v_9, in which v_1 is retested. When exploring is_2, SIT tests v_5, v_2, v_9, v_7, in which v_5, v_2, v_9 are retested. Such retesting is redundant. We use a coordinate-based mechanism to name different input spaces and cache hash values for tested vertexes. This enables us to test each vertex at most once.

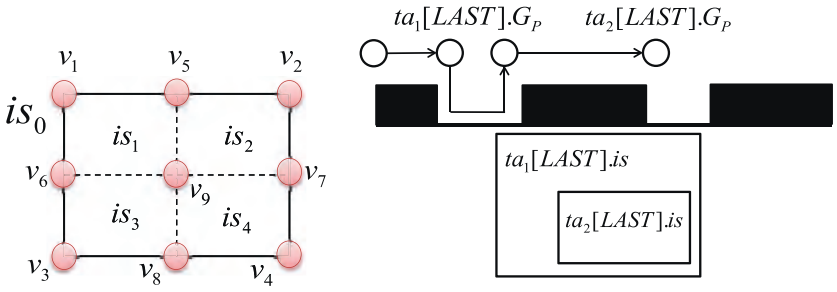
Fig. 6(b) illustrates the second type of redundancy, *space subsumption redundancy*. It occurs when two abstract traces from two iterations have their app's global variables G_P values equal but one's input space subsumes the other's (considering their last trace segments only). Then exploring the latter's input space is redundant. Here, G_P refers to those global variables in app *P*, as mentioned earlier when we introduce our IAM model. We use G_P only (i.e., no need to compare G_E) as an input space already carries sufficient information for environment *E*. G_P values are derived based on the instrumentation made for app P, as mentioned earlier. We define two sets of global variables having "equal" values as follows: for real-number variables (e.g., float or double types), their value differences should be less than a reasonable small threshold, say 0.01; for other variables (e.g., integer, string or char types), their value should be identical or their `equals` methods return true. With this definition, ta_1[LAST].G_P and ta_2[LAST].G_P in Fig. 6(b) can be equal, since their *task*'s values are both "TURNING_RIGHT", and *past*'s values are very close to each other due to similar environmental conditions.

4. Evaluation

In this section, we discuss the evaluation of our SIT approach with real-world self-adaptive apps. Section 4.1 introduces the implementation of our SIT approach. Section 4.2 presents our research questions for study, as well as variables and metrics for measuring these variables derived from the questions. Section 4.3 introduces the selected subjects. Section 4.4 discusses issues relating to our preparations for experiments. Section 4.5 elaborates on our experimental design and procedure. Section 4.6 presents and analyzes the experimental results, as well as answering earlier raised research questions. Finally, Section 4.7 discusses the threats to the validity of our experiments.

4.1. SIT implementation

We implemented our SIT approach in Java 8. It consists of 4 packages, 34 classes and 8700 LOC (lines of code) without comments. Each of the packages represents a module of our SIT approach, which includes `exploring`, `sampling`, `interaction` and `recording`. The first two modules implemented our SIT framework, as shown in Algorithms 1 and 2. Module `interaction` drives the target app and its environment by



(a) Vertex retest redundancy (b) Space subsumption redundancy

Fig. 6. Two types of redundancy in sampling.

following our IAM model. It also provides interfaces for observing and controlling the app and environment. Module `recording` records the execution trace of the IAM and analyzes it for testing support.

4.2. Research questions

Research questions and variables

We aim to answer the following research questions:

RQ1: *How does SIT compare to random testing (RT), dynamic symbolic execution (DSE), and caiipa (a mobile app testing tool Liang et al., 2014) for effectiveness and efficiency in testing self-adaptive apps?*

RQ2: *How do the settings of SIT, e.g., uncertainty level and space-splitting threshold, affect the effectiveness and efficiency of SIT in testing self-adaptive apps?*

RQ3: *Do our optimizations help on SIT's testing efficiency, and in what way?*

We compare SIT with RT and DSE since we believe that they are good representatives for unguided and guided testing approaches. We also chose caiipa to compare its effectiveness in detecting bugs with SIT, since caiipa, as a mobile app testing tool (Caiipa, 2014), also considers the impact of environment in generating its test inputs.

The independent variables of our evaluation include the specific approach used for self-adaptive app testing (i.e., SIT, RT, DSE or caiipa), uncertainty level (i.e., the scale rate of error ranges specified by uncertainty specification U, varying from 20–100% of original error ranges), space-splitting threshold (for deciding when to stop splitting an input space, ranging from 1/4 to 1/64), and applied optimization setting (i.e., reducing vertex retest redundancy only, reducing space subsumption redundancy only, reducing both, or without any optimization). The dependent variables include the *effectiveness* and *efficiency* in testing self-adaptive apps.

We measure effectiveness by the number of detected bugs and branch coverage. We measure efficiency indirectly by comparing the number of detected bugs given the same time budget for different testing approaches for RQ1 and RQ2. For RQ3, we measure efficiency by comparing the time of exploring the same iterations in executing experimental apps for SIT with different optimization settings.

Metric explanations

We further explain two metrics in measuring the effectiveness, namely, detected bugs and branch coverage. For detected bugs, we note that manifesting a bug in a self-adaptive app differs from that in a traditional program. Such a bug has to manifest in a certain environment in which a self-adaptive app runs for a couple of iterations. Therefore, the environment's configuration plays an important role in the bug detection. Recall that we use initial configuration for this purpose as defined in IAM. Thus in our experiments, we consider an app P and an initial configuration C (for initializing P and its associated E) as a *test instance*. We define *failure* as an app P violating any of its assertions under its initial configuration C within a given time budget. If a testing approach can detect any failure for a test instance (P, C) within the time budget, it is said to detect the *bug* associated with this test instance.

For branch coverage, we define it as follows, considering that a self-adaptive app can execute the same code snippet many times (by multiple iterations). Let B_i be the set of visited branches during the i-th iteration in app P's execution, and B be the set of all branches of P. Then for an execution that contains n iterations, its branch coverage is defined as $|\bigcup_{1\le i\le n} B_i|/|B|$, i.e., all visited branches are accumulated for an execution in calculating the branch coverage metric.

4.3. Experimental subjects

We selected three real-world self-adaptive apps as our experimental subjects. They run on different hardware platforms and undertake different adaptive tasks, including automated driving-control (Robot-car Yang et al., 2014), runtime profile-switching (PhoneAdaptor[2]) and city-wide navigation (SECONDO, 2009).

The Robot-car app is the motivating example discussed earlier. The app for experiments contains a full set of functionalities such as environmental sensing, map drawing and collision avoidance. It has been under development over five years in our university, and participated in various research activities (Yang et al., 2014; Xu et al., 2012; 2013). The app contains 3100 LOC.

The PhoneAdaptor app was originally proposed by Sama et al. (2008) for illustrating common bug patterns in adaptive apps, and later implemented by L iu et al.[2] for public access. It runs on Android phones, and automatically adapts a phone's profile (e.g., ring mode and vibration status) according to sensed environmental conditions and user-configured rules. The app uses various built-in phone sensors, e.g., GPS, Bluetooth and accelerometer. The app contains 1400 LOC.

The SECONDO app simulates traffic conditions in large-scale cities for scientific experiments (SECONDO, 2009). Mobile users navigate in a city, Berlin in our evaluation, to find places of interest or track moving objects (e.g., a driving bus or walking person). The app was developed by a research team at Fernuniversitat Hagen, and has participated in quite a few research projects (Xu and Guting, 2012; Duntgen et al., 2009). The app contains 38,000 LOC.

4.4. Experimental preparation

We explain our preparations for the experiments below.

4.4.1. IAM preparation

Each experimental subject app has a corresponding IAM model, and the model includes, besides the app P itself, also an environment E, initial configuration C and uncertainty specification U, as explained earlier in Section 2.1. We further explain how we prepared the IAM models for the three apps as follows.

The Robot-car app

The app came along with an accompanying environmental simulator and we used it as its running environment E. The app's initial configuration C specifies the space where the concerned car drives. We manually set the space boundary for the simulated space. Then random obstacles and their layouts were automatically generated within the specified boundary for the space. Besides, a pair of initial position and direction as startup parameters for the car were also randomly generated as part of C. The app's uncertainty specification U was from the car's field test results (Yang et al., 2014), containing error ranges for its installed ultrasonic sensors.

The PhoneAdaptor app

We setup up the PhoneAdaptor app by strictly following its original specification given by Sama et al. (2008), which includes seven states and 18 adaptation rules. The app's environment E consists of a smartphone emulator AVD (Android Virtual Device) and other supporting software modules. E is able to feed simulated sensory data to the app through its host phone according to how the environment is built and how the user walks in this environment.

For initial configuration C, we manually set the space boundary for E where the app's host phone stays. Then random blocks and their locations were automatically generated inside E. We also randomly generated a pair of start point and end point for the phone's simulated user. More precisely, these initial settings of the environment served as constraints for specifying functions and contexts for blocks (e.g., home, meeting room or exercise field) in the environment. These constraints were used to generate corresponding sensor inputs when the user walked from the start point to the end point in this simulated environment. The app's host phone is supposed to switch its profile based on changing environmental conditions and user pre-configured rules (e.g., enabling Bluetooth mode when entering a car or switching off ring tone when entering a meeting room). The app's uncertainty specification U mostly concerns GPS sensor, and its error range was set according to existing studies (Toftkjær and Kjærgaard, 2012; Shang et al., 2003).

The SECONDO app

We made the SECONDO app run in the simulated Berlin city as mentioned, and we used its benchmark toolkit MWGen as its environment. It could automatically build and customize a simulated city of Berlin and its residents according to preset controlling parameters. It could also observe and manipulate a simulated visitor in its built Berlin city for testing purposes.

The app's initial configuration C specifies the whole street network of the city, which contains more than 3000 streets, 4000 buildings, 80 bus routes and 10 metro routes. We also used MWGen to automatically generate a pair of start point and destination point for a visitor (taking a car, taxi, or simply walking) to this city, who is supposed to travel from the start point to the destination point. MWGen generated the travel plan based on its analysis on real-world people for their traveling activities. The visitor is supposed to reach the destination, by following the app's suggested navigation. The app obtains the visitor's GPS location every 500 m for optimizing its navigation suggestions. The app's uncertainty specification U also concerns GPS sensing and was set similarly as in the PhoneAdaptor app. Due to GPS noise, a suggested route may deviate from the optimal route and may lead the user to enter forbidden places.

4.4.2. Other preparations

Subject instrumentation and assertion check

We instrumented the three experimental subjects for collecting their execution traces (e.g., taken branches, input and output values) as well as values of their global variables G_P. We used assertions embedded in the three experimental subjects for exposing failures. SIT and other testing approaches were adapted for being able to monitor whether any assertion in an app was violated when feeding test inputs to the app and observing its execution. For the Robot-car app, its assertions check whether its concerned car crashes into any obstacle and whether the car is too far from its right-hand obstacle. For the PhoneAdaptor app, its assertions check whether any well-known fault patterns (e.g., adaptation race or cycle Sama et al., 2008) exist. For the SECONDO app, its assertions check whether its visitor's current route is optimal and whether he has been led into any forbidden place.

We illustrate how the assertions work by explaining two assertions in the SECONDO app. One assertion records the length of the visitor's actually taken route as navigated by the app, and compares it with the length of a pre-calculated optimal route from the start point to the destination point in the city. When the two lengths are found to have a big difference, say 20%, the assertion is violated. The other assertion records a set of locations that represent forbidden locations for this city. If the visitor is found to appear in any of these locations, the assertion is violated.

Environmental preparation

We also modified the code for environmental simulators or emulators for collecting their execution traces (e.g., global variables, input and output values). While capturing the latter two is relatively easy, we collected values of specific parameters from the simulators or emulators as values of their corresponding environmental global variables G_E. For the Robot-car app, we collected values of the location and direction of the car and obstacle layout as for G_E. For the PhoneAdaptor app, we collected its user's spacial relationships between the preset blocks, e.g., inside home and outside a meeting room as for G_E. For the SECONDO app, we collected the user's GPS location and his used transportation way as for G_E.

Approach implementation

We implemented RT and DSE for comparison purposes. While the former is easy, we built the latter by adapting Comedy (Jin et al., 2015), a concolic debugging tool built on Java PathFinder (JPF, 2013). Our DSE implementation connects multiple iterations of an app's reaction loop and solves their path constraints. We replaced JPF's original constraint solver with Z3 (De Moura and Bjørner, 2008), which is considered to be state-of-the-art in better solving constraints of different data types.

We implemented a controller for running caiipa and connecting it to Monkey (2013) for testing the PhoneAdaptor app. We did not compare SIT's effectiveness with caiipa on the other two apps since they are not Android apps and cannot be supported by caiipa. We also implemented a tool to validate the bugs detected by caiipa.

4.5. Experimental design and procedure

Test instances

Besides the three experimental subjects, we used (MuJava, 2013) to generate 135 mutants from the original apps, 45 for each, after excluding those that do not compile. MuJava used its built-in mutation operators, like arithmetic operator replacement (AOR), conditional operator replacement (COR) and logical operator replacement (LOR), for mutation.

For each original app, we prepared 32 different initial configurations. Considering that different initial configurations may lead to different experimental results (e.g., detected bugs), we randomly generated these initial configurations. For each mutated app, we randomly selected four initial configurations from 32 ones for scale consideration. Thus, we obtained a total of 636 test instances for experiments, 96 test instances for original apps (3×32), which are denoted as test set *O*, and 540 test instances for mutated apps (135×4), which are denoted as test set *M*.

Experimental procedure

We conducted experiments with the 636 test instances to answer research questions, which took more than 600 hours. All experiments were conducted on a PC with an Intel(R) Core(TM) i7 CPU@4.5GHz and 8GB RAM. We first evaluated and compared the effectiveness and efficiency of SIT, RT, DSE and caiipa in testing self-adaptive apps, in which each test was given a 10-min time budget. Then we studied how different settings, e.g., uncertainty level and space-splitting threshold, affect SIT's testing effectiveness and efficiency based on the 96 test instances from the original apps. Finally, we compared SIT's testing efficiency with different optimization settings also based on the 96 test instances. In all experiments, whenever a test instance failed (app bug detected), we moved on to the next test instance. In other words, each test instance could fail at most once for each testing approach. In this sense, detected bugs are considered distinct since the same test instance will not fail twice for the same testing approach.

For RQ1, we compared detected bugs for SIT, RT and DSE using all 636 test instances. We also compared detected bugs for SIT and caiipa using the 212 test instances derived from the PhoneAdaptor app, which is the only app caiipa can support, as mentioned earlier. Since mutation might change an app's branches accidentally, we compared branch coverage for SIT, RT and DSE using only the 96 test instances from test set *O* (based on original apps). We then compared testing efficiency (by means of the number of detected bugs given the same time budget) for SIT, RT and DSE using all 636 test instances. Finally, we also compared SIT, RT and DSE's testing efficiency within a one-hour time budget using 50 randomly selected test instances from all 636 test instances, in order to validate our choice of the 10-min time budget.

For RQ2, we first studied the effect of different uncertainty levels. We compared detected bugs for SIT, RT and DSE with different uncertainty levels (i.e., scaling the applied error range to 20–100% of the original error ranges specified by uncertainty specification *U*) using the 96 test instances from test set *O* (based on original apps). Then we studied SIT's performance with different space-splitting threshold values, namely, 1/4, 1/8, 1/16, 1/32 and 1/64 of the error ranges from *U*. We compared detected bugs and testing efficiency (by means of the number of detected bugs given the same time budget) for SIT with different threshold values also based the 96 test instances from test set *O*.

For RQ3, we compared SIT's testing efficiency (by means of the time of exploring the same iterations in executing experimental apps) with different optimization settings using the 96 test instances from test set *O*. The optimization strategies includes no optimization, reducing vertex retest redundancy only, reducing space subsumption redundancy only, and reducing both redundancy types.

4.6. Experimental results and analyses

4.6.1. RQ1: Comparison of effectiveness and efficiency in testing self-adaptive apps

We first compare detected bugs and branch coverage for testing effectiveness, and then compare for testing efficiency.

Table 1
Comparison of detected bugs for SIT, RT and DSE.

Approach	All subjects (636 test instances)	Robot-car (212)	PhoneAdaptor (212)	SECONDO (212)
SIT	581 (91.4%)	184 (86.8%)	202 (95.2%)	195 (92.0%)
RT	313 (49.2%)	76 (35.8%)	108 (50.9%)	129 (60.8%)
DSE	432 (69.0%)	126 (59.5%)	147 (69.3%)	159 (75.0%)

Table 2
Comparison of bug distribution for SIT, RT and DSE.

Bug groups	All subjects	Robot-car	PhoneAdaptor	SECONDO
(+ , + , +)	242	62	83	97
(+ , + , -)	45	8	13	24
(+ , - , +)	160	56	54	50
(+ , - , -)	138	62	52	24
(- , + , +)	18	4	7	7
(- , + , -)	8	2	5	1
(- , - , +)	12	4	3	5

SIT's comparison with RT and DSE on detected bugs

Table 1 compares SIT, RT and DSE in their detected bugs with respect to all 636 test instances (212 for each subject). We observe that SIT consistently detected more bugs in these test instances. The improvement is 27.3–51.0%, 25.9–44.3% and 17.0–31.2% for the three subjects, respectively. Overall, SIT improved the detection rate by 22.4–42.2%. DSE performed better than RT as expected, since it used the same time budget to explore different program paths in apps. Besides, we observe that all approaches detected less bugs for the Robot-car app than the other two subjects. After a closer study, we found that the Robot-car app has relatively more difficult bugs. These bugs manifested after 15 iterations, while those for the PhoneAdaptor and SECONDO apps did after 5–10 iterations. Then given the same time budget, all approaches detected less bugs for the Robot-car app. Furthermore, we did not observe any bug detected in the first iteration. This exhibits how testing self-adaptive apps differs from, and is also more challenging than, testing of traditional programs.

The results also reveal the different nature of RT and DSE. In relatively shallower iterations (e.g., before the 5–*th* iteration), DSE detected more bugs than RT (93.4–128% more). This is because RT can easily miss many dedicated paths in apps due to its simple (random) strategy for generating test inputs while DSE uses a constraint solver to consider every possible path carefully. On the other hand, in relatively deeper iterations (e.g., after the 10–*th* iteration), RT detected more bugs than DSE (236–268% more). This is because DSE cannot go for too deep iterations when given the same time budget since constraint solving is computation-intensive while RT can easily go to those deep iterations.

Table 2 compares the distribution of the detected bugs for SIT, RT and DSE. We partition all detected bugs into seven groups, by whether they can be detected by a particular approach. This is annotated by a triple (*x, y, z*), in which *x* indicates "yes" by " + " or "no" by " − " for SIT, and *y* and *z* for RT and DSE in a similar way. Table data can tell the uniqueness of an approach in testing self-adaptive apps. In brief, most (over 95%) of the bugs detected by the other approaches could also be detected by SIT. On the other hand, altogether 138 bugs (23.6%) were detected *by and only by* SIT. The counterparts for RT and DSE are 8 and 12, respectively, which are much less (94.2% and 91.3% less). This suggests that our SIT is indeed effective in detecting bugs for self-adaptive apps and this effectiveness does not rely on certain initial configurations.

We also observe that both RT and DSE were indeed able to detect some bugs that could not be detected by SIT approach (6–12 or 2.8–5.7% bugs detected by RT, and 8–12 or 3.8–5.7% bugs de-

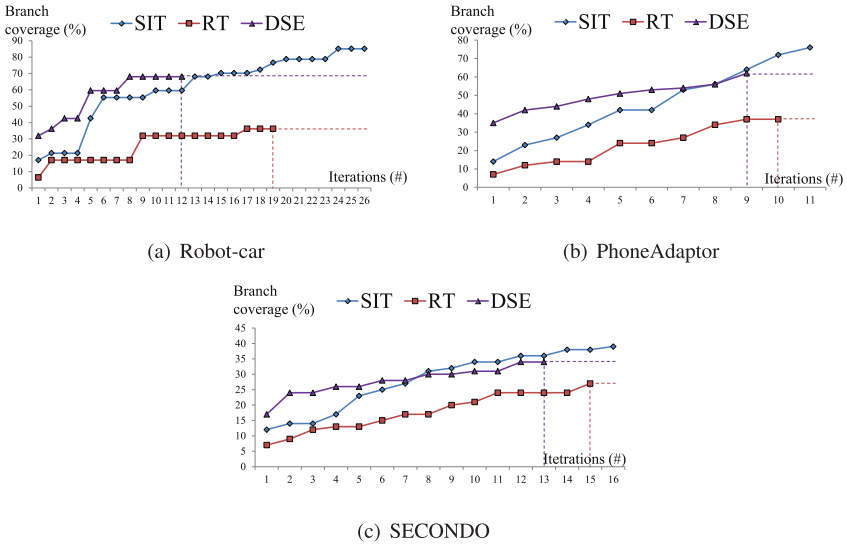


Fig. 7. Comparison of branch coverage for SIT, RT and DSE (with respect to different subjects).

tected by DSE). We analyzed these bugs. For the bugs detected by RT but not by SIT, we found that most of them (83.3–87.5%) were detected after the 10–*th* iteration in executing an app. For the bugs detected by DSE but not by SIT, most of them (70–87.5%) were detected before the 5–*th* iteration in executing an app. Such results have been caused by the different nature of RT and DSE. We further explain it below.

First, RT can go into relatively deeper iterations in executing an app when given the same time budget, due to its simple (random) strategy for generating test inputs. This explains why RT could detect some bugs SIT did not detect in relatively deeper iterations (e.g., after the 10–*th* iteration). However, this strategy also sacrificed RT's completeness in testing, as RT failed to detect many bugs (69.3%) in relatively shallower iterations (e.g., before the 5–*th* iteration) as compared to SIT. SIT does not suffer this limitation, because it tries to avoid exercising the same path in app execution by systematically exploring different partitions in an input space. Our experimental results (Fig. 7) show that SIT achieved higher branch coverage (12.3–47.9% higher) than RT.

Second, DSE considers every path in executing an app, and tries that path as long as it can generate a corresponding test input by constraint solving. As such, it can exercise more paths in relatively shallower iterations (e.g., before the 5–*th* iteration) in executing an app, when given the same time budget. This explains why DSE could detect some bugs SIT did not detect in relatively shallower iterations. However, as constraint solving is computation-intensive, DSE cannot go for too deep iterations when given the same time budget. Actually, DSE explored less iterations (22.2–116.7% less) than SIT when given the 10-min time budget. SIT does not suffer this limitation, because its test input generation is light-weighted. Our experimental results (Fig. 8) show that SIT detected bugs much faster than DSE. SIT detected 90% bugs (524 in number) it could detect in 300 s, while DSE cost 550 s (83.3% more time) for detecting 90% bugs (390 in number). If one fixes the time budge to be 300 s for DSE as well, it can detect only 117 bugs, which is much less than 524 bugs detected by SIT (77.7% less).

SIT's comparison with caiipa on detected bugs

Table 3 compares SIT and caiipa in their detected bugs with respect to the 212 test instances derived from the PhoneAdpator app. We observe that caiipa detected slightly more bugs (7 or 3.4% more) than SIT, but most of its detected bugs are false positives (78.5%). The reason for caiipa's high false positive rate is that it

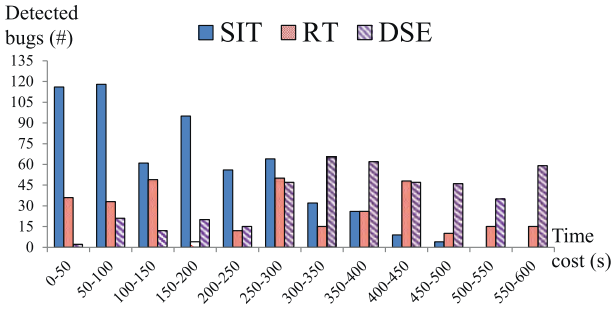


Fig. 8. Comparison of testing efficiency for SIT, RT and DSE (overall, 10-min time budget).

Table 3
Comparison of detected bugs for SIT and caiipa.

Approach	Detected bugs	False positives	Distribution of true bugs		
			(+, +)	(+, −)	(−, +)
SIT	202 (95.2%)	0	42	160	3
caiipa	209 (98.6%)	164 (78.5%)			

failed to consider the complete interaction and adaptation loop between the app and its environment. In other words, the relationships between I_P values across different iterations in generated test inputs were not respected. For example, caiipa might generate a sequence of failure-inducing test inputs indicating that the user's two consecutive GPS locations are too far away with each other for the user to reach in one iteration. In our experiments, we replayed sequences of failure-inducing test inputs from the bug reports of caiipa in our IAM model to check whether the app and its environment interact with each other as expected. In each iteration, we executed the IAM model to derive the app's input space *is* for the next iteration. If the I_P value of the next iteration (from the bug reports of caiipa) did not fit into the space *is*, then the corresponding bug was considered as a false positive.

SIT did not incur any false positive since it already validated all detected bugs through concrete execution of the app in its environment. Considering true positives only, SIT detected 339% more bugs than caiipa. We also analyzed the bug distribution for SIT and caiipa, as in Table 3. Apparently, most of the bugs detected by cai-

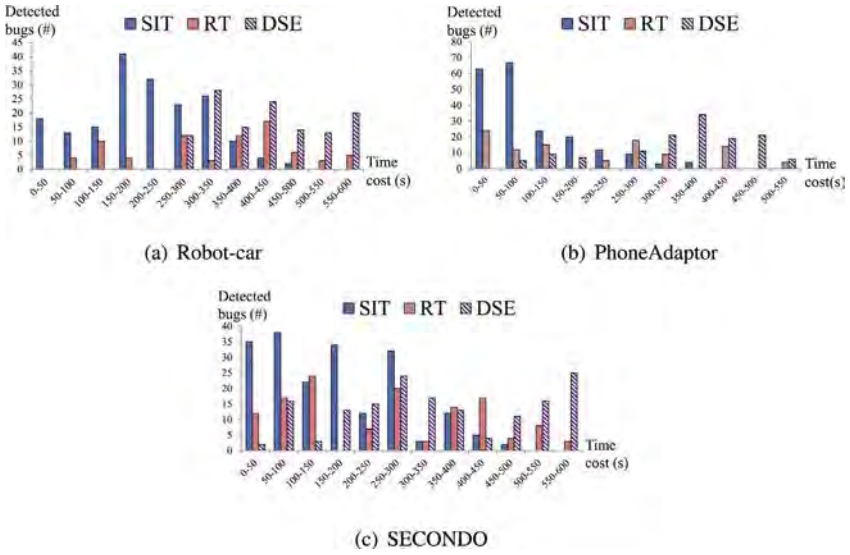


Fig. 9. Comparison of testing efficiency for SIT, RT and DSE (with respect to different subjects, 10-min time budge).

ipa can also be detected by SIT except only three ones. This shows our SIT's unique effectiveness in detecting bugs for self-adaptive apps, as compared to other approaches catered for testing mobile apps.

SIT's comparison with RT and DSE on branch coverage

Fig. 7 compares SIT, RT and DSE in branch coverage with respect to the 96 test instances from the original apps. Data were averaged for each subject. For each approach, its connected line segments illustrate achieved branch coverage from its first iteration to the last iteration it could reach within the given time budget. We observe that SIT achieved the highest branch coverage (12.3–47.9% higher than RT and 13.8–21.4% higher than DSE), as well as the most iterations (6.7–36.8% more than RT and 22.2–116.7% more than DSE). This explains why SIT detected more bugs than RT (31.2–54.3% more) and DSE (17–27.3% more) as reported earlier. It is easily understandable that RT achieved low coverage since its strategy was purely random in exploring input spaces, but DSE's low coverage needs more explanations. DSE tries to consider every path in executing an app, and exercises that path as long as it can generate a corresponding test input by constraint solving. However, this can be very time-consuming and cause it to fail to balance the testing's completeness and efficiency. The experimental results show that DSE achieved the highest branch coverage at the beginning since it worked in a guided way, but this strength was quickly gone when it stopped exploring more iterations within the given time budget. Beside, DSE also suffered failed constraint solving. For example, in the SECONDO app DSE's explored branch coverage was instead lower than that of our SIT even for the same number of iterations.

SIT's comparison with RT and DSE on testing efficiency

Fig. 8 compares SIT, RT and DSE on testing efficiency (by means of the number of detected bugs given the same time budget) with respect to all 636 test instances. Fig. 9 gives detailed results with respect to each subject. Overall, SIT detected bugs much faster than the other two approaches. It detected 90% bugs it could detect within all time budget in 300 s (50% budget) only, while the counterpart time is 500 s (83% budget) and 550 s (92% budget) for RT and DSE, respectively. Besides, RT is unstable in that it detected more bugs early and less later for the PhoneAdaptor and SECONDO apps, but things reversed for the Robot-car app. This reflects its inherent randomness. Regarding DSE, it seems to miss quite a lot

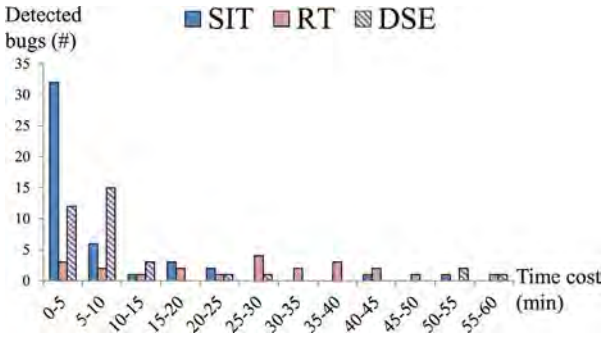


Fig. 10. Comparison of testing efficiency for SIT, RT and DSE (overall, one-hour time budget).

of bugs in the early stage, especially for the Robot-car app. We conjecture that this is because DSE spent too much time on constraint solving, and this made it detect much less bugs within first several minutes. That is, given the same time budget, while DSE was working on early iterations, SIT already worked on later iterations. As such, for the same time budget, SIT can detect bugs from more iterations than DSE. Besides, failed constraint solving also reduced the bugs that can be detected by DSE, while SIT does not require constraint solving at all. Altogether, we observe that our SIT is both light-weight and effective. We also notice that although the PhoneAdaptor app uses a non-manipulatable environment, it did not take SIT much longer time to test this app. This is because PhoneAdaptor's executions contain less iterations (usually less than 10), which mean restarting app executions does not bring much overhead.

Fig. 10 compares SIT, RT and DSE on testing efficiency (by means of the number of detected bugs given the same time budget) with a one-hour time budget for 50 test instances randomly selected from all 636 test instances. The result shows that most detected bugs by SIT and DSE were found in the first 10 min, e.g., 82.6% bugs for SIT and 77.1% bugs for DSE. This justifies our selection of 10 min as the time budget for most experiments (data of RT are not considered due to its random nature).

Answer to RQ1

Our SIT approach can test self-adaptive apps effectively and efficiently. Compared with existing approaches (RT, DSE and caiipa),

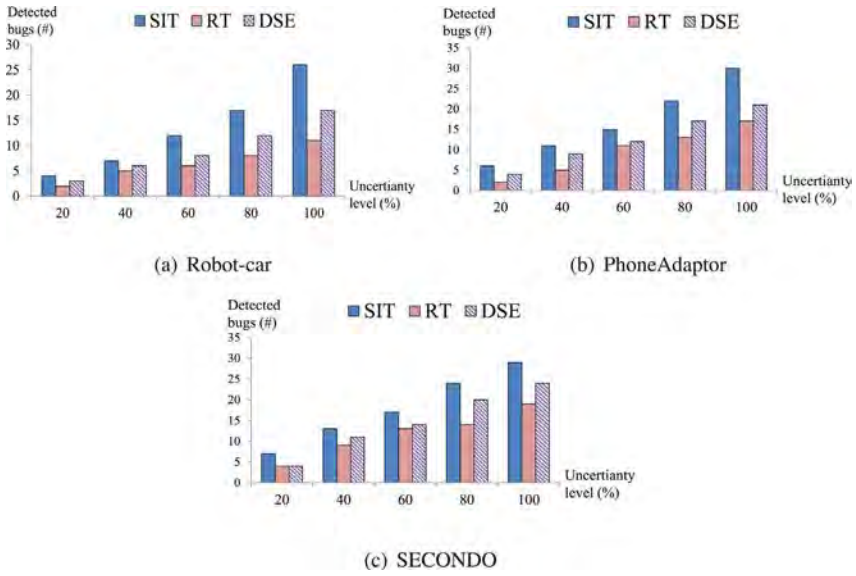


Fig. 11. Comparison of detected bugs for SIT, RT and DSE under different uncertainty levels (with respect to different subjects).

SIT can detect more bugs, as well as more unique bugs that cannot be detected by other approaches. It can also detect bugs more quickly by exploring more iterations and covering more branches.

4.6.2. RQ2: Study of impact of different settings on SIT

SIT's comparison with RT and DSE on detected bugs using different uncertainty levels. We studied how the uncertainty level affects the bug detection in terms of detected bugs for different testing approaches. Fig. 11 compares SIT, RT and DSE in their detected bugs under different uncertainty levels with respect to the 96 test instances from test set *O*. Data were averaged for each subject. We observe that for all testing approaches, the number of detected bugs consistently increases with the growth of uncertainty level. For example, comparing the setting of 100% uncertainty level to that of 20% uncertainty level, SIT detected 414–650% more bugs, RT detected 475–850% more bugs, and DSE detected 525–600% more bugs. This shows that uncertainty indeed plays an important role in causing self-adaptive apps to fail, and more severe uncertainty implies more failing cases, as well as more responsible program bugs (failing to consider such cases). We also observe that SIT detected always the most bugs under all uncertainty levels, showing its effectiveness in testing self-adaptive apps. Besides, the differences between SIT and the other two approaches increase with the growth of uncertainty level. This is probably because increasing the uncertainty level causes the growth of an app's state space, which then leads to more challenges for traditional testing approaches to detect bugs.

SIT's performance using different space-splitting threshold values on testing effectiveness. We also studied how the space-splitting threshold affects the effectiveness and efficiency of our SIT approach in testing self-adaptive apps. Table 4 compares SIT's detected bugs with different threshold values based on the 96 test instances from test set *O*. The column "Bugs" represents the number of detected bugs, column "Iter." represents the average number of iterations explored by SIT, and column "B.C." represents the branch coverage. The experimental results show that SIT detected bugs satisfactorily when the threshold was set to 1/16 of error ranges, and also roughly detected the most bugs (2–12 or 7.4–74.6% more) as compared to those under other threshold value settings. This can be explained by the fact that SIT achieved the highest branch coverage (1–29% higher than those under other settings) when the threshold value was set to 1/16 of error ranges.

Table 4 also compares the distribution of the detected bugs of SIT under different threshold value settings, in which column "U. Bugs" represents the number of unique bugs that can only be detected under one threshold value setting, and "C. Bugs" represents the number of common bugs that can be detected under at least three threshold value settings. The results tell that different threshold values indeed enable SIT to detect some unique bugs (3.1–12.5%) that cannot be detected under other threshold value settings. On the other hand, most of the detected bugs (62.5–93.6%) can be detected under at least three threshold value settings. This suggests that SIT's effectiveness in testing self-adaptive apps is not limited to specific space-splitting threshold value settings.

Fig. 12 compares testing efficiency (by means of the number of detected bugs given the same time budget) of SIT with different threshold value settings. We observe that a larger threshold value could enable SIT to detect bugs more quickly than a smaller threshold value when given the same time budget, since it helps SIT to exercise an app into deeper iterations. So large threshold values can help quickly answer whether an app can fail within a limited time budget, while small threshold values can be used to explore different failing executions precisely when the time budget is sufficient.

Answer to RQ2

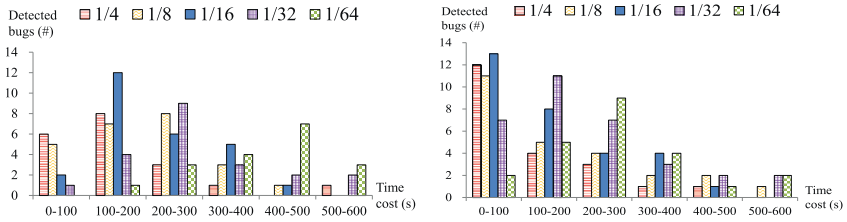
Regarding the uncertainty level, SIT can detect more bugs with the growth of uncertainty level associated with self-adaptive apps. SIT detected always the most bugs under all uncertainty levels, as compared with existing approaches (RT and DSE). Regarding the space-splitting threshold, SIT's testing effectiveness peaked when the threshold was set to 1/16 of error ranges, with which it can roughly detect the most bugs as compared to other threshold settings when given the same time budget. SIT's testing efficiency decreased with the growth of the threshold value, in which a larger threshold value could enable SIT to detect bugs more quickly than a smaller threshold value when given the same time budget.

4.6.3. RQ3: Study of impact of different optimization settings on SIT

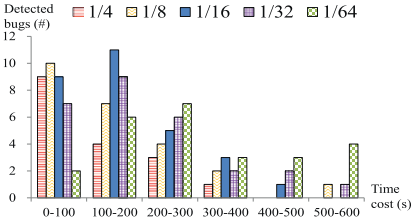
Our earlier comparisons were made based on SIT with all optimizations enabled. We finally study how the optimization setting helps on SIT's testing efficiency. Fig. 13 compares SIT's testing efficiency (by means of the time of exploring the same iterations in executing experimental apps) with different optimization settings

Table 4
Comparison of testing effectiveness for SIT under different space-splitting threshold values.

Threshold values	Robot-car					PhoneAdaptor				
	Bugs	U. Bugs	C. Bugs	Iter.	B.C.(%)	Bugs	U. Bugs	C. Bugs	Iter.	B.C.(%)
1/4	19	1	16	33	58	21	1	19	11	48
1/8	24	3	20	28	83	25	2	21	11	61
1/16	26	4	20	21	85	30	3	26	11	76
1/32	21	1	18	18	64	31	4	27	10	77
1/64	18	1	17	13	56	23	2	19	9	54
Threshold values	PhoneAdaptor									
	Bugs	U. Bugs	C. Bugs	Iter.	B.C.(%)					
1/4	17	1	15	24	26					
1/8	24	1	22	18	30					
1/16	29	3	24	16	39					
1/32	27	3	22	15	36					
1/64	25	1	24	12	30					

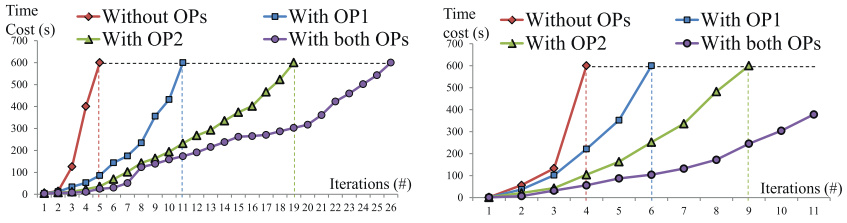


(a) Robot-car app (b) PhoneAdaptor app

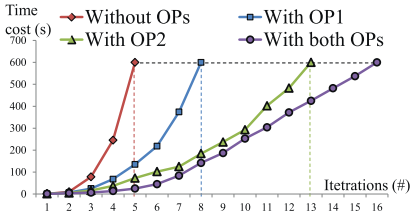


(c) SECONDO app

Fig. 12. Comparison of testing efficiency for SIT under different space-splitting threshold values (with respect to different subjects).



(a) Robot-car (b) PhoneAdaptor



(c) SECONDO

Fig. 13. Comparison of testing efficiency for SIT with different optimization settings.

for the 96 test instances from the original apps (data averaged for each subject). We considered four optimization settings, namely, disabling all optimizations (denoted as "Without OPs"), reducing vertex retest redundancy only (denoted as "With OP1"), reducing space subsumption redundancy only(denoted as "With OP2"), and reducing both redundancy (denoted as "With both OPs"). We observe that when disabling all optimizations, SIT could only explore up to 4–5 iterations before it drained out all time budget. The reduction of explored iterations ranged from 63.6% to 80.8% for different subjects, which is significant. Considering different optimizations, reducing space subsumption redundancy is more effective than reducing vertex retest redundancy, which enables SIT to explore 50.0–72.2% more iterations for different subjects. This is because vertex retest redundancy only causes a polynomial amplification on the number of explored input spaces, while space subsumption redundancy causes an exponential amplification. We note that for PhoneAdaptor, the "With both OPs" curve did not reach the limit of 600 s and this is because our optimized SIT finished all its test instances by then. Without optimizations, SIT spent time in an exponential way, while with optimizations, it was almost in a linear way. This makes our optimized SIT useful in practice.

Answer to RQ3

Our optimizations significantly increase SIT's efficiency and scalability in the bug detection for self-adaptive apps. Specifically, reducing space subsumption redundancy is more effective than reducing vertex retest redundancy.

4.7. Threats to validity

Internal threats

We set 10 min as the time budget for testing each test instance in experiments, and this might threaten the internal validity of our evaluation, causing different comparison conclusions. This setting is for scale-controlling since we had to compare three testing approaches with respect to 636 test instances, which are many. We note that we are comparing different approaches' bug detection capabilities under a limited time budget, which is a realistic setting in practice. Besides, 10 min suffices for this purpose as bugs detected in the first 10 min occupied about 80% bugs detected in one hour, as mentioned earlier. Furthermore, our experimental data show that SIT detected bugs clearly faster than RT and DSE, with increased branch coverage and more explored iterations. The three aspects together justify SIT's effectiveness. The likelihood of RT and DSE suddenly beating SIT with a more time budget is low.

External threats

Two threats might weaken the external validity of our evaluation. First, we conducted experiments with third-party simulators or emulators. This is for controlling purposes, e.g., time controlling as the Robot-car app needs recharging after 15-min driving, and parameter controlling as we could get and set values of environmental variables easily, as mentioned earlier. Nevertheless, our SIT approach does apply to real-world apps with some platform-specific support. For example, The PhoneAdaptor app can already run on real Android phones, since Android provides direct access to its resource status (e.g., ring mode, vibration level and Bluetooth switch). For the Robot-car app, the only required support is to reset the car's location to an earlier one if required. Currently, we do not have a mechanical infrastructure help on this (e.g., using another robot to move the car with its arm, which can be expensive and time-consuming). Although using a mechanical infrastructure can introduce error in resetting environmental variables, our SIT approach already considers this by allowing the modeling of error ranges in uncertainty specifications. This makes our SIT approach still useful for future extensions with such mechanical support.

Second, we selected only three experimental subjects and the number is not many. This is because a comprehensive evaluation requires the support of suitable environments, which should be observable and resettable. Although the selected subjects are not many, we tried to make them representative as real-world self-adaptive apps. These subjects cover different functionalities (e.g., automated driving, location-based service and navigation) and also use different prevailing platforms (e.g., ARM-based robot-car, Android-based smartphone and commodity PC). Therefore, our selected experimental subjects can represent real-world self-adaptive apps to some extent. Nevertheless, trying our SIT approach on more types of apps and platforms still deserves for better evaluation.

Theoretical reliability

It is possible that our implementation biased for our own SIT approach. To alleviate this threat, we made all approaches under test use a shared app/environment interaction module (`interaction`, as discussed earlier in Section 4.1), which feeds the output from one side (app/environment) as the input to the other side (environment/app), and drives the interaction continually. Each approach only implements its internal logic for deciding test inputs. We implemented RT ourselves as it is simple, and implemented DSE and caiipa based on available code from existing work (Caiipa, 2014; Jin et al., 2015).

5. Related work

In this section, we discuss related work on handling uncertainty for self-adaptive apps and testing for self-adaptive, context-aware, mobile or numeric programs.

Uncertainty in self-adaptive apps

Uncertainty causes challenges to quality assurance for self-adaptive apps. Ramirez et al. (2012) presented a taxonomy of uncertainty factors typical for self-adaptive apps. They include requirement uncertainty, design uncertainty and runtime uncertainty. Many pieces of work focus on alleviating impact of design uncertainty on self-adaptive apps. Ghezzi et al. (2013) proposed an adaptation framework to manifest non-functional uncertainty via model-based development. Famelis et al. (2012) used partial models to specify uncertainty and reason its impact on app functionalities. Esfahani (2011) proposed framework support for designing decision-making functions for self-adaptive apps, which helps avoid misguided behavior and subjective preferences. Different from them, we in this article focus on runtime uncertainty and check whether a self-adaptive app's implementation has considered uncertainty precisely and adequately.

Testing self-adaptive apps

Some pieces of work focus on testing techniques to ensure quality of self-adaptive or context-aware apps. Fredericks et al. (2014) used utility functions to guide the design and adaptation of test cases for self-adaptive apps. Xu et al. (2012) proposed monitoring error patterns to track responsible defects in context-aware adaptation. Tse et al. (2004) relied on metamorphic relations to decide whether contexts and their upper-layer apps behave abnormally. Ramirez et al. (2011) proposed discovering specific combinations of environmental conditions that produce violated behavior in adaptive systems. These pieces of work used different observations, but in general still relied on random testing. This implies that they do not guarantee systematic exploration of an app's

space. As a contrast, our SIT approach explores an app's input space and its corresponding behavior in a systematic and guided way.

Testing context-aware apps

Some pieces of work on testing context-aware apps also consider interaction between apps and their environments. Griebe and Gruhn (2014b) proposed a model-based testing approach that generates test cases using model transformation on context-enriched design-time system models. Amalfitano et al. (2013) used predefined event patterns to generate context event traces to explore different behaviors for context-aware apps. Jang et al. (2005) proposed a framework to simulate both context-data production and service execution for testing context-aware apps. The framework is also able to generate simulated context data based on user-specified operational components, such as virtual sensors, devices, and even human beings. Wang et al. (2007) improved test coverage for context-aware apps by utilizing context-switching points in apps. Lu et al. (2006) proposed a family of test adequacy criteria to cover new data flows induced by interactions between an app and its underlying middleware. They then extended the work to support testing apps with external inconsistency resolution services Lu et al. (2008), which call for new coverage criteria. These pieces of work echo our work in that both consider challenges from interactions between an app with its environment, but our work further considers uncertainty in the interactions, which include both environmental sensing and behavioral adaptation.

Testing mobile apps

Self-adaptive apps resemble some mobile apps in that such apps have to acquire inputs from environments. Liang et al. (2014) proposed contextual fuzzing to build a comprehensive library of contexts of different types for mobile apps and a learning-based technique to explore the context space for apps testing. Rege et al. (2015) used existing user traces to generate realistic and correlated context traces automatically for the propose of guiding simulator-based mobile apps testing. Griebe and Gruhn (2014a) gave a model-based approach to generating useful contextual inputs for mobile apps by deducing context information from design-time system models. These pieces of work differ from our work in that they treat environment as a simple source for an app's input, while the app's adaptation and reaction on the environment is largely simplified or overlooked. Our approach focuses on the many iterations between apps and their environments, and this implies that both apps and their environments can affect each other.

Other pieces of work focus on testing event-based mobile apps or GUI testing. Yeh et al. (2013) proposed an approach to analyzing GUI models during the testing process and generating corresponding event sequences for testing based on GUI models derived from mobile apps. Adamsen et al. (2015) presented an approach that leverages existing test cases such that each test case is systematically exposed to adverse conditions where certain unexpected events may interfere with the test execution. Anand et al. (2012) studied how to generate sequences of events automatically and systematically based on concolic testing. The approach alleviated the path-explosion problem by checking conditions from program executions to identify subsumption relationships between different event sequences. We believe that these approaches may not be directly applicable to testing self-adaptive apps since our targeted apps may not have GUIs (e.g., two of our experimental subjects do not contain any GUI, while the remaining one has a very simple GUI but that is not our testing focus; we are testing an app's interaction with its environment) and may not necessarily be event-based.

Testing numeric programs

Self-adaptive apps also resemble some numeric programs in that such apps often take sensory data as inputs, which range in a scope. Some pieces of work focused on testing numeric programs. For example, Chen et al. (2010) studied the diversity nature of adaptive random testing, which is useful for partitioning input space and testing numeric programs. Bao et al. (2012) proposed white-box sampling to test scientific computation programs with inputs of uncertainty. Chaudhuri et al. (2011) used static analysis to quantify a numeric program's robustness to inputs of uncertainty, by proving whether the program has encoded a functionality in a robust way. The test generation part of our work was inspired by white-box sampling, but we extended its idea to multi-dimensional sensory data for multiple iterations, and enhanced its effectiveness by taking into account sensing and adaptation uncertainty as well as optimizations for effective iteration exploration.

6. Conclusion and future work

In this article, we focus on testing self-adaptive apps. We analyzed characteristics of such apps and studied their challenges to software testing. We proposed a novel approach, named SIT, to testing self-adaptive apps in a systematic and light-weight way. Our experimental evaluation reported promising results, showing that SIT can detect more bugs by covering more code and exploring more iterations, but with smaller time cost, as compared with existing work.

Our work still has limitations. It currently relies on app-specific support if an app's running environment cannot be easily manipulated for testing. Still, it is understandable since such apps themselves also require additional hardware to run as platforms, and such support can be regarded as necessary components of platforms, not to mention that some platforms (e.g., Android system) already have such support.

Besides, the work also brings new research opportunities. First, we assume that input parameters to an app P should take values from continuous domains since SIT uses sampling to explore P's input space. Continuous input values enable one to split an input space precisely and obtain meaningful samples as P's input values. This setting applies to many real-world self-adaptive apps, which take sensory data as inputs, which are naturally continuous. However, this setting does not directly apply to apps with discrete input values. One reason is that dichotomy no longer works for splitting an input space of discrete input values.. Even if it can be alleviated by special treatment, a more serious reason is that app P's behavior may depend more likely on certain discrete input values for decision making. This makes P's behavior may no longer change gradually with the change in input values, and SIT might miss critical program paths in its exploration in testing P. One possible way is to first learn the relationships between the program paths P will take and the range of input values fed to P, and then use such relationships to guide the splitting of an input space. Clearly, this needs further research and validation, and we keep it as future work.

Second, the assertions we used to define program failures were composed manually in apps. Such assertions are closely related to the apps' specifications and only use observable variables of app P and environment E, i.e., the inputs and outputs of P and E. It might be possible to derive such assertions automatically for self-adaptive apps, as suggested by existing work, e.g., Zoom-In (Pastore and Mariani, 2015) and MuTest (Fraser and Zeller, 2010). One advantage of automatic assertion generation is that it requires less knowledge about targeted apps than manually writing assertions for them. Another advantage is that automatically-generated assertions could be defined on values of internal variables (i.e., not sensor variables)

in such apps, which could provide a more accurate monitoring of an app's internal status. We are also working along this line.

Acknowledgment

This work was supported in part by National Basic Research 973 Program (Grant no. 2015CB352202), and National Natural Science Foundation (Grant nos. 61472174, 91318301, 61321491) of China.

References

Adamsen, C.Q., Mezzetti, G., Møller, A., 2015. Systematic execution of android test suites in adverse conditions. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA '15, pp. 83–93.

Amalfitano, D., Fasolino, A.R., Tramontana, P., Amatucci, N., 2013. Considering context events in event-based testing of mobile applications. In: Proceedings of IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops, ICSTW' 13, pp. 126–133.

Anand, S., Naik, M., Harrold, M.J., Yang, H., 2012. Automated concolic testing of smartphone apps. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE '12, pp. 649–660.

Bao, T., Zheng, Y., Zhang, X., 2012. White box sampling in uncertain data processing enabled by program analysis. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '12, pp. 897–914.

Brun, Y., Serugendo, G.D.M., Gacek, C., Giese, H., Kienle, H., Litoiu, M., Muller, H., Pezze, M., Shaw, M., 2009. Engineering self-adaptive systems through feedback loops. In: Software eEgineering for Self-adaptive Systems. Springer, pp. 48–70.

Caiipa, 2014, http://research.microsoft.com/en-us/projects/contextual-fuzzing/.

Chaudhuri, S., Gulwani, S., Lublinerman, R., Navidpour, S., 2011. Proving programs robust. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ser. ESEC/FSE '11, pp. 102–112.

Chen, T.Y., Kuo, F.-C., Merkel, R.G., Tse, T.H., Jan. 2010. Adaptive random testing: the art of test case diversity. J. Syst. Softw. 83 (1), 60–66.

Cheng, B.H.e. a., 2009. Software engineering for self-adaptive systems. In: Software Engineering for Self-Adaptive Systems: A Research Roadmap. In: LNCS, vol. 5525, pp. 1–26.

De Moura, L., Bjørner, N., 2008. Z3: an efficient smt solver. In: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, ser. TACAS'08/ETAPS'08, pp. 337–340.

Duntgen, C., Behr, T., Guting, R.H., 2009. Berlinmod: a benchmark for moving object databases. VLDB J. 18 (6), 1335–1368.

Esfahani, N., 2011. A framework for managing uncertainty in self-adaptive software systems. In: Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, ser. ASE '11, pp. 646–650.

Esfahani, N., Kouroshfar, E., Malek, S., 2011. Taming uncertainty in selfadaptive software. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ser. ESEC/FSE '11, pp. 234–244.

Famelis, M., Salay, R., Chechik, M., 2012. Partial models: towards modeling and reasoning with uncertainty. In: Proceedings of the 34th International Conference on Software Engineering, ICSE '12, pp. 573–583.

Java Path Finder, 2013. http://babelfish.arc.nasa.gov/trac/jpf/.

Fraser, G., Zeller, A., 2010. Mutation-driven generation of unit tests and oracles. In: Proceedings of the 19th International Symposium on Software Testing and Analysis, ISSTA '10, pp. 147–158.

Fredericks, E.M., DeVries, B., Cheng, B.H.C., 2014. Towards runtime adaptation of test cases for self-adaptive systems in the face of uncertainty. In: Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self–Managing Systems, SEAMS '14, pp. 17–26.

Fredericks, E.M., Ramirez, A.J., Cheng, B.H.C., 2013. Towards run-time testing of dynamic adaptive systems. In: Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS '13, pp. 169–174.

Ghezzi, C., Pinto, L.S., Spoletini, P., Tamburrelli, G., 2013. Managing nonfunctional uncertainty via model-driven adaptivity. In: Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, pp. 33–42.

Godefroid, P., Klarlund, N., Sen, K., 2005. Dart: Directed automated random testing. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '05, pp. 213–223.

Griebe, T., Gruhn, V., 2014a. A model-based approach to test automation for context-aware mobile applications. In: Proceedings of the 29th Annual ACM Symposium on Applied Computing, SAC'14, pp. 420–427.

Griebe, T., Gruhn, V., 2014b. A model-based approach to test automation for context-aware mobile applications. In: Proceedings of the 29th Annual ACM Symposium on Applied Computing, SAC '14, pp. 420–427.

Jang, M., Kim, J., Sohn, J.-C., 2005. Simulation framework for testing context-aware ubiquitous applications. In: Proceedings of the 7th International Conference on Advanced Communication Technology, ICACT '05, pp. 1337–1340.

Jin, H., Jiang, Y., Liu, N., Xu, C., Ma, X., Lu, J., 2015. Concolic metamorphic debugging. In: Proceedings of the 2015 IEEE 39th Annual Computer Software and Applications Conference, COMPSAC '15.

Kulkarni, D., Tripathi, A., 2010. A framework for programming robust context-aware applications. IEEE Trans. Softw. Eng. 36 (2), 184–197.

Lee, Y., Ju, Y., Min, C., Yu, J., Song, J., 2012. Mobicon: Mobile context monitoring platform: Incorporating context-awareness to smartphone-centric personal sensor networks. In: Procedding of the 9th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks, SECON '12, pp. 109–111.

Liang, C.M., Lane, N.D., Brouwers, N., Zhang, L., Karlsson, B.F., Liu, H., Liu, Y., Tang, J., Shan, X., Chandra, R., Zhao, F., 2014. Caiipa: automated large-scale mobile app testing through contextual fuzzing. In: Proceedings of the 20th annual international conference on Mobile computing and networking, MobiCom-14, pp. 519–530.

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, ICSE '08, pp. 61–70.

Lu, H., Chan, W.K., Tse, T.H., 2006. Testing context-aware middlewarecentric programs: a data flow approach and an rfid-based experimentation. In: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT '06/FSE- 14, pp. 242–252.

McKinley, P.K., Sadjadi, S.M., Kasten, E.P., Cheng, B.H.C., 2004. Composing adaptive software. Computer 37 (7), 56–64.

Monkey, 2013http://developer.android.com/guide/developing/tools/monkey.html.

MuJava, 2013http://cs.gmu.edu/offutt/mujava/.

Pastore, F., Mariani, L., 2015. Zoomin: discovering failures by detecting wrong assertions. In: Proceedings of the 37th International Conference on Software Engineering, ICSE '15, pp. 66–76.

Perera, C., Zaslavsky, A., Christen, P., Georgakopoulos, D., 2014. Context aware computing for the internet of things: a survey. in Communications Surveys & Tutorials, IEEE 16 (1), 414–454.

Ramirez, A.J., Jensen, A.C., Cheng, B.H.C., 2012. A taxonomy of uncertainty for dynamically adaptive systems. In: Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS '12, pp. 99–108.

Ramirez, A.J., Jensen, A.C., Cheng, B.H.C., Knoester, D.B., 2011. Automatically exploring how uncertainty impacts behavior of dynamically adaptive systems. In: Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE '11, pp. 568–571.

Rege, M.R., Handziski, V., Wolisz, A., 2015. Poster: a context simulation harness for realistic mobile app testing. In: Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys'15, p. 489.

Sama, M., Elbaum, S., Raimondi, F., Rosenblum, D.S., Wang, Z., 2010. Context-aware adaptive applications: fault patterns and their automated identification. IEEE Trans. Softw. Eng. 36 (5), 644–661.

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, SIGSOFT '08/FSE-16, pp. 261–271.

Sen, K., Marinov, D., Agha, G., 2005. Cute: a concolic unit testing engine for C. In: Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE-13, pp. 263–272.

SECONDO, 2009. http://dna.fernuni-hagen.de/Secondo.html/index.html.

Shang, Y., Ruml, W., Zhang, Y., Fromherz, M.P.J., 2003. Localization from mere connectivity. In: Proceedings of the 4th ACM International Symposium on Mobile Ad Hoc Networking, MobiHoc '03, pp. 201–212.

Toftkjær, T., Kjærgaard, M.B., 2012. The impact of sensor errors and building structures on particle filter-based inertial positioning. Pervasive Mob. Comput. 8 (5), 764–776.

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, COMPSAC '04, vol. 01, 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, ICSE '07, pp. 406–415.

Xu, C., Cheung, S.C., Ma, X., Cao, C., Lu, J., 2012. Adam: identifying defects in context-aware adaptation. J. Syst. Softw. 85 (12), 2812–2828.

Xu, C., Yang, W., Ma, X., Cao, C., Lu, J., 2013. Environment rematching: toward dependability improvement for self-adaptive applications. In: Proceedings of the 28th ACM/IEEE International Conference on Automated Software Engineering, ASE '13, pp. 592–597.

Xu, J., Guting, R.H., 2012. Mwgen: A mini world generator. In: Proceedings of the 2012 IEEE 13th International Conference on Mobile Data Management, MDM '12, pp. 258–267.

Yang, W., Xu, C., Liu, Y., Cao, C., Ma, X., Lu, J., 2014. Verifying selfadaptive applications suffering uncertainty. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ASE '14, pp. 199–210.

Yeh, C., Huang, S., Chang, S., 2013. A black-box based android GUI testing system. In: Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys '13, pp. 529–530.

Yi Qin is a Ph.D. student with the State Key Laboratory for Novel Software Technology and the Department of Computer Science and Technology of Nanjing University. His research interests include self-adaptive application testing.

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

Ping Yu is an assistant professor with the State Key Laboratory for Novel Software Technology and the Department of Computer Science and Technology of Nanjing University. She received her Ph.D. degree in computer science and technology from Nanjing University. Her research interests include software engineering and methodology.

Jian Lu is a full professor with the State Key Laboratory for Novel Software Technology and the Department of Computer Science and Technology of Nanjing University. He received his Ph.D. degree in computer science and technology from Nanjing University. Her research interests include software engineering and methodology.