Verifying Self-adaptive Applications Suffering Uncertainty

Wenhua Yang[§†], Chang Xu[§†*], Yepang Liu[‡], Chun Cao[§†], Xiaoxing Ma[§†], Jian Lu[§†]
[§] State Key Lab for Novel Soft. Tech., Nanjing University, Nanjing, China
[†] Dept. of Comp. Sci. and Tech., Nanjing University, Nanjing, China
ihope1024@gmail.com, {changxu, caochun, xxm, lj}@nju.edu.cn
[‡] Dept. of Comp. Sci. and Engr., The Hong Kong Univ. of Sci. and Tech., Hong Kong, China
andrewust@cse.ust.hk

ABSTRACT

Self-adaptive applications address environmental dynamics systematically. They can be faulty and exhibit runtime errors when environmental dynamics are not considered adequately. It becomes more severe when uncertainty exists in their sensing and adaptation to environments. Existing work verifies self-adaptive applications, but does not explicitly consider environmental constraints or uncertainty. This gives rise to inaccurate verification results. In this paper, we address this problem by proposing a novel approach to verifying self-adaptive applications suffering uncertainty in their environmental interactions. It builds Interactive State Machine (ISM) models for such applications and verifies them with explicit consideration of environmental constraints and uncertainty. It then refines verification results by prioritizing counterexamples according to their probabilities. We experimentally evaluated our approach with real-life self-adaptive applications, and the experimental results confirmed its effectiveness. Our approach reported 200-660% more counterexamples than not considering uncertainty, and eliminated all false counterexamples caused by ignoring environmental constraints.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D. 2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Verification, Reliability, Experimentation

Keywords

Self-adaptive application; verification; uncertainty

1. INTRODUCTION

Self-adaptive applications are gaining increasing popularity, e.g., Locale [1], Phone-Adapter [28, 29] and Navia [2]. These applications continually sense their environments and make adaptation according to their predefined logics [23, 28]. This forms a reaction loop [6], and an application's adaptation upon certain environmental changes can then affect environmental sensing afterwards. Thus self-adaptive applications and their environments are together creating complex and correlated interactions. This causes challenges to building dependable self-adaptive applications, since developers have to consider every detail of such interactions in a predictive way. Real-world self-adaptive applications are thus error-prone [20, 28, 32, 34].

To assure the quality of self-adaptive applications, many research efforts focus on sophisticated testing or debugging techniques. Some techniques [8, 24, 34] use fault patterns to dynamically detect and analyze faults in these applications. Others work on test case generation [30, 32] or test adequacy criteria [22] to support effective fault detection. However, one outstanding challenge in fault detection for self-adaptive applications is that these applications keep interacting with dynamic environments. It is generally infeasible to predict and enumerate all possible environmental conditions that an application can encounter at runtime [23, 27]. This makes existing techniques unable to test self-adaptive applications adequately and precisely. For example, Ramirez et al. [27] used simulation to explore environmental conditions that may cause requirement or behavior violation to an application, but only limited simulation environments can be tested due to its complexity and cost.

On the other hand, formal methods such as model checking and theorem proving rigorously verify an application's behavior [33]. Given an application, these techniques exhaustively explore its state space to detect potential errors (e.g., dead state or abnormal behavior). Recent studies have also reported promising results in applying these techniques to verifying properties of self-adaptive applications, e.g., safety [13, 40], reliability [7, 17], liveness and reachability [21, 28, 29, 40], consistency [21] and stability [4, 28, 29]. However, when it comes to verifying self-adaptive applications under real-world environments, these pieces of existing work have two major limitations:

Lack of modeling environmental constraints. First, an application's running environment can be subject to implicit constraints enforced by this environment's physical laws or assumed by the application's prior knowledge. Consider an example robot-car application [38, 39] that aims to explore an unknown area without bumping into any obstacle. The car can sense its four-directional distances (front, back, left and right) to nearby obstacles using its built-in ultrasonic sensors. Based on these sensed distances, it decides its next movement to avoid obstacles. Suppose that the car keeps walking forward with an obstacle detected ahead. Its sensed distance to this obstacle would keep decreasing. One can derive a constraint for this situation: the last sensed distance should be equal to the summation of the new sensed distance and how far the

*Corresponding author

ASE'14, September 15-19, 2014, Vasteras, Sweden.

http://dx.doi.org/10.1145/2642937.2642999 .

car has walked since its last sensing. Such constraints relate environmental sensing (e.g., distance sensing) to application adaptation (e.g., car walking). We name them *environmental constraints*. Overlooking them can lead to inaccurate verification results. For example, Sama et al. [28, 29] used model checking to detect faults for self-adaptive applications. The work does not model or use environmental constraints in verification, and thus many reported faults are false positives [21].

Lack of modeling uncertainty. Second, an application's environmental sensing and adaptation can be affected by uncertainty, which is inevitably caused by unreliable environmental sensing and flawed physical actions [26]. For environmental sensing, an application may only be able to obtain an estimate of its environmental conditions, but never know its real state. For example, the car's distance sensing always contains unpredictable noise, and sensed values may not faithfully reflect real distances from the car to its nearby obstacles. Similarly, for adaptation an application can only interact with its environment as designed, but may not know whether the interaction indeed proceeds as expected. For example, the car can make 90°- left or -right turns but with unpredictable error. In practice, a left turn can actually be 85° and the car may not be able to know this error. Such sensing and adaptation uncertainty can cause inconsistency between an application's understanding to its environment and its actual environmental conditions, thus affecting the application's functionalities [27]. Similar to environmental constraints, overlooking such uncertainty can also lead to inaccurate verification results. However, to the best of our knowledge, none of existing work [4, 7, 17, 21, 28, 29, 40] has explicitly modeled and considered such uncertainty in verifying self-adaptive applications.

In this paper, we address these two limitations by proposing a novel approach to verifying self-adaptive applications under uncertain environmental dynamics. Our approach works in three phases:

Phase 1: Modeling adaptation logic and environmental constraints. We model the reaction loop of a self-adaptive application with Interactive State Machine (ISM), and explicitly specify environmental constraints for the application.

Phase 2: Verifying the model with uncertainty considered. We consider the uncertainty in this application's sensing and adaptation to its environment by specifying the error range and distribution of each related variable. We then verify the application through its ISM model taking all error ranges into account.

Phase 3: Prioritizing counterexamples. From the verification, we obtain counterexamples that lead to the application's potential runtime errors . We refine verification results by ranking generated counterexamples according to their occurrence probabilities.

We show that by modeling and exploiting such environmental constraints and uncertainty, verifying self-adaptive applications can receive results with greatly increased accuracy. Our approach reported 200-660% more counterexamples than not considering uncertainty, and eliminated all false counterexamples caused by ignoring environmental constraints. The experimental results consistently showed that our verification approach is more accurate and achieves good performance as well as scalability.

We summarize our contributions in this paper below:

- We propose a novel ISM model to explicitly consider environmental constraints and uncertainty in verifying self-adaptive applications. This greatly increases verification accuracy.
- We propose a prioritization technique to rank counterexamples generated from verification according to their occurrence probabilities. This enables developers to focus on most likely faults.
- We evaluate our verification approach with self-adaptive applications by both real and simulation experiments, and validate its usefulness in practice.

The remainder of this paper is organized as follows. Section 2 introduces our modeling of self-adaptive applications. Section 3 uses a motivating example to explain the inadequacy of existing work and motivate our work. Section 4 presents our verification approach in detail. Section 5 validates our approach with self-adaptive applications. Section 6 discusses related work, and finally Section 7 concludes this paper.

2. MODELING SELF-ADAPTIVE APPLICATIONS

As mentioned earlier, self-adaptive applications continually sense their environments and make adaptation upon perceived environmental changes. Their functionalities are realized by the collaboration of three parts: environmental sensing, decision making, and application adaptation. Thus, a self-adaptive application's running can be described by a reaction loop, as illustrated in Figure 1. First, the application senses its environment to capture interesting environmental changes. Then, according to its predefined logics, the application makes a decision by selecting appropriate adaptation to such environmental changes. The adaptation can change the environment and affect the application's environmental sensing afterwards. Thus the environmental sensing is related to the adaptation by environmental constraints as explained earlier. As Figure 1 shows, the sensing and adaptation can also be affected by uncertainty, which is caused by imperfect sensing technologies and flawed physical actions nowadays. Since existing modeling approaches for self-adaptive applications such as A-FSM [28, 29] do not explicitly consider how an application's adaptation affects its environmental sensing afterwards, we propose a new model, named Interactive State Machine (ISM) to meet this requirement.

Given a self-adaptive application, we define its ISM as a tuple $M := (S, V, R, s_0)$, in which symbols are explained below:

- S is a set of this application's all states, and $s_0 \in S$ is its initial state, with which the application starts.
- V is a set containing this application's all variables. $V = V_s \cup V_n$, in which V_s and V_n represent two disjointed categories. V_s contains all *sensing variables*, which store values of environmental attributes interesting to this application (updated by relevant sensing devices). V_n contains other normal variables, i.e., *non-sensing variables*.
- R is a set containing this application's all *adaptation rules* (or rules for short). For each rule $r \in R$, r is associated with a state $s \in S$, which is r's source state. Rule r takes a form of $r := (condition, actions)$. *condition* is a logical formula built on V, and its satisfaction would trigger the execution of this rule. *actions* specifies what should be done when executing this rule. *actions* can include *internal actions* and *interactive actions*. The former takes internal adaptation by updating values of non-sensing variables, e.g., updating the application's current state, and in this case the application transits to a new state. The latter takes interactive adaptation by interacting with the application's environment directly, e.g., making a robot-car move forward. *interactive actions* can also specify the extents of their effects on the environment by updating certain non-sensing variables, e.g., recording the distance that the car has moved forward. *interactive*

actions would also update values of sensing variables implicitly through environment constraints (the updates are explicitly performed upon next environmental sensing).

ISM is executable. Starting from its initial state s_0, an ISM $M := (S, V, R, s_0)$ repeatedly reads values of its sensing variables (automatically updated by environmental sensing), then evaluates and decides which rule to execute, and finally conducts the executed rule's associated actions. When a state $s \in S$ is set as M's current state, rules having this state as source state are enabled, while other rules are disabled. Only enabled rules participate in rule evaluation upon each environmental sensing. When an enabled rule r's condition *r.condition* is satisfied, the rule is triggered for execution. If multiple rules are triggered, only one of them is selected to execute. This tie can be resolved by some priority or random mechanisms [28, 35, 36], which are not our focus in this paper and therefore omitted. When a rule r is selected to execute, its actions *r.actions* are conducted in a sequential way. Conducting actions concurrently might be possible for certain scenarios, but this also is not our focus in this paper and therefore omitted. Thus, an ISM M's execution can be conceptually modeled by a path which is a sequence of states and rules: $\sigma = s_0\ r_1\ s_1 \ldots r_n\ s_n$. Similar to traditional programming, we define *path condition* of execution σ as:

$$pc(\sigma) = \bigwedge_{i=1}^{n} r_i.condition$$

To be representative, we adopt a quantifier-free first-order logic based language for specifying a rule's condition. With this language, a rule's condition can be specified by a logical formula that is recursively constructed using the following syntax:

$$f := (f)\ \texttt{and}\ (f) \mid (f)\ \texttt{or}\ (f) \mid (f)\ \texttt{implies}\ (f) \mid \texttt{not}\ (f) \mid bfunc(v, \ldots, v).$$

Note that the above "and", "or", "implies" and "not" logical connectives follow their traditional interpretations, i.e., representing conjunction, disjunction, implication and negation operations, respectively. Terminal *bfunc* refers to any user-defined or domain-specific function that returns either true or false. When a *bfunc* is easy to understand, one may also use its corresponding operator to simplify its representation, e.g., "$largerThan(v, 50)$" can also be represented by "$v > 50$".

3. MOTIVATING EXAMPLE

In this section, we present a motivating example using our aforementioned self-adaptive robot-car application. The application controls a robot-car to explore an unknown area and avoid bumping into any obstacle. If the car bumps into an obstacle, we say that the application *fails*. We first use our ISM model to specify this application's adaptation logics. We illustrate it partially in Figure 2 due to page limit, but this suffices for explaining our problem.

The ISM model for this application is: $M := (S, V, R, s_0)$, where $S := \{A, B, \ldots, E, \ldots\}$, $R := \{r_0, r_1, \ldots, r_{14}, \ldots\}$, V is the set of variables used in this application (in particular, by R), and the application's initial state s_0:= A. We in the following take state A and its associated rules for example, and other states and rules can be explained similarly.

There are four rules associated with State A, i.e., having A as their source states: r_0, r_1, r_2 and r_3 (multiple actions are sequentially separated by semicolon ";"):

r_0 := ("$disF \geq 20$", "walkF").
r_1 := ("($disF < 20$) and ($disL \geq 20$)", "turnL; walkF; turnL; updateState(B)").
r_2 := ("($disF < 20$) and (($disL < 20$) and ($disR \geq 20$))", "turnR; walkF; turnR; updateState(D)").
r_3 := ("($disF < 20$) and (($disL < 20$) and ($disR < 20$))", "walkB; updateState(E)").

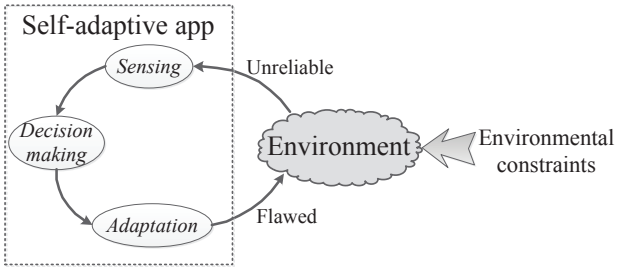

Figure 1: Reaction loop between a self-adaptive application and its environment.

These rules reference four variables: $disF$, $disB$, $disL$ and $disR$. They are all sensing variables, representing sensed distances between the car and its four-directional obstacles (front, back, left and right), respectively. If a sensed distance is no less than 20 cm, we consider this distance *safe*. These rules also involve some actions. For example, actions walkF and walkB mean driving the car to walk forward and backward by a unit distance (say, 10cm), respectively. Actions turnL and turnR represent turning the car left and right by 90°, respectively. At State A, the car keeps walking forward if its sensed distance ahead is safe (Rule r_0). If this distance is no longer safe but the car's left distance is still safe, the car would turn left, walk forward for a unit distance, and then turn left again (Rule r_1). After these interactive actions, the application also conducts a state-transition internal action, $updateState(B)$, which transits the application to a new state (B). At State B, new rules associated with this state(r_4, r_8, r_9) are enabled while the previous rules (r_0, r_1, r_2,r_3) are all disabled. Rule r_2 works similarly as Rule r_1 except that it turns the car to right and transits the application to State D. If the car's sensed distances at three directions (front, left and right) are all not safe, the application would drive the car to walk backward for a unit distance and then transit to State E (Rule r_3).

Consider our earlier failure definition. The car should not bump into any obstacle. Since the car can walk only forward or backward, if its last action is walkF, the failure condition is "$disF \leq 0$", or otherwise (i.e., walkB) the failure condition is "$disB \leq 0$" (the failure condition would be different with uncertainty and environment constraints, which will be discussed later).

Now we verify this ISM model to check whether it contains any problem. We note that if one does not model and consider environmental constraints and uncertainty, the verification results can be inaccurate. For example, consider an execution: $A\ r_0\ A\ r_1\ B$. Its path condition is "$disF_0 \geq 20$ and $disF_1 < 20$ and $disL_1 \geq 20$" (since removing parentheses from a conjunction formula does not change its value, all parentheses are omitted for simplicity). Here, different subscripts represent sensed values at different time points. Suppose that after executing action walkF in Rule r_0, the car bumps into an obstacle, i.e., the failure condition "$disF \leq 0$" is satisfied. Without considering uncertainty caused by unreliable sensing, $disF$ would equal to $disF_1$, which is the sensed value of $disF$ after executing action walkF in Rule r_0. We concatenate this failure condition to the earlier path condition and try to solve the whole constraint. If we can successfully solve this constraint, we would ob-

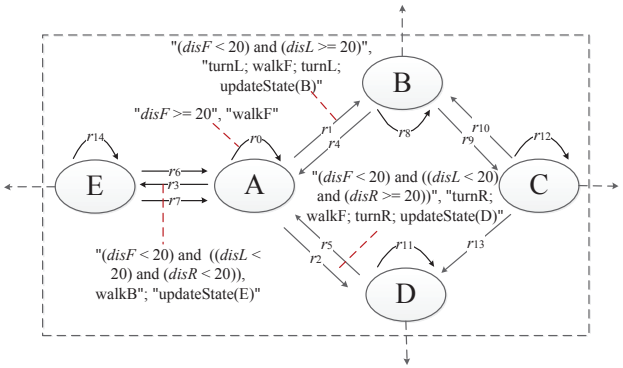

Figure 2: A partial ISM model for our example self-adaptive robot-car application (ellipses represent states and arcs represent rules).

tain a concrete answer on why this execution fails. This execution with the answer is also known as a *counterexample*.

Regarding the above constraint "$disF_0 \geq 20$ and $disF_1 < 20$ and $disL_1 \geq 20$ and $disF_1 \leq 0$", one possible counterexample for solving this constraint is: $disF_0 = 100$, $disF_1 = 0$ and $disL_1 = 30$. This counterexample seems feasible but it violates an environmental constraint connecting $disF_0$ and $disF_1$, which is established by action walkF in Rule r_0. We observe that the distance between $disF_0$ and $disF_1$ is too large ($0 - 100 = -100$ cm) such that it cannot be accomplished by walkF (10cm). This makes the counterexample unreal and thus useless (i.e., a false positive).

On the other hand, real counterexamples (i.e., true positives) may also be missed if one does not consider uncertainty in verifying this application. For instance, consider the same execution in the above example: $A\ r_0\ A\ r_1\ B$. By using the approach of modeling environmental constraints, which will be introduced later, we can have a constraint: $disF_1 = disF_0 - unit$, since $disF_0$'s value becomes $disF_1$'s value due to the effect of action walkF, which moves the car forward by a unit distance. Suppose that one wants to know whether after executing action walkF in the first r_0, the application can fail, i.e., "$disF_1 \leq 0$"is satisfied. Concatenating the path condition and the failure condition together and trying to solve the whole constraint would return no result, since $unit = 10$ and thus $disF_1$, which equals to $disF_0 - unit$, must be larger than 0 based on the fact "$disF_0 \geq 20$". However, both environmental sensing and application adaptation can contain uncertainty as explained earlier. Suppose that $disF$'s value is subject to an error range of [-6, 6] and the car's walked unit distance falls in an error range of [-5, 5]. Then there still exists possibility that the car bumps into an obstacle in this situation. Therefore, this calls for new effort to model such uncertainty to enhance our constraints so that one can discover such potential problems in a self-adaptive application in a more accurate way.

4. VERIFYING SELF-ADAPTIVE APPLICATIONS

In this section we present our approach to verifying self-adaptive applications with environmental constraints and uncertainty. We begin with an overview of the approach, followed by detailed explanations.

4.1 Approach Overview

Given a self-adaptive application, we build its ISM model. As discussed in the motivating example, the failure condition of our robot-car application is that the car bumps into any obstacle. This is verified by checking each path in the ISM to see whether the concatenation of its path condition and the failure condition can be satisfied. If so, a counterexample is found, which corresponds to an application failure.

Algorithm 1 Verification algorithm.

Input:
ISM $M := (S, R, V, s_0)$, failure condition fc, and bound k.
Output:
Set C of counterexamples with probabilities.

```
1:  C := ∅
2:  path :=< s0 >; i := 0;
3:  repeat
4:    s := the last state of path;
5:    if i < k && s has unexplored rules then
6:      r := an unexplored rule of s;
7:      s' := the state that r leads to;
8:      append r and s' to path; i := i + 1;
9:      extract the path's path condition pc;
10:     augment pc with environmental constraints;
11:     modify pc by introducing uncertainty;
12:     check pc && fc with a constraint solver;
13:     if pc && fc is satisfied then
14:       estimate the probability of the counterexample;
15:       add the counterexample with probability to C;
16:     end if
17:   else
18:     remove s' and r from path;
19:     i := i − 1;
20:   end if
21: until path == ∅;
22: return C;
```

The overview of our verification approach is shown in Algorithm 4.1. The algorithm takes an ISM M, a failure condition fc, and a bound k as its inputs. The main data structure used in the algorithm is a list *path*, which records the path that is currently being explored. The algorithm traverses the ISM in a depth-first manner to find new paths (Lines 5-8, Lines 18-19). Since the number of optional paths could be infinitely many, for practical considerations, our approach bounds the length of a path being explored with a configurable integer during the traversal (Line 5: $i < k$). Our approach also supports setting a time budget to prevent endless traversal. The traversal ends when there is no unexplored path within the bound (Line 21).

For each selected path, the algorithm extracts its path condition pc. The ISM explicitly specifies the adaptation logics of an application, but does not include its environmental constraints and uncertainty. So we augment the path condition with environmental constraints and uncertainty for realistic verification. Then the path condition is passed to Z3 [12], an efficient SMT solver, to check whether it can be satisfied. If so, a counterexample is found. It is possible that many counterexamples can be reported. To make the verification results more actionable to users, our approach prioritizes the reported counterexamples according to their occurrence probabilities. In the following three subsections, we present our ideas of modeling environmental constraints, dealing with uncertainty, and prioritizing counterexamples in detail.

4.2 Modeling Environmental Constraints

A self-adaptive application's execution can be described by a reaction loop. Each environmental sensing provides information for

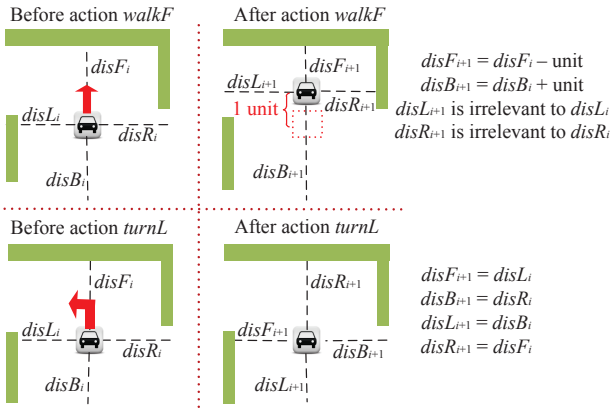

Figure 3: Environmental constraints for actions.

the application to select an appropriate adaptation, which can further change the environment and affect the application's next environmental sensing. Since the new environment is the result of the adaptation's effects on the previous environment, it cannot be arbitrary. This is because inherent constraints in the environment, such as physical laws or the constraints predefined by the application domain, could be violated. For example, for the robot-car application, suppose that the car keeps walking forward with an obstacle detected ahead. If each sensed environment of the application is treated independently and can be arbitrary, we can have that the application's sensed distance to its front obstacle in one environmental sensing is larger than that in the previous sensing. This clearly contradicts physical laws and makes thus reported counterexample useless.

The constraints related each environmental sensing to its previous sensing and adaptation are named *environmental constraints*, as explained before. For a self-adaptive application, we build its ISM $M := (S, V, R, s_0)$. Suppose a rule $r \in R$ has been just triggered, and *interactive actions* of *r.actions* has affected the environment. Then an environmental constraint is a formula $con_{env}(V, V', E)$, where V and V' are the set of sensing variables before and after the *interactive actions*, respectively, and E is a set of non-sensing variables representing the effects of *interactive actions*. The semantics of con_{env} can be derived from inherent constraints of the environment. Take the robot-car application as an example again. Suppose that the car takes action walkF, which makes the car walk forward for a distance *unit*. Then from physical laws we can derive an environmental constraint $con_{env}(\{disF_i\}, \{disF_{i+1}\}, \{unit\})$ meaning that the last sensed distance $disF_i$ should be equal to the summation of the new sensed distance $disF_{i+1}$ and the distance *unit* that the car has walked since its last sensing, i.e., $con_{env}(\{disF_i\}, \{disF_{i+1}\}, \{unit\})$:$disF_i = disF_{i+1} + unit$. The environmental constraint $con_{env}(V, V', E)$ requires that when values of the sensing variables in V' are explicitly updated upon an environmental sensing, these updated values should satisfy $con_{env}(V, V', E)$. Thus, through environmental constraints, we provide a means to enable *interactive actions* to update sensing variables implicitly. For $con_{env}(\{disF_i\}, \{disF_{i+1}\}, \{unit\})$ in the above example, it requires that the update of $disF_{i+1}$ should satisfy $disF_{i+1} = disF_i - unit$, as illustrated in Figure 3 (top). However, for action walkF, there are no constraints about $disL_{i+1}$ and $disL_i$, or $disR_{i+1}$ and $disR_i$. This is because after the robot-car walks forward for a distance *unit*, the distances between the car and its left and right obstacles can be arbitrary and thus no environmental constraints are specified.

Figure 3 also shows another example in the bottom about action turnL, which is a turning-left action. Similar environmental constraints exist for actions walkB and turnR, which are the walking-backward and turning-right action, respectively. They are omitted due to page limit.

Now we can augment the path condition of a selected path with such environmental constraints, as mentioned in the overview. For each rule in the path, if there are environmental constraints for taking the rule's *interactive actions*, we concatenate the environmental constraints to the path condition. After it is done, we get a new formula of constraints for the path, which is called the *ideal condition*. For example, for path $\sigma = A\ r_0\ A$ in the robot-car example, its path condition is "$disF_0 \geq 20$". For Rule r_0, it has an interactive action walkF. Let $disF_1$ be the distance between the car and its front obstacle after taking the action walkF. Then as mentioned before, there is an environmental constraint "$disF_1 = disF_0 - unit$". After concatenating the environmental constraint to the path condition, we get the ideal condition of path σ, which is "$disF_0 \geq 20$ `and` $disF_1 = disF_0 - unit$". For each of the selected paths, we process the path condition in the same manner to get the ideal condition, which will be further processed to include uncertainty, as described in the next subsection.

4.3 Dealing with Uncertainty

Self-adaptive applications' environmental sensing and adaptation can be affected by uncertainty, which is naturally caused by unreliable sensing and flawed adaptation [26]. The uncertainty can cause inconsistent understandings for an application between its sensed environment and actual environment [38]. As suggested by the motivating example, the application may fall into failure due to its inaccurate understanding to the environment caused by uncertainty. Thus, we should consider uncertainty in verifying self-adaptive applications. Otherwise, the accuracy of verification results cannot be guaranteed.

However, it is difficult to specify uncertainty precisely in the modeling and verification process, because uncertainty comes from various sources and appears in different forms. We studied uncertainty in many real cases and observed that uncertainty caused by unreliable sensing or flawed adaptation demonstrates regular patterns. The sensed value of unreliable sensing or the effects of flawed adaptation often fall into an error range, with a distribution determined by the physical characteristics of sensing technologies and actions. In this section, we explain how to model this kind of uncertainty in the verification of self-adaptive applications.

Uncertainty affects self-adaptive applications' sensing and adaptation, and thus affects values of sensing variables and non-sensing variables that represent the effects of adaptation. Given an ISM, to model uncertainty, we first need to identify the variables in an ideal condition that would be affected by uncertainty. Then for each of these variables, we give an error range and a distribution for its potential value, i.e., for a variable v affected by uncertainty, let $[a, b]$ $(a < b)$ be the error range of v. Then the lower bound and upper bound of variable v are $v + a$ and $v + b$ respectively. Meanwhile, we set a distribution p of the variable's value between its lower and upper bounds. The lower bound, upper bound and the distribution of a variable can be obtained from field studies or experiments with statistical analysis. In the robot-car application, the distance $disF$ between the car and its front obstacle is affected by uncertainty. We found that the ultrasonic sensor used in the robot-car has an error in sensing. Field studies show that its error range is [-6, 6] cm and its error distribution is a Gaussian one. So, for variable $disF$, we give an error range of $[-6, 6]$. Then the lower bound and upper bound are $disF - 6$ and $disF + 6$, respectively. The distribution of $disF$'s

value between its lower and upper bounds is a Gaussian distribution. Similarly, from field studies we learned that the distance *unit* of the car's walking action falls into an error range [-5, 5] cm, and the distribution of its value is also a Gaussian distribution.

Now we can show how to augment an ideal condition with uncertainty. In an ideal condition, for all its variables, there is no uncertainty considered. Therefore, for each variable v, which is affected by uncertainty in the ideal condition, since v does not include uncertainty, we use a new variable v' to represent v with uncertainty. v' satisfies the constraint $v+a \leq v' \leq v+b$, where $[a,b]$ is the error range of v. This means that the value of v' can range from $v+a$ to $v+b$. Clearly, for environmental sensing, the value of v' is a sensed value of the application, and the value of v is the actual value about the environment. For adaptation, the value of v' records its actual effects on the environment, and the value of v records its ideal effects assumed by the application. We also set a distribution $P(v')$ for v' that will be used to prioritize later reported counterexamples, which is explained in the next subsection. Then we replace v with v' in the ideal condition, and join the constraint $v+a \leq v' \leq v+b$ with the ideal condition. This forms a new condition named the *actual condition*.

Take path $\sigma = A\ r_0\ A$ in the robot-car application mentioned earlier for illustration. The ideal condition of the path is "$disF_0 \geq 20$ `and` $disF_1 = disF_0 - unit$", in which variables $disF_0$, $disF_1$, and *unit* are affected by uncertainty. The error ranges for $disF_0$ and $disF_1$ are [-6, 6], and the error range of *unit* is [-5, 5]. We use new variables $disF_0'$, $disF_1'$ and $unit'$ to replace the counterparts in the ideal condition, respectively. The constraint between $disF_0$ and $disF_0'$ is "$disF_0 - 6 \leq disF_0' \leq disF_0 + 6$". Other variables' constraints between themselves and the new variables are similar. Then we combine all these new constraints into the ideal condition, and get the actual condition of σ, i.e., "$disF_0' \geq 20$ `and` $disF_1' = disF_0' - unit'$ `and` $disF_0 - 6 \leq disF_0' \leq disF_0 + 6$ `and` $disF_1 - 6 \leq disF_1' \leq disF_1 + 6$ `and` $unit - 5 \leq unit' \leq unit + 5$".

4.4 Prioritizing Counterexamples

To check whether an execution can lead to application failure, we concatenate a failure condition to the actual path condition associated with this execution to get a new constraint, which is then passed to Z3 to check whether there is a satisfying solution. If yes, the execution can fall into failure. In the solution, for each variable, Z3 will provide it with a value. The values of all sensing variables represent an application's understanding to the environment. Based on these values, we can construct a partial environment in which the application fails. The reason why the environment may be partial is that the application may not sense all environmental attributes. A path σ contains a sequence of actions that the application takes, which would lead to the application's failure if a solution exists. The path σ and the solution together indicate an application failure, and that is why we name them a counterexample. A counterexample is a tuple $t = (\sigma, L)$, where L is a mapping $L : V \rightarrow A$, where V is a set $\{v_0, v_1, \ldots, v_n\}$ containing all variables in the actual condition of σ and A is a set of values $\{a_0, a_1, \ldots, a_n\}$ where $L(v_i) = a_i\ (0 \leq i \leq n)$ in the solution. A counterexample represents a failed application execution in a certain environment. For example, for the robot-car application, we already know the actual condition of path $\sigma = A\ r_0\ A$ (cf. the previous subsection). The failure condition of this path σ is $disF_1 \leq 0$. Then by solving the concatenation of the actual condition and the failure condition, we can obtain a solution of "$disF_0 = 14, disF_0' = 20, disF_1 = 0, disF_1' = 6, unit = 10, unit' = 14$". This solution corresponds to the following failure scenario. The distance between the robot-car and its front obstacle is initially 14 ($disF_0$), but due to uncertainty, the application's sensed distance is 20 ($disF_0'$). Based on the sensed distance, the application makes the car walk forward for a unit distance that is assumed to be 10cm (*unit*). However, the actually walked distance is 14cm ($unit'$). As a result, although the new sensed distance of the application is 6cm ($disF_1'$), in reality the car has already bumped into the obstacle ($disF_1 = 0$). This counterexample would not have been reported, if we do not consider uncertainty in environmental sensing and application adaptation.

Self-adaptive applications are more likely to fall into failures in an environment with uncertainty [14, 26, 27]. As a result, many counterexamples could be reported by solving concatenation of actual conditions and failure conditions. However, for a certain counterexample $t = (\sigma, L)$, an application may not fail by exactly following the actions specified in σ under the environment constructed from the values of variables in L. This is because each time the application runs, due to uncertainty, the application's sensed environment can be different from its actual environment, and its adaptation's effects can be different from its assumption. The likelihood for application making the same actions as the ones in σ that lead to failure in the environment corresponding to the variables' values in L is called the probability of counterexample $t = (\sigma, L)$. We propose to prioritize reported counterexamples to save the effort for fault inspection and fixing. We believe that the more likely a counterexample is to occur, the more attention it needs. Therefore, we rank counterexamples according to their probabilities from high to low.

For a counterexample $t = (\sigma, L)$ to occur in the environment constructed from L, it requires that the application should make the same sequence of actions as specified in the rules of σ. This means that for each rule r in σ, *r.condition* should be satisfied so that *r.actions* can be taken. Thus we first estimate the probability of satisfaction of *r.condition*. Suppose that *r.condition* involves one variable v affected by uncertainty, and the variable replacing v to model uncertainty is v'. Let the error range of v be $[a,b]$, and the value of v in L be d. So the lower bound and upper bound of the value of v' are $d+a$ and $d+b$, respectively. Then the probability of *r.condition* being true **true** can be calculated according to the following Equation 1. Function $P(v')$ is the probability density function of v'. Function $R(v')$ is defined as Equation 2. For a given value of v', $R(v')$ returns 1 if *r.condition* is **true**, and returns 0 otherwise. For other possible variables involved in *r.condition*, their values are fixed using those values in L.

$$Prob = \int_{d+a}^{d+b} P(v')R(v')\mathrm{d}v' \tag{1}$$

$$R(v') = \begin{cases} 1, & \text{if } r.condition = \text{true} \\ 0, & \text{if } r.condition = \text{false} \end{cases} \tag{2}$$

For example, we assume that Rule r_0 is the first rule in a counterexample's path. The rule's condition $disF \geq 20$ involves one variable $disF$ affected by uncertainty, and we replace $disF$ with $disF'$. The error range of $disF$ is $[-6,6]$, and the value of $disF$ solved in the counterexample is 17. It means that $disF'$'s value can range from 11 to 23 due to uncertainty. The distribution of $disF'$'s value complies with the Gaussian distribution $N(17, 2^2)$. So the probability of $disF' \geq 20$ being **true** is calculated according to Equation 1 as $\int_{11}^{23} P(disF')R(disF')\mathrm{d}disF'$. Since $R(disF') = 0$ when $disF' \in [11, 20)$, and $R(disF') = 1$ when $disF' \in [20, 23]$, the above equation is equivalent to $\int_{20}^{23} P(disF')\mathrm{d}disF'$, which results in

$$\int_{20}^{23} \frac{1}{2\sqrt{2\pi}} e^{-\frac{(disF'-17)^2}{8}} \mathrm{d}diF' = 0.0655.$$

Note that there can be more than one variable affected by uncertainty in a rule's condition, and there can be more than one rule in a path. Variables in one rule's condition may affect each other, and variables across rules may have effects on each other as well. In theory, if we want to calculate the probability of satisfaction of all rules' conditions, we need to treat all the conditions in a path as a whole, and perform a multiple integral on all variables that are affected by uncertainty. As we know, the multiple integral can be very inefficient when its condition is complex and there are many variables in the condition. Thus, we give an approximated solution by treating each variable independently. First, for each rule in a counterexample, we calculate the probability of satisfaction of the rule's condition according to Equation 1. If there are more than one variable affected by uncertainty in the rule's condition, a multiple integral is used. Then, we multiply the probabilities of satisfaction of each rule's condition to get an estimated probability of whole counterexample.

5. VALIDATING OUR APPROACH

In this section, we validate the effectiveness of our approach. We implemented our approach as a prototype, and the validation was carried out in the context of our robot-car application case study. In this study, we are going to answer the following three research questions:

- **RQ1:** *How does our modeling of environmental constraints and uncertainty improve the accuracy of verification of self-adaptive applications?*
- **RQ2:** *Can our approach give an accurate estimate of occurrence probabilities for its reported counterexamples?*
- **RQ3:** *How does the bound of path length configured during verification affect the scalability and effectiveness of our approach?*

5.1 Experimental Setup and Design

Self-adaptive applications are different from conventional applications. They interact with their running environments, and adapt their behavior based on their sensed environmental changes. To conduct our evaluation, we need to carefully select our experimental application subjects. Specifically, both the selected applications and their running environments need to be manageable, as otherwise it is hard for us to deploy the applications for experiments. Guided by this requirement, we selected 12 different robot-car applications with various sizes as our experimental subjects. These applications were independently developed by different researchers and students in our university during the past four years. They adopt different strategies to control a robot-car to explore unknown areas based on collected sensory data. These applications have up to 40 different states and rules.

To answer our research question RQ1, we conducted two experiments. The first experiment studies whether modeling environmental constraints can improve the verification results by eliminating false counterexamples. For comparison purposes, we implemented another approach *naive*, which ignores environmental constraints and uncertainty. We applied both our approach and the *naive* approach to the 12 robot-car applications, and checked the reported counterexamples to see how many of them are false counterexamples. Our second experiment studies whether modeling uncertainty can help improve the verification results. Similar to the first experiment, we implemented an approach *ideal*, which considers environmental constraints but ignores uncertainty, for comparison. We applied both our approach and the *ideal* approach to the 12 robot-car applications, and recorded their reported counterexamples. Then we ran the 12 robot-car applications in both real deployments (i.e., real field study) and simulation to validate these counterexamples. We then compared true counterexamples reported by the two approaches to see whether there is any counterexample reported by *actual* (i.e., our approach) but not by *ideal*, and vice versa. In other words, our goal is to study whether *actual* can report more counterexamples than *ideal*.

Table 1: Experimental results without considering environmental constraints

Application	Counter-example	False positive	False positive rate
App1	120	94	78.33%
App2	60	42	70.00%
App3	108	90	83.33%
App4	74	53	71.62%
App5	120	102	85.00%
App6	90	75	83.33%
App7	120	90	75.00%
App8	51	39	76.47%
App9	62	48	77.42%
App10	45	32	71.11%
App11	51	43	84.31%
App12	62	48	77.42%

To answer research question RQ2, for each counterexample, we need to know its likelihood of occurrence in its corresponding environment in real cases. The likelihood here serves as a ground truth to assess our predicted occurrence probability for each counterexample. So we let an application run in the corresponding environment of every counterexample, and counted the number of failures encountered by this application with respect to its corresponding counterexample. The experiment was conducted for top 10 reported counterexamples of each application in both field study and simulation. As we know, to get an accurate probability one has to collect a fairly large number of samples. The simulation is thus used to refine our experimental results by providing more sampling data. Then we checked calculated probabilities of counterexamples against their observed likelihoods of occurrences, to see how close they are.

The bound of path length is a parameter in our approach which can affect the approach's performance and the number of reported counterexamples. Besides, since our approach takes one path for verification at a time, the maximum bound of the path, instead of the whole size of the ISM, affects the scalability of the approach virtually. Therefore, we need to investigate how the bound impacts our approach. In particular, we want to know how our approach scales and how the number of counterexamples grows as the bound increases. So, to answer our research question RQ3, we applied our approach to the 12 robot-car applications to measure the performance of our approach, and recorded the number of reported counterexamples as the bound increased. The results of all these experiments are discussed in the next section.

5.2 Experimental Results

In this section, we present experimental results to answer our earlier raised three research questions.

RQ1. First we examined the effects of modeling environmental constraints. We implemented the *naive* approach, and verified the 12 robot-car applications with *naive*. The bound of path length was set to 30. Table 1 gives the verification results. Column 2 shows

Table 2: Comparison of experimental results (reported counterexamples) between *actual* and *ideal*

Application	Actual	Ideal	More	Improvement
App1	86	22	64	290.9%
App2	89	29	60	206.9%
App3	81	20	61	305.0%
App4	112	29	83	286.2%
App5	76	10	66	660.0%
App6	65	16	49	306.3%
App7	103	31	72	232.3%
App8	114	27	87	322.2%
App9	102	34	68	200.0%
App10	124	25	99	396.0%
App11	114	27	87	322.2%
App12	102	34	68	200.0%

the number of reported counterexamples. We examined these counterexamples one by one manually, and found that many of them violated environmental constraints. For example, in one counterexample, after the car moved forward for a unit distance, the distance between the car and its front obstacle was bigger than that before the car took the move. These counterexamples will not happen in real deployments, and thus are false counterexamples (false positives). Column 3 shows the number of reported counterexamples which are false. As we can see from Column 4, the false positive rate can vary from 70% to over 84%, which indicates the necessity and importance of considering environmental constraints in the verification process.

Then we examined the effects of modeling uncertainty by comparing the verification results of approach *ideal* and our approach *actual*. We verified the 12 robot-car applications with both approaches, with the bound of path length set to 30. The results are shown in Table 2. Column 2 is the number of reported counterexamples of *actual*, and Column 3 is the number of reported counterexamples of *ideal*. Our approach *actual* reported clearly more counterexamples than *ideal*. Furthermore, by careful examination we found that all the counterexamples reported by *ideal* are also reported by *actual*. In the meantime, we confirmed that all counterexamples reported by *actual* can happen in real environments. Most confirmations were acquired by field study (more than 85%). The others (less than 15%) were acquired by simulation, because these counterexamples have a fairly low probability to occur, and therefore were difficult to be witnessed in the field study. The robot-car and the simulation used to do the field study and simulation are shown in Figure 4. The experimental results show that our approach *actual* reported 200-660% more counterexamples than approach *ideal*, which demonstrates that our approach has a better accuracy (more complete).

Based on the above discussed experimental results, we derive our answer to research question RQ1: *Modeling environmental constraints and uncertainty can greatly improve the accuracy of verification of self-adaptive applications*.

RQ2. We also ranked the counterexamples reported by our approach from the above experiment with their predicted probabilities. To assess the accuracy of these probabilities, we selected the top 10 counterexamples from each of the 12 robot-car application (i.e., 120 counterexamples in total) for further study. Specifically, for each selected counterexample, we let the concerned application to run in the environment constructed from this counterexample in field study for 100 times, and also in simulation for 1,000 times

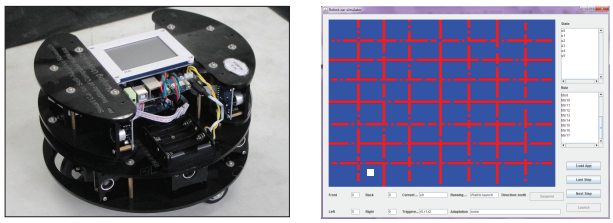

Figure 4: The robot-car and the simulation used in experiments.

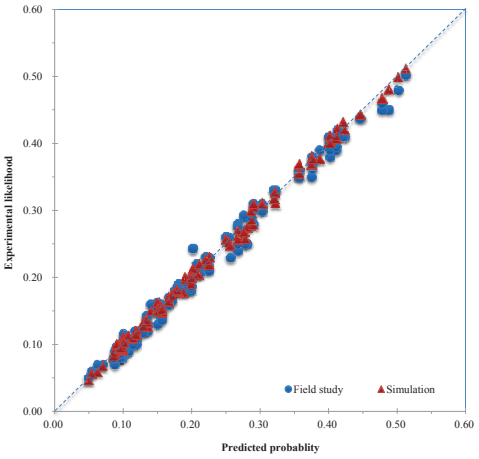

Figure 5: Comparison of predicted probabilities and experimental probabilities.

(i.e., totally 1,100 runs for each counterexample). For each counterexample, we counted the number of its corresponding failures, and obtained a likelihood value for the failure's occurrence. Figure 5 illustrates the results. The horizontal axis represents predicted probability from our approach, and the vertical axis represents the likelihood of occurrence observed from experiments. Each counterexample corresponds to two points in the figure: blue points (solid circle) are for the field study, and red points (solid triangle) are for the simulation. It is clear that for a counterexample, if its predicted probability from our approach is close to its likelihood of occurrence from experiments, its corresponding point in the figure would be close to the diagonal line. As we can observe from Figure 5, most of the points are scattered near the diagonal line. From these results, we derive our answer to research question RQ2: *Our approach can accurately estimate occurrence probabilities for its reported counterexamples*.

RQ3. We ran our approach to verify the robot-car applications with different bounds of path length. The experimental platform is a Dell Desktop PC with an Intel Core 2 CPU @2.53GHz and 2GB RAM, running Windows 7. We recorded the spent time and memory in each run, which are shown in Table 3. As we can observe from the table, when the bound is set to 50, our approach spent about 15 minutes and consumed around 250 MB memory on average. Clearly, given enough time, our approach is able to handle larger bounds. However, if we focus on counterexamples with relatively high probabilities, a bound of 50 is already sufficient for the robot-car applications. This is because we observed that when the bound was set to 50, the increase of the number of counterexamples with a probability higher than 10^{-5}, which is a very small likelihood for a counterexample to occur, tends towards

Table 3: Time and memory costs for verification with different bounds

Application	5	10	15	20	25	30	35	40	45	50
App1	4.0s 40MB	6.8s 48MB	10.5s 61MB	19.8s 72MB	38.5s 84MB	72.3s 97MB	109.3s 118MB	196.3s 129MB	310.8s 153MB	500.3s 178MB
App2	2.3s 55MB	3.8s 70MB	7.0s 87MB	12.8s 102MB	22.5s 116MB	41.3s 134MB	75.5s 153MB	147s 168MB	243.8s 186MB	491.3s 204MB
App3	5.0s 31MB	9.3s 44MB	16.5s 52MB	28.8s 63MB	54.0s 75MB	92.8s 81MB	164.3s 89MB	323.5s 97MB	480.8s 108MB	886.3s 124MB
App4	5.8s 44MB	9.5s 51MB	17.5s 59MB	33.0s 68MB	60.8s 81MB	105.3s 103MB	198.8s 117MB	344.8s 132MB	508.0s 155MB	976.3s 196MB
App5	3.8s 58MB	7.0s 65MB	11.5s 83MB	19.5s 94MB	38.5s 112MB	71.5s 137MB	137.0s 162MB	243.8s 194MB	419.5s 205MB	529.3s 217MB
App6	2.3s 38MB	3.5s 45MB	6.8s 53MB	13.0s 59MB	24.3s 64MB	47.0s 71MB	90.8s 78MB	177.0s 84MB	331.0s 96MB	633.3s 113MB
App7	5.3s 41MB	9.0s 53MB	15.3s 64MB	27.5s 71MB	51.5s 79MB	96.3s 94MB	177.5s 124MB	314.3s 167MB	528.8s 195MB	971.8s 211MB
App8	4.3s 47MB	7.5s 55MB	14.5s 67MB	26.0s 84MB	47.3s 95MB	94.8s 110MB	178.0s 135MB	302.3s 157MB	528.3s 198MB	1000.5s 245MB
App9	5.5s 39MB	9.5s 47MB	18.8s 55MB	37.3s 63MB	60.3s 79MB	116.0s 90MB	203.0s 99MB	385.0s 110MB	742.0s 157MB	1012.5s 203MB
App10	3.3s 29MB	6.3s 43MB	11.3s 60MB	22.0s 75MB	39.0s 91MB	69.5s 105MB	130.5s 119MB	248.5s 168MB	472.5s 211MB	810.0s 264MB
App11	4.5s 45MB	8.0s 53MB	15.0s 60MB	24.8s 76MB	45.0s 91MB	92.5s 105MB	180.3s 129MB	313.8s 154MB	525.3s 212MB	998.5s 231MB
App12	5.0s 35MB	8.8s 46MB	17.3s 60MB	33.5s 68MB	59.0s 81MB	113.8s 90MB	208.5s 107MB	382.3s 113MB	692.0s 163MB	1026.5s 217MB

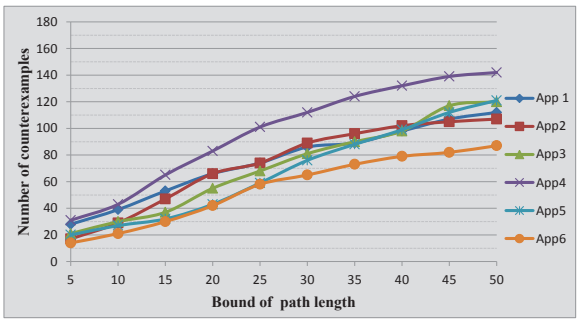

Figure 6: The growing trend of the number of counterexamples with the increase of the bound of path length.

stability. Figure 6 illustrates the number of counterexamples with a probability higher than 10^{-5} when the bound was set from 5 to 50 (for App 1 to App 6). From these discussions, we can derive our answer to research question RQ3: *Our approach can scale well to large bounds of path length and it can detect more counterexamples when the bound increases*.

5.3 Threats to Validity

We analyze threats to the validity of our conclusions below.

Threats to construct validity. The main threat to construct validity for our study is that we may not have run our robot-car applications adequate times in the field study in order to obtain accurate likelihoods of occurrence of the counterexamples. To reduce this threat, we ran each application in simulation for 1,000 times to get the simulated probabilities as complementary evidence. Another potential threat to construct validity is the threshold of probability used to filter out low-probability counterexamples when evaluating the impact of the bound of path length. Nevertheless, we set the probability threshold to 10^{-5}, which is an extremely small likelihood for a counterexample to occur in real cases. In fact, we believe developers should focus mainly on counterexamples with much higher probabilities in a realistic setting.

Threats to internal validity. There are mainly two threats to internal validity. One is that the given ranges and distributions of variables for modeling uncertainty may not be realistic. To reduce this threat, we have conducted a lot of field studies and performed careful statistical analysis. The other threat to internal validity is the selected bound of path length. It is possible to find more counterexamples with a larger bound. Nevertheless, according to our experiments, the number of counterexamples will tend to be stable as the bound increases. So we selected a proper value for the bound in experiments.

Threats to external validity. The main threat to external validity is that our conclusions may not generalize to other self-adaptive applications. To reduce this threat, we selected 12 different robot-car self-adaptive applications with various sizes as our experimental subjects. Meanwhile, we made our best efforts in selecting them being independently developed by different developers, and validated by both simulation and field study. The results consistently support our conclusions. Although we try to make our approach applicable to other self-adaptive applications, there still is a strong need for validating our approach with more realistic applications.

6. RELATED WORK

In this section, we discuss selected related work on self-adaptive applications. Cheng et al. [10] and Lemos et al. [11] presented a comprehensive and in-depth analysis of the current research status including methods and challenges, to which interested readers can refer. Here we focus on modeling, testing and verifying self-adaptive applications.

Modeling self-adaptive applications. To model a self-adaptive application, there are several optional methodologies, such as goal-oriented or rule-based methodologies. Goal-based modeling offers a means to identify and visualize different alternatives for satisfying the overall objectives of a system [9]. The goals here capture the intentions of a stakeholder on a self-adaptive application and its execution environment [25], and can be used to model the requirements of self-adaptive applications [19]. Rule-based modeling uses rules explicitly or implicitly to model an application's expected reactions to monitored events [31], such as the A-FSM approach

[29, 34]. There are also other pieces of work related to modeling self-adaptive applications. Andersson et al. [3] defined four categories of dimensions for modeling self-adaptive applications: self-adaptive goals, causes or triggers of self-adaptation, mechanisms used to adapt, and effects of those mechanisms on applications. Dobson et al. [13] identified four aspects of self-adaptive applications around which decisions can be organized: collection, analysis, decision and action. Brun et al. [6] discussed the importance of making the adaptation control loops explicit during an application's development process, and outlined several types of control loops that can lead to adaptation.

Verifying and testing self-adaptive applications. Designing and deploying certifiable verification and validation methods for self-adaptive applications is one of the major research challenges for the software engineering community in general and the self-adaptive applications community in particular [11]. Concerned properties for self-adaptive applications include safety [13, 40], liveness and reachability [21, 28, 29, 40], reliability[7, 17], and stability [4, 28, 29]. In recent years, different methods have been used in [40, 4, 5] to verify self-adaptive applications. Zhang and Cheng [40] introduced an approach to creating formal models for the behavior of self-adaptive applications. They also presented a process to construct, verify, and validate these models. Bartels et al. used the process algebra CSP for the specification, verification and implementation of self-adaptive applications [4]. Camara et al. [7] proposed an approach for the verification of self-adaptive applications based on stimulation and probabilistic model-checking. It stimulates the environment and collects data about how an application reacts environmental changes to evaluate whether important properties are satisfied within certain confidence levels. Model checking is also widely used to detect some well-known fault patterns in self-adaptive applications [21, 28, 29].

There are also many studies focusing on testing self-adaptive applications. Xu et al. [34] used error patterns to dynamically detect and analyze responsible faults. Tse et al. [30] used isotropic properties of contexts as metamorphic relations for testing context-sensitive software and presented techniques for generating effective test cases. Wang et al. [32] proposed to augment existing test cases to expose faults, by focusing on context switching points that can affect application adaptations. Besides test case generation and augmentation, there are also efforts spent on test adequacy criteria research. For instance, Lu et al. [22] proposed a new set of coverage criteria to test data flows caused by context uses in self-adaptive applications. These pieces of work detect faults in self-adaptive applications, and contribute to their dependability at design and development phases. Some other studies also tried to improve self-adaptive applications' dependability, but from different perspectives. For example, related studies [35, 36, 37] detected and resolved inconsistency in contexts fed to a self-adaptive application. Consistent context data is an important prerequisite for dependable adaptations. Our previous work [38, 39] focused on improving the dependability of self-adaptive applications by monitoring application executions and checking consistency constraints at runtime. Kulkarni et al. [20] also introduced a runtime error-handling framework for programming robust self-adaptive applications by adopting forward recovery strategies. Last but not least, Ramirez et al. [27] introduced a technique for automatically discovering combinations of environmental conditions that produce requirement violations and latent behaviors in a self-adaptive application.

Managing uncertainty. Uncertainty poses a big threat to correct and reliable self-adaptations. This problem is gaining increasing attention in recent years. Ramirez et al. [26] reported a taxonomy of uncertain factors that can affect self-adaptive applications. Their work called for a spectrum of research efforts from requirement specification, application design to runtime support. There are some studies focusing on how to handle uncertainty at design time for self-adaptive applications [15, 14, 9]. Esfahani et al. [15] described an approach to tackling the challenge of uncertainty by assessing both positive and negative consequences of uncertainty, and proposed a framework for managing uncertainty in self-adaptive applications [14]. Cheng et al. [9] proposed a requirement language RELAX to explicitly address uncertainty by enabling engineers to specify uncertainty in application requirements. In their work, adaptation is achieved by relaxing non-critical requirements. Ghezzi et al. [18] proposed a framework that supports adaptation to non-functional manifestations of uncertainty relying on alternative or optional functionalities. The framework allows engineers to derive a finite state automaton augmented with probabilities from an initial model of a self-adaptive application. Famelis et al. [16] specified uncertainty using annotations with well-defined semantics that transforms an application model into a partial model and presented an approach to reasoning with such models.

Our work differs from the existing work in three aspects. First, we explicitly model environmental constraints and uncertainty caused by unreliable sensing and adaptation for self-adaptive applications. Second, we present a novel approach, which exploits the power of SMT solvers, to verifying the correctness of self-adaptive applications affected by uncertainty. Third, we propose to rank counterexamples according to their probabilities, which has not yet been considered in the above literature.

7. CONCLUSION

In this paper, we propose a novel approach to verifying self-adaptive applications. By explicitly considering the environmental constraints, the approach avoids reporting false counterexamples that will not happen in real environments. At the same time, by taking error ranges of the environment-related variables into account, the approach can find lots of potential counterexamples (faults) that would otherwise be overlooked by methods not considering uncertainty in environmental interactions. Our approach also prioritizes its reported counterexamples according to their occurrence probabilities, whose accuracy has been well validated by both simulated experiments and field study.

This work can still be improved. For example, we need to validate this work with more real-world applications. In addition to environmental interactions, we also plan to extend our consideration of uncertainty to those from other sources, such as requirements and adaptation decisions.

8. ACKNOWLEDGMENTS

This research was partially funded by National High-Tech Research & Development Program (863 program 2013AA01A213) and National Natural Science Foundation (61100038, 91318301, 61321491, 61361120097) of China.

9. REFERENCES

[1] Locale. http://www.twofortyfouram.com/.

[2] Navia. http://induct-technology.com/.

[3] J. Andersson, R. Lemos, S. Malek, and D. Weyns. Modeling dimensions of self-adaptive software systems. volume 5525 of *LNCS*, pages 27–47. Springer Berlin Heidelberg, 2009.

[4] B. Bartels and M. Kleine. A csp-based framework for the specification, verification, and implementation of adaptive

systems. In *Proc. SEAMS' 11*, pages 158–167, Honolulu, USA, May 2011.

[5] R. V. Borges, A. d'Avila Garcez, and L. C. Lamb. Integrating model verification and self-adaptation. In *Proc. ASE' 10*, pages 317–320, Antwerp, Belgium, Nov 2010.

[6] Y. Brun, G. Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, and et al. Engineering self-adaptive systems through feedback loops. In *Software Engineering for Self-Adaptive Systems*, pages 48–70. Springer Berlin Heidelberg, 2009.

[7] J. Camara and R. De Lemos. Evaluation of resilience in self-adaptive systems using probabilistic model-checking. In *Proc. SEAMS' 12*, pages 53–62, Zurich, Switzerland, June 2012.

[8] L. Capra, W. Emmerich, and C. Mascolo. Carisma: context-aware reflective middleware system for mobile applications. *IEEE Trans. Softw. Eng.*, 29(10):929–945, 2003.

[9] B. Cheng, P. Sawyer, N. Bencomo, and J. Whittle. A goal-based modeling approach to develop requirements of an adaptive system with environmental uncertainty. volume 5795 of *LNCS*, pages 468–483. Springer Berlin Heidelberg.

[10] B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, J. Magee, and et al. Software engineering for self-adaptive systems: A research roadmap. In *Software Engineering for Self-Adaptive Systems*, pages 1–26, 2009.

[11] R. de Lemos, H. Giese, H. MÍźller, M. Shaw, J. Andersson, and et al. Software engineering for self-adaptive systems: A second research roadmap. In *Software Engineering for Self-Adaptive Systems II*, volume 7475 of *LNCS*, pages 1–32. Springer Berlin Heidelberg, 2013.

[12] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *TACAS'08/ETAPS'08*, pages 337–340, Budapest, Hungary, March-Apr 2008.

[13] S. Dobson and et al. A survey of autonomic communications. *ACM Trans. Auton. Adapt. Syst.*, 1(2):223–259, Dec 2006.

[14] N. Esfahani. A framework for managing uncertainty in self-adaptive software systems. In *Proc. ASE' 11*, pages 646–650, Kansas, USA, Nov 2011.

[15] N. Esfahani, E. Kouroshfar, and S. Malek. Taming uncertainty in self-adaptive software. In *Proc. Joint ESEC/FSE' 11*, pages 234–244, Szeged, Hungary, Sept 2011.

[16] M. Famelis, R. Salay, and M. Chechik. Partial models: Towards modeling and reasoning with uncertainty. In *Proc. ICSE' 12*, pages 573–583, Zurich, Switzerland, June 2012.

[17] A. Filieri and G. Tamburrelli. Probabilistic verification at runtime for self-adaptive systems. volume 7740 of *LNCS*, pages 30–59. Springer Berlin Heidelberg, 2013.

[18] C. Ghezzi, L. S. Pinto, P. Spoletini, and G. Tamburrelli. Managing non-functional uncertainty via model-driven adaptivity. In *Proc. ICSE' 13*, pages 33–42, San Francisco, USA, May 2013.

[19] H. Goldsby, P. Sawyer, N. Bencomo, B. H. C. Cheng, and D. Hughes. Goal-based modeling of dynamically adaptive system requirements. In *Proc. ECBS' 08*, pages 36–45, Belfast, Northern Ireland, March 2008.

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

[21] Y. Liu, C. Xu, and S. Cheung. Afchecker: Effective model checking for context-aware adaptive applications. *J. Syst. Softw.*, 86(3):854 – 867, 2013.

[22] H. Lu, W. K. Chan, and T. H. Tse. Testing context-aware middleware-centric programs: a data flow approach and an rfid-based experimentation. In *Proc. FSE' 06*, pages 242–252, Portland, USA, 2006.

[23] P. McKinley, S. Sadjadi, E. Kasten, and B. H. C. Cheng. Composing adaptive software. page 56ÍC64 Vol. 37. IEEE Computer, 2004.

[24] I. Park, D. Lee, and S. Hyun. A dynamic context-conflict management scheme for group-aware ubiquitous computing environments. In *Proc. COMPSAC' 05*, pages 359–364 Vol. 2, Kyoto, Japan, July 2005.

[25] K. Pohl. Requirements engineering: Fundamentals, principles, and techniques. Springer Publishing Company, Incorporated, 2010.

[26] A. Ramirez, A. Jensen, and B. H. C. Cheng. A taxonomy of uncertainty for dynamically adaptive systems. In *Proc. SEAMS' 12*, pages 99–108, Zurich, Switzerland, June 2012.

[27] A. J. Ramirez, A. C. Jensen, B. H. C. Cheng, and D. B. Knoester. Automatically exploring how uncertainty impacts behavior of dynamically adaptive systems. In *Proc. ASE' 11*, pages 568–571, Kansas, USA, Nov 2011.

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

[29] M. Sama, D. S. Rosenblum, Z. Wang, and S. Elbaum. Model-based fault detection in context-aware adaptive applications. In *Proc. FSE' 08*, pages 261–271, Atlanta, USA, Feb 2008.

[30] T. H. Tse and S. Yau. Testing context-sensitive middleware-based software applications. In *Proc. COMPSAC' 04*, pages 458–466 vol.1, Hong Kong, China, Sept 2004.

[31] Q. Wang. Towards a rule model for self-adaptive software. *SIGSOFT Softw. Eng. Notes*, 30(1):8–, Jan 2005.

[32] Z. Wang, S. Elbaum, and D. Rosenblum. Automated generation of context-aware tests. In *Proc. ICSE' 07*, pages 406–415, Minneapolis, USA, May 2007.

[33] D. Weyns, M. U. Iftikhar, D. G. de la Iglesia, and T. Ahmad. A survey of formal methods in self-adaptive systems. In *Proc. C3S2E' 12*, pages 67–79, Montreal, Canada, June 2012.

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

[35] C. Xu and S. C. Cheung. Inconsistency detection and resolution for context-aware middleware support. In *Proc. Joint ESEC/FSE' 05*, pages 336–345, Lisbon, Portugal, Feb 2005.

[36] C. Xu, S. C. Cheung, W. K. Chan, and C. Ye. On impact-oriented automatic resolution of pervasive context inconsistency. In *Proc. Joint ESEC/FSE' 07*, pages 569–572, Dubrovnik, Croatia, Mar. 2007.

[37] C. Xu, S. C. Cheung, W. K. Chan, and C. Ye. Partial constraint checking for context consistency in pervasive computing. *ACM Trans. Softw. Eng. Methodol.*, 19(3):9:1–9:61, Feb 2010.

[38] C. Xu, W. Yang, X. Ma, C. Cao, and J. Lu. Environment rematching: Toward dependability improvement for self-adaptive applications. In *Proc. ASE' 13*, pages 592–597, Palo Alto, USA, Nov 2013.

[39] W. Yang, C. Xu, and L. Zhang. Idea: Improving dependability for self-adaptive applications. In *Proc. of the 2013 Middleware Doctoral Symposium*, pages 1:1–1:6, Beijing, China, Dec 2013.

[40] J. Zhang and B. H. C. Cheng. Model-based development of dynamically adaptive software. In *Proc. ICSE' 06*, pages 371–380, Shanghai, China, May 2006.