Efficient Validation of Self-adaptive Applications by Counterexample Probability Maximization

Wenhua Yang[a,b], Chang Xu[a,b,*], Minxue Pan[a], Chun Cao[a,b], Xiaoxing Ma[a,b], Jian Lu[a,b]

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

Abstract

Self-adaptive applications' executions can be affected by uncertainty factors like unreliable sensing and flawed adaptation and therefore often error-prone. Existing methods can verify the applications suffering uncertainty and report counterexamples. However, such verification results can deviate from reality when the uncertainty specification used in verification is itself imprecise. This thus calls for further validation of reported counterexamples. One outstanding challenge in counterexample validation is that the probabilities of counterexamples occurring in real environment are usually very low, which makes the validation extremely inefficient. In this paper, we propose a novel approach to systematically deriving path-equivalent counterexamples with respect to original ones. The derived counterexamples guarantee to have higher probabilities, making them capable of being validated efficiently in field test. We evaluated our approach with real-world self-adaptive applications. The results reported that our approach significantly increased counterexample probabilities, and the derived counterexamples were also consistently and efficiently validated in both real environment and simulation.

Keywords: Self-adaptation, Validation, Optimization, Probability

*Corresponding author
Email addresses: ywh.nju@outlook.com (Wenhua Yang), changxu@nju.edu.cn (Chang Xu), mxp@nju.edu.cn (Minxue Pan), caochun@nju.edu.cn (Chun Cao), xxm@nju.edu.cn (Xiaoxing Ma), lj@nju.edu.cn (Jian Lu)

1. Introduction

Self-adaptive applications have been capturing increasing attention recently because of their capabilities to adjust their behavior in response to their perception of environment [20, 25, 65]. However, developing high-assurance self-adaptive applications needs nontrivial effort, since developers have to guarantee the correctness of such applications' predefined business logics while considering the applications' interactions with environment in presence of uncertainty [25, 33, 40]. As a result, self-adaptive applications are error-prone [72, 48, 70], which makes checking the correctness of self-adaptive applications vitally important and significant. Verification, which is regarded as effective methods to achieve a dependable self-adaptive application, can provide evidence that the set of stated functional and non-functional properties are satisfied during the application's operation [71, 7]. However, verification results themselves can be imprecise when uncertainty is not appropriately considered. This thus calls for further validation of the verification results.

There are many ways to obtain verification results for self-adaptive applications. Recent promising approaches, developed by others and us, have proposed to verify self-adaptive applications against various properties (e.g., stability and reliability) [66, 9, 15, 35, 50, 76]. The verification techniques in article [66] explore an application's state space to detect faults, yet they can report numerous false positives that can never happen in the application's running environment, as the model may over-approximate the behavior of the application, while [50, 76] eliminate the false positives by leveraging environmental constraints. However, most of them did not address the orthogonal but crucial issue of uncertainty in verifying self-adaptive applications. Uncertainty is the gap between certainty and what is currently known [33, 31, 6].

According to existing literature [33], uncertainty can be categorized as aleatory or epistemic. Aleatory uncertainty captures the uncertainty that is caused by randomness and is usually modeled using probabilities, while epistemic uncertainty corresponds to the lack of knowledge and it is usually not possible to

represent this kind of uncertainty as a probability distribution. For example, oftentimes the user's uncertainty in specifying system's requirements and objectives can be considered as epistemic, since it is due to the lack of knowledge, and not randomness. In self-adaptive applications, uncertainty arises naturally when the applications interact with the environment through inevitably imprecise sensors or flawed actors [60, 32]. This type of uncertainty is deemed to be aleatory, since it is due to randomness. In paper [76] we proposed to model a self-adaptive application with Interactive State Machine which is a variant of Finite State Machine and exploit an SMT solver [26] to verify the State Machine with uncertainty modeled by approximated error ranges and distributions. Specifically, we depict the application's failures by assertions (failure conditions) and collect the application's behavior by enumerating each candidate path within a bound of path length in the State Machine to get a path condition, in which environment-related variables are augmented with error ranges to include uncertainty. The failure condition and the path condition are then joined and checked for satisfiability. If satisfied, it means the application could fail, and one concrete solution will be returned by the solver. The solution contains an assignment of values for the variables in the failure and path conditions and specifies the environment settings sensed by the application and the application's internal states that causes the application failure. As this path and the corresponding solution give an example of an application failure which counters the quality requirement, they are together called a *counterexample*.

While the verification results of self-adaptive applications are informative, they can deviate from reality because of the inaccurate specification of uncertainty [33, 28]. This stresses the need for further validation, especially for critical applications [78]. The accuracy of verification results has a significant impact on the debugging process, since imprecise results would mislead developers' attentions. Less important counterexamples could be falsely highlighted, or false counterexamples could be reported. To refine the verification results, validation is a natural and widely-adopted approach. Simulation has been a successful validation approach in many cases [5, 4], however, its power is limited for self-

adaptive applications, since an imprecise uncertainty specification can result in an inaccurate simulated environment. Thus, in order to validate the verification results (i.e., counterexamples), one has to run the application in real deployment under the environment setting realizing the conditions specified by the counterexample, and observe whether the application follows the path specified by the counterexample. This process is labor-intensive and time-consuming. It is often very difficult to force the application to execute as specified by the counterexample. When validating a self-adaptive application, sensing variables' values are difficult to control because they depend on the environmental attributes sensed from its environment setting. It is practically infeasible to continually change the environment setting, when the application is running, to make the concerned variables take specific values. Furthermore, the sensing will be affected by uncertainty such as unpredictable noises in its environmental sensing. If one tried to directly set the error values caused by uncertainty, the validation would be meaningless, since these chosen values may not even exist in a real environment. Thus, in order to validate the verification results (i.e., counterexamples), one has to run the application in real deployment, and may have to repeatedly run the application until the counterexample is witnessed to be a real failure. In other words, there is a probability that a counterexample will happen in the real deployment.

The probability has a significant effect on the validation cost for a counterexample. For instance, if a counterexample's probability is 0.001, in theory, we may need to run the application in real deployment under this counterexample's setting for about 1,000 times to observe an occurrence of the counterexample. This could require a considerable investment. For example, for our experiment subject of a robot-car application, to validate one behavior that only contains ten adaptation steps costs about one minute averagely. Besides, SMT solvers only give counterexamples that satisfy the failure conditions but not guarantee their probabilities, so it is possible many counterexamples' probability values are small, according to the studies in [76]. It will definitely hinder efficient validation of self-adaptive applications. To expedite the process and mitigate the

validation cost, in this paper we propose to efficiently validate a self-adaptive application's counterexample by finding a path-equivalent counterexample with a higher probability. Counterexamples are path-equivalent if they comprise the same application path but different solutions of the path conditions. With a high probability counterexample, it will be more likely for the application to follow the counterexample's specified path and fail. This can save much time in validation. As shown in our experiment, the searching for a high probability counterexample only costs several minutes, which is just a small fragment compared with the validation time saved by our approach.

For a self-adaptive application's counterexample, the root cause of its probability is uncertainty, since uncertainty can affect the application's sensing and adaptation, which further impact the application's internal states. Uncertainty that arises from sensing and adaptation can be specified by error ranges and distributions [33, 76], and with this information, we first propose to formulate a counterexample's probability into a function. The inputs of the function include variables reflecting the environmental sensing, adaptation, application states and uncertainty. Apparently their values cannot be arbitrary, but on the contrary, subject to constraints (e.g., within some ranges). With this probability function, to find a higher probability counterexample, we just need to maximize the function, i.e., finding a set of inputs such that the probability function can reach its maximum value. Thus, the problem of finding a path-equivalent counterexample with a higher probability is reduced to a constrained optimization problem. To alleviate the cost of solving the optimization problem, we propose to separate the optimization problem into several sub-problems. These smaller problems require less solving cost; and what's more, their results can be reused when they appear in other counterexamples' optimization problems. Furthermore, for each (sub-)problem, we devise a method to simplify its objective function for easier computation. We then leverage the genetic algorithm to solve the problem. We show that by maximizing each counterexample's probability, self-adaptive applications' verification results can be validated in real deployment more efficiently. The efficiency is two-fold— counterexamples can

be validated more efficiently provided that uncertainty is correctly modeled, and imprecise uncertainty modeling can also be exposed faster otherwise. We conducted a series of experiments on real applications and the results confirmed our approach's effectiveness and efficiency. The main contributions of this paper are:

- We propose a novel approach to improving the efficiency of validating verification results for self-adaptive applications by finding high-probability counterexamples. By exploiting the probability function of the counterexample, the problem is reduced to a constrained optimization problem.
- We leverage multiple techniques to save the solving cost of the constrained optimization problem: a technique to reduce the problem into smaller subproblems, a technique to simplify the objective function, and a genetic algorithm based technique to effectively solve the problem.
- We implement all concerned algorithms in a prototype tool and evaluate our validation approach on self-adaptive applications with both real and simulation experiments.

The remainder of this paper is organized as follows. Section 2 presents some basic notions concerning self-adaptive applications and uncertainty thereof. Section 3 uses a motivating example to explain the inadequacy of existing work and motivate our work. Section 4 presents our validation approach in detail. Section 5 evaluates our approach with self-adaptive applications. Section 6 discusses related work, and finally Section 7 concludes this paper.

2. Self-adaptive Applications Suffering Uncertainty

Self-adaptation endows an application with the ability to satisfy certain objectives by automatically modifying its behavior based on the environment at runtime [20, 33]. However, the ever-growing complexity of applications is challenging the development of self-adaptive applications, since self-adaptation can manifest itself in different forms, such as web applications [55] that can adapt to

changing load or robotics designed for distributed search and rescue [32]. The latter falls in the class of self-adaptive applications that interact with physical environment, which can be characterized with Brun et al. [12]'s widespread and notable feedback loop: a self-adaptive application first senses its environment, then makes decisions according to its predefined logics, and lastly adapts to the environment by actuators.

For a self-adaptive application that interacts with physical environment, we formalize its running process for a better and more precise understanding, and therefore facilitate the subsequent verification and validation. A self-adaptive application P is defined as a tuple (V, S, L, A). V is a set of variables where $V = V_s \cup V_n$ $(V_s \cap V_n = \emptyset)$. 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*. S is an assignment which assigns the sensing variables with values provided by sensors. The application reacts to the sensed environment according to its predefined logics specified by L, which is a function $L : V^{|V|} \to A$. The function L contains a set of conditions, and evaluates the conditions with the sensed environment (values stored in V_s) and the application internal state (values stored in V_n). Then it chooses an adaptation $a \in A$ based on the evaluation. The adaptation can further change the environment and affect the application's next environmental sensing, which forms an adaptation loop. Thus, the running of the application can be specified as an (infinite) sequence $(S_0, L, a_0) \to (S_1, L, a_1) \to \cdots \to (S_n, L, a_n) \to \cdots$.

It is a fact that uncertainty is prevalent in self-adaptive applications [33, 60, 32]. Commonly, uncertainty is described as the gap between certainty and what is currently known [60, 6]. There have been many attempts to clarify and classify uncertainty [33, 31, 60]. However, there is a multitude of sources for uncertainty, and not all sources have similar characteristics [33]. One of the major sources is uncertainty caused by unreliable sensing or flawed adaptation. It affects values of variables in self-adaptive applications, and therefore impacts their executions. Previous studies [76, 32] suggested that this kind of uncertainty demonstrates

patterns that can be modeled with error ranges and distributions. With this information, the applications can be verified to see whether they will fail when suffering uncertainty.

The verification process can be concisely summarized as follows. First, for a self-adaptive application, we build its Interactive State Machine (ISM) from the application code [76]. As a variant of finite-state machine [7], an ISM also contains states and transitions. In the states, the application performs environmental sensing. Transitions correspond to the application's logic decisions and adaptations. Each transition is associated with a *condition* and *actions*. The *condition* is evaluated on the transition's source state, which includes the values of the variables representing environmental sensing. The *actions* are used to explicitly model an application's adaptation and its effects on the environment. As we can see, the update from one state to another via a transition models exactly one adaption (S, L, a) of a self-adaptive application, which makes the ISM naturally suitable for self-adaptive application modeling. Then, in order to verify the application, we check whether there exists a path in its ISM such that the application fails or does not meet a required property. Since the length and number of paths can be infinite, a bound is required for the path length. Finding such a path is equivalent to finding a set of values such that all the conditions along the path, which is called a path condition, together with a failure condition can be satisfied. The values related to environmental interaction are affected by uncertainty. To incorporate uncertainty, for each affected variable v whose error range is $[a, b]$, we use a new variable v' to represent v with uncertainty, and v' satisfies the constraint $v + a \leq v' \leq v + b$. This means that the value of v' can range from $v+a$ to $v+b$. Here, v contains the actual value in the environment setting, and v' contains the value sensed by the application affected by uncertainty. By replacing v with v', we embrace uncertainty and get a new path condition. The verification is performed on the new path condition, and if a solution is found, the corresponding path with the solution of the constraints is called a *counterexample*. An example illustrating this verification process is presented in the next section.

Counterexample. A counterexample is a tuple $t = (\sigma, K)$, in which σ is a path and K is an assignment $K : V \rightarrow E$, where V is a set $\{v_0, v_1, \ldots, v_n\}$ containing all variables of σ and E is a set of values $\{e_0, e_1, \ldots, e_n\}$ such that $K(v_i) = e_i$ $(0 \leq i \leq n)$ is in the solution of the path and failure condition.

In a counterexample $t = (\sigma, K)$, σ is the path chosen for verification. K contains both values to the variables reflecting the actual environment (the unprimed variables) and the application's sensed environment (the primed variables), respectively. In theory, a counterexample contains enough information for an application behavior and its corresponding environment. The values of the unprimed variables correspond to the real environment, so from these values we can construct in reality an environment setting, in which verification results are validated. The values of the primed variables are sensed values, based on which the application can adapt and execute. However, in the real environment, when executing the application, the sensed values come from sensors are affected by uncertainty. One cannot control their values and therefore, is not capable to force the application to exhibit exactly the desired behavior, which results in a probability to witness a counterexample's triggering under its environment setting. In this work, for the obtained counterexamples, we propose an approach to finding path-equivalent ones with higher probabilities to increase the chances that counterexamples to be witnessed in validation. Counterexamples $t = (\sigma, K)$ and $t' = (\sigma', K')$ are path-equivalent if and only if $\sigma = \sigma'$ and $K \neq K'$.

3. Motivating Example

We now provide an example that illustrates how the verification is conducted. The verification results will present the challenges of validating self-adaptive applications, which motivates our work.

Consider the simplified code in Figure 1(a). It is a fragment of a self-adaptive robot-car application that explores an unknown area by the car's built-in ultrasonic sensors in four directions [76, 73, 77]. The application controls the

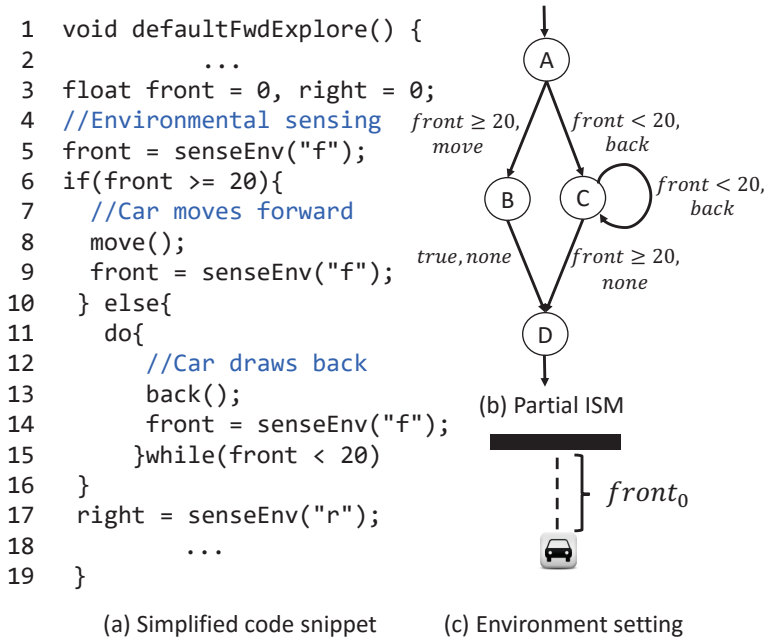


Figure 1: Robot-car application example.

robot-car to sense the environment by method `senseEnv(String dir)`, where the parameter `dir` specifies the current sensing direction. Based on the sensed information, the application will make adaptations, such as moving forward or drawing back. Figure 1(b) shows the corresponding ISM. In state A an environmental sensing (Line 5) is performed. The branch from A corresponds to the `if`-statement in Line 6 and the `else`-statement in Line 10. The *condition* and *actions* are associated with the transitions, which represent different logic decisions and the chosen adaptations. Note that some conditions are deduced from others following the axiom rules of Floyd-Hoare logic [46]. For example, the transition from C to D is the end of the loop. The loop condition is $front < 20$, so for the loop to end, the negation of the loop condition is added to the transition.

In a self-adaptive application, uncertainty caused by environmental interactions affects the variables' values, which needs to be considered during the verification. For example, to check the path $A\ C\ D$, we should first find inputs

that satisfy its path condition: " $front_0 < 20 \wedge front_1 \geq 20$ ". Here, $front_0$ and $front_1$ represent sensed values between the robot-car and its front obstacles at different time points returned by method `senseEnv(String dir)`. We know that $front_0$ and $front_1$ will be affected by uncertainty. Assume that distributions and error ranges of $front_0$ and $front_1$ are both Gaussian distributions and $[-5, 5]$. To embrace uncertainty, we introduce a new variable $front'_0$ to represent $front_0$ with uncertainty, and make $front_0$ represent the actual distance. Derived from the error range, the two variables satisfy an additional constraint $front_0{-}5 \leq front'_0 \leq front_0{+}5$. In the path condition, variables' values are obtained by application sensing, which has uncertainty involved. Thus, we replace the variables affected by uncertainty in the path condition with their primed versions, and get a new path condition "$front'_0 < 20 \wedge front'_1 \geq 20$". Additionally, the adaptation can change the environment and affect the application's next environmental sensing [76]. Correspondingly, the execution of the program, e.g., methods `move()` and `back()`, can have side-effects and change the values of the variables. In the example, because of the effect of adaptation `back()`, there is a constraint between $front_0$ and $front_1$ that $front_1$ should be equal to the sum of $front_0$ and the distance that the robot-car has drawn back (e.g., 10cm). The new path condition, and the additional constraints together form a constraint "$front'_0 < 20 \wedge front'_1 \geq 20 \wedge front_1 = front_0 + 10 \wedge front_0 - 5 \leq front'_0 \leq front_0 + 5 \wedge front_1 - 5 \leq front'_1 \leq front_1 + 5$".

We utilize Z3 [71], an SMT solver, to solve the above constraint to obtain a solution. The values in the solution for $front_0$, $front'_0$, $front_1$, and $front'_1$ can be 24, 19, 34 and 30, respectively. For example, the value of $front_0$ specifies the actual initial distance between the car and its front obstacle, and $front'_0$ is the sensed initial distance. Based on the value of unprimed variable $front_0$, a partial initial environment setting can be simply constructed as illustrated in Figure 1(c). Ideally, without uncertainty the application would strictly follow this path since all the path conditions would be satisfied. However, uncertainty can cause some variables to acquire imprecise values which can result in that the values of the primed variables change from time to time and some conditions become

unsatisfied. Thus, the application follows this path at a probability, which is the likelihood of the path condition's satisfaction under this environment setting. Now let us show how to calculate the probability with the example. For the former part of the path condition: $front'_0 < 20$, the actual distance between the car and its front obstacle is 24. From statistical experiments, we have that $front'_0$ satisfies a Gaussian distribution, with a variance of 2^2, and a mean of 24 (the value of $front_0$). Furthermore, we need to set up an error range to restrict the value of $front'_0$ for the purpose of verification, since $front'_0$ without an error range can be an arbitrary value (e.g., a very large number with an extremely low probability of occurrence) and makes the verification useless by reporting counterexamples at any conditions. In this example, we set up the error range as $[-5, 5]$, so with a mean of 24 the value range of $front'_0$ is $[19, 29]$, which covers 98.76% of all the possible values of $front'_0$. The probability of $front'_0$ being less than 20 can be calculated by

$$\int_{19}^{20} \frac{1}{2\sqrt{2\pi}} e^{-\frac{(front'_0-24)^2}{8}} \, \mathrm{d}front'_0 = 0.0166.$$

The variable $front'_1$ has the same error range and distribution, so for the latter part of the path condition "$front'_1 \geq 20$", we can similarly compute its probability, which is 0.9876. The overall probability for the application to follow path $A\ C\ D$ under this environment setting is $0.0166 \times 0.9876 = 0.0164$. Statistically, validating this path in real deployment needs about 61 times ($1/0.0164 = 60.98$) of running the application, which requires that a considerably amount of labor and time be invested in the whole validation process. To save the validation effort, a high probability counterexample is expected, because it is more likely to happen. Next, we will describe our efficient validation approach that finds path-equivalent counterexamples but with high probabilities.

4. Efficient Validation Approach

In this section we present our approach to validating self-adaptive applications. We begin with an overview of the approach, followed by a fundamental

introduction to the optimization theory and a detailed presentation of our approach.

4.1. Overview

Our validation approach takes as inputs the counterexamples of a self-adaptive application and the uncertainty model used in verifying the application. As discussed earlier, a counterexample has a probability of occurrence because of the uncertainty in the sensing inputs. To increase the efficiency of validation, we propose to obtain a path-equivalent counterexample with the highest probability. The validation process is mainly composed of four steps. First, we formulate a probability function for calculating the counterexample's probability. This turns our goal into how to obtain the inputs that make the probability function reach its maximum value. Meanwhile, the probability function should also satisfy certain constraints when it is maximized, so secondly we propose to model related constraints. These constraints can be about the value ranges of inputs or enforced by physical laws, e.g., an obstacle in the robot-car application cannot move arbitrarily to anywhere at anytime in practice. With these efforts, we can reduce this problem to a constrained optimization problem whose objective function is the probability function. Thirdly, to further improve the efficiency of solving the optimization problem, we propose multiple techniques to divide the problem into smaller optimization sub-problems and simplify the objective function's complexity. Finally, for the constrained optimization problem we employ the genetic algorithm to search a set of feasible parameters (i.e., inputs of the probability function) that maximize the probability function under the constraints. Figure 2 manifests the high-level description of our approach. After the process, we can obtain counterexamples with higher probabilities and thus make their validation more efficient.

4.2. Optimization Theory

We recall some fundamental notions of optimization theory, which is one of the theories that support our approach. For an extensive exposition, interested readers could refer, for example, to [62].

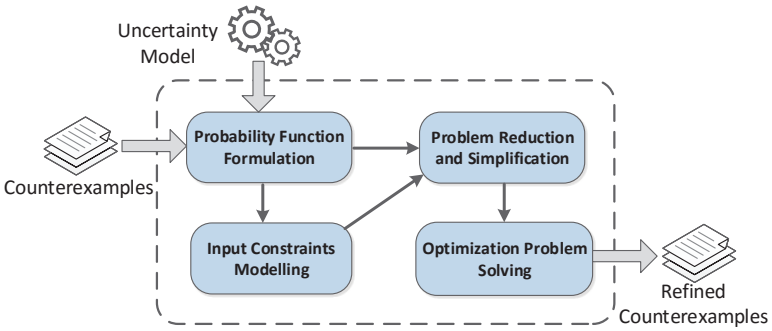


Figure 2: Approach overview.

In mathematics and computer science, an optimization problem is about finding the best solution from all feasible ones, i.e., to find a vector of parameters $\mathbf{X} = [X_1, X_2, \ldots, X_n]$ that can maximize or minimize an objective function $f(\mathbf{X})$. The function $f(\mathbf{X})$ to be optimized can be subject to constraints in the form of equality constraints, $G_i(\mathbf{X}) = 0$ $(i = 1, \ldots, m_e)$, and inequality constraints, $G_i(\mathbf{X}) \leq 0$ $(i = m_e + 1, \ldots, m)$. The solution of the optimization problem includes a scalar value returned by $f(\mathbf{X})$ and values of each element in $\mathbf{X}$.

Constrained optimization problem has been studied for a long time. An efficient and accurate solution to this problem depends not only on the scale of the problem in terms of the number of constraints and variables, but also on characteristics of the objective function and constraints [1, 62]. When both the objective function and the constraints are linear functions of the variables, the problem is known as a *Linear Programming problem* and reliable solution procedures are readily available. Even for a *Nonlinear Programming problem* in which the objective function and constraints can be nonlinear functions of the variables, several proposed algorithms can still perform quite well, such as genetic algorithm and particle swarm optimization algorithm [62, 54].

4.3. Probability Function Formulation

A counterexample's probability of occurrence is due to the fact that its corresponding path condition has a probability of satisfaction under the counterex-

ample's environment setting, since the environmental sensing can be affected by uncertainty. When executing the application to validate a counterexample in real deployment (e.g., physical environment), sensing variables' values are difficult to control because they depend on the environmental attributes sensed in its environment setting. It is practically infeasible to continuously change the environment setting when the application is running to make sensing variables take specific values. Furthermore, the sensing of these variables will be affected by uncertainty. The sensed values often fall into error ranges, with distributions determined by physical characteristics of sensing technologies [33, 76]. Some of the values in the error ranges satisfy the path condition, while others not. For a counterexample $t = (\sigma, K)$ under validation, it has a probability over the range of $(0, 1)$. The probability is defined as the likelihood for the application's execution to follow the path σ, which means that σ's path condition is satisfied when the application is running.

We now present how to formulate a counterexample's probability function. Given a counterexample $t = (\sigma, K)$, let PC be the σ's path condition with uncertainty considered. In general, the probability function PF that we aim to obtain can be formally denoted as:

$$PF = \int_D \mathbb{1}_{PC}(\mathbf{X}) \cdot p(\mathbf{X}) \tag{1}$$

where D is the input domain defined as the Cartesian product of domains of variables in the path condition PC, $p(\mathbf{X})$ is the probability density function of an input $\mathbf{X}$ that specifies the distribution of $\mathbf{X}$'s value in its error range, and $\mathbb{1}_{PC}(\mathbf{X})$ is the indicator function on PC that returns 1 when $\mathbf{X}$ satisfies PC, and 0 otherwise.

Among variables in input $\mathbf{X}$, some are non-sensing variables and some are sensing variables. The counterexample's probability is irrelevant to the values of non-sensing variables, since they are not affected by uncertainty. Thus, we replace the non-sensing variables in Equation 1 with their values in the counterexample, and only focus on the sensing variables. Still, it can be expensive

to directly calculate Equation 1 as a whole. We devised a method to divide Equation 1 into smaller parts and calculate them separately and then merge the results. Prior to this, remind that as discussed in Section 2, the execution of a self-adaptive application can be modeled as a sequence of transitions, so the path condition PC of σ is a conjunction of the transitions' conditions (i.e., $TC_1 \wedge TC_2 \wedge \cdots \wedge TC_n$). Based on this observation that conjunction plays a major role in composing the path condition, we present the following theorem, which divides two conjunctive conditions.

Theorem 1. *Let F be a probability function in the form of $\int_D \mathbb{1}_C(\boldsymbol{X}) \cdot p(\boldsymbol{X})$, where D, $\mathbb{1}_C(\boldsymbol{X})$ and $p(\boldsymbol{X})$ are interpreted the same as in Equation 1. Suppose $C = C_1 \wedge C_2$, and $\mathbf{X_1}$ and $\mathbf{X_2}$ contain exactly all the variables appeared in C_1 and C_2, respectively. If there is no variable x in both $\mathbf{X_1}$ and $\mathbf{X_2}$, then*

$$F = \int_{D_1} \mathbb{1}_{C_1}(\mathbf{X_1}) \cdot p(\mathbf{X_1}) \times \int_{D_2} \mathbb{1}_{C_2}(\mathbf{X_2}) \cdot p(\mathbf{X_2})$$

where D_1 and D_2 are the Cartesian product of domains of variables in $\mathbf{X_1}$ and $\mathbf{X_2}$, respectively.

The proof of Theorem 1 is straightforward and omitted here. It follows that the calculation of a probability function PF can be reduced to the calculation of two smaller sub-functions. Note that clause C_1 and C_2 can be further divided into sub-clauses if they also consist of conjunctive clauses. Therefore, we can repeatedly divide the clause and apply Theorem 1 to rewrite the probability function, until no clause is dividable.

Now to calculate probability function PF, we just need to calculate every sub-function $\int_D \mathbb{1}_C(\mathbf{X}) \cdot p(\mathbf{X})$ and merge the results. Since we have replaced the non-sensing variables with concrete values, the variables in $\mathbf{X}$ are all sensing variables including uncertainty, i.e. the primed variables. As mentioned earlier, the value of each sensing variable v' is affected by uncertainty, which is the result of the value of its corresponding unprimed variable v plus an error within the error range $[a, b]$. So function $\int_D \mathbb{1}_C(\mathbf{X}) \cdot p(\mathbf{X})$ ($\mathbf{X} = [v'_1, v'_2, \ldots, v'_j]$) can be rewritten into a more concrete form of Equation 2. Note that the in-

tegral variables are primed variables and the result would be a function over the unprimed variables, since primed variables have been substituted by the unprimed variables in the integral upper and lower limits. The unprimed variables correspond to the environment, so given an environment setting we can tell a counterexample's probability with Equation 2.

$$\int_{v_1+a_1}^{v_1+b_1} \cdots \int_{v_j+a_j}^{v_j+b_j} \mathbb{1}_C(v'_1, \cdots, v'_j) \cdot p(v'_1, \cdots, v'_j) \mathrm{d}v'_1 \cdots \mathrm{d}v'_j \tag{2}$$

Example. Let us take the robot-car application as an example again (Figure 1). Assume that the path segment $A\ C\ D$ is part of a counterexample's path. As discussed in Section 3, after considering uncertainty, the path condition PC of this segment is "$front'_0 < 20 \wedge front'_1 \geq 20$". Since the two clauses "$front'_0 < 20$" and "$front'_1 \geq 20$" share no common variables, PC can be divided into two parts. Suppose that the error ranges and distributions of $front'_0$ and $front'_1$ are both $[-5, 5]$ and Gaussian distribution. Then, the probability that the application executes by following the path segment $A\ C\ D$ can be calculated as:

$$\begin{array}{l} \int_{front_0-5}^{front_0+5} \mathbb{1}_{(front'_0<20)} \cdot \frac{1}{2\sqrt{2\pi}} e^{-\frac{(front'_0-front_0)^2}{8}} \mathrm{d}front'_0 \times \\ \int_{front_1-5}^{front_1+5} \mathbb{1}_{(front'_1\geq 20)} \cdot \frac{1}{2\sqrt{2\pi}} e^{-\frac{(front'_1-front_1)^2}{8}} \mathrm{d}front'_1. \end{array}$$

4.4. Input Constraints Modeling

The probability function formulated in the last phase is the objective function to be maximized. Meanwhile, the probability function is subject to constraints concerning its inputs, i.e., unprimed variables. Thus, the input constraints are just about the unprimed variables. Note that it is not necessary to constrain the primed variables, since from the mathematical view, they do not appear in the final form after the transforming of the probability function; and from the practical view, when the unprimed variables are constrained, the primed variables are also constrained by error ranges.

Among the constraints, some are used to limit the value ranges of inputs,

e.g., an input is required to be greater than 0. In practice, many variables are restricted by this kind of constraints. However, we noticed that besides such simple constraints, there are also other constraints including the *application-specific* or *environment-specific* constraints that must be satisfied by the application. We need to pay extra efforts to obtain such constraints, since they are often implied by the self-adaptive application and its environment.

Before explaining how to model application-specific input constraints, we need to know that self-adaptive applications are different from traditional software for their tight interactions with environment. These applications continuously sense their environment and make adaptation upon certain environmental changes, and the adaptation can further change the environment and affect the next environmental sensing. Since the new environment is the result of the adaptation's effects on the previous one, it cannot be arbitrary. Otherwise the constraints predefined by the application domain could be violated. For example, the car runs in a physical world domain, so it must subject to physical laws. Suppose that the car is moving forward with an obstacle ahead. If the environment after each movement was treated independently and could be arbitrary, we can have the distance between the car and its front obstacle before a forward movement larger than the one after the forward movement, which clearly contradicts physical laws.

To model application-specific input constraints, we first formulate them as abstract constraints. An *abstract constraint*, which is a mathematical representation of an application-specific constraint, behaves as an equality constraint $G(\mathbf{x_i}, \mathbf{x_{i+1}}, \mathbf{y}) = 0$ or an inequality constraint $G(\mathbf{x_i}, \mathbf{x_{i+1}}, \mathbf{y}) \leq 0$. The vector $\mathbf{x_i} = [x_{i_1}, \ldots, x_{i_n}]$ and $\mathbf{x_{i+1}} = [x_{i+1_1}, \ldots, x_{i+1_n}]$ are the unprimed variables representing the actual environment before and after the i-th adaptation, respectively, and $\mathbf{y} = [y_1, \ldots, y_m]$ are the non-sensing variables representing the effects of the adaptation. Consider our robot-car application, and suppose that the car moves forward for a distance a. Then from physical laws one knows that the new distance in the front x_{i+1} should be the previous distance in the front x_i minus the distance a, i.e., $x_{i+1} = x_i - a$, which can be transformed

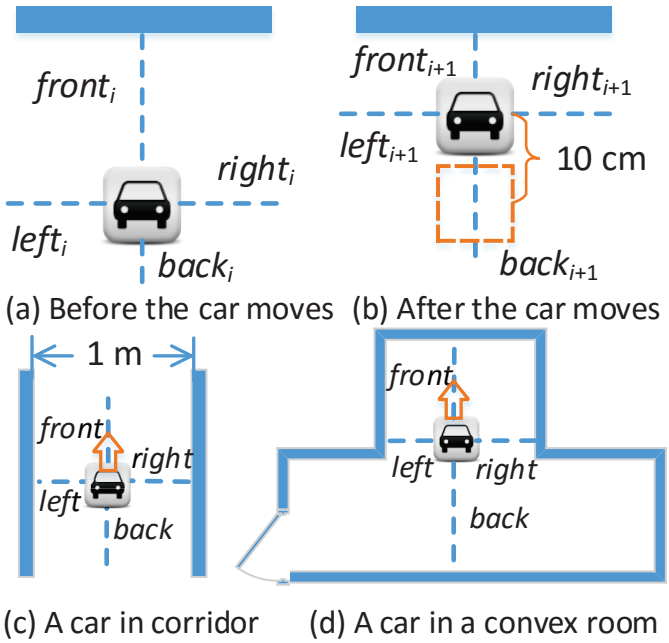


Figure 3: Examples of input constraints.

into $x_i - x_{i+1} - a = 0$. The abstract constraint is instantiated for every application adaptation that is subject to the constraint. Figure 3(a) and (b) show an example, in which every time the car moves forward, it moves for a distance of 10cm. Its distance to the front obstacle before the movement is stored in variable $front_i$, and the one after the movement is in the variable $front_{i+1}$. By applying the abstract constraint, we get $front_i - front_{i+1} - 10 = 0$.

Application-specific constraints are derived from knowledge of an application, and will hold in all environments in which the application runs. Environment-specific constraints, on the other hand, hold only for some particular environment. For example, in Figure 3(c), the robot-car moves in a narrow corridor. The width of the corridor is 1m (100cm), so there is a constraint that $left + right = 100$. However, in a different environment illustrated in Figure 3(d), this constraint no longer holds. Users can model input constraints based on characteristics of an application and its environment.

4.5. Simplification and Optimization

In this section, we explain how to solve the constrained optimization problem whose objective function and constraints are acquired in the above two phases, respectively. As we know, the solving cost of the constrained optimization problem grows quickly as the complexity of the objective function and the constraints increases. To alleviate the solving cost, we propose three techniques in solving the problem. First, an optimization problem separation technique is employed to divide the entire problem to several sub-problems, each of which consists of a smaller objective function and less constraints. Then this smaller objective function is further simplified with a technique of indicator function elimination, which removes the indicator function and transforms the objective function to an equivalent form for easier computation. Finally, each reduced and simplified sub-problem is solved by a customized genetic algorithm, and the results are merged to produce the solution of the original optimization problem, which corresponds to the highest probability path-equivalent counterexample.

Optimization problem separation. The inputs of the problem are the objective function PF and a set of constraints C. Solving a constrained optimization problem could be a costly task if the objective function is intricate and the constraints are complex [62]. To save the cost of solving the optimization problem, we provide *optimization problem separation* to reduce complexity by dividing PF and C into independent parts, each of which forms an optimization sub-problem. These problems are smaller in size and in turn, require less solving cost. The other motivation behind *optimization problem separation* is that we frequently observed semantically equivalent optimization sub-problems among the constrained optimization problems. Indeed, a good number of counterexamples's paths of the same self-adaptive application have some overlapping or similar parts, which often share semantically equivalent path conditions. Thus, those optimization sub-problems' results can be reused when they appear in other counterexamples.

To separate a constrained optimization problem to several smaller sub-problems, we need to separate its constraints and the objective function (note

that the objective function has been rewritten as the arithmetic computation of multiple indivisible sub-functions, following the method presented in Section 4.3). The separation is based on a relation defined on the unprimed variables. We call two unprimed variables are related, if they satisfy either or both the following two conditions: (1) the two variables are in the integral limits of the same indivisible sub-function of the objective function; and (or) (2) the two variables are in the same constraint. Clearly this relation is an equivalence relation, which divides the variables into several equivalent classes. Then for each class of variables, the sub-functions whose integral limits contain the variables from this class are grouped and multiplied as an objective function, together with the constraints which contain the variables from the same class, forms an optimization sub-problem. In this way, the original optimization problem is separated into several sub-problems.

Algorithm 1 depicts the separation process based on the equivalence relation. It starts with the separation of the unprimed variables, which is later used to guide the separation of the objective function and the constraints. The input is the entire optimization problem P, from which the algorithm obtains the set of unprimed variables, the set of indivisible sub-functions and the set of constraints (Line 2-4). The separation is an iterative process, each of which finds one sub-problem (Line 5-34). The essential data structure of the algorithm is the set V_s, which is used to store unprimed variables of the sub-problem. In each iteration, initially it contains only one variable that is randomly picked (Line 9). Then the algorithm repeatedly fills V_s with variables that are related to any variable in V_s either by appearing in the integral limits of the same indivisible sub-function (Line 15-20) or by appearing in the same constraint (Line 21-26). Meanwhile, the sub-function and the constraint that contain variables in V_s are added to the sub-problem (Line 18 and Line 24). A sub-problem is found when no new variables can be added to V_s (Line 29). It is a class consisting of a set of constraints, and a set of indivisible sub-functions whose multiplication forms the objective function of the constrained optimization sub-problem. The algorithm removes the variables that belong to found sub-problems, and thus

the separation process ends when no variable is left for separation (Line 34).

Algorithm 1 Optimization problem separation algorithm.

Input: the optimization problem P.
Output: a set S of independent sub-problems.

```
1: S :=new Set();
2: V_p := the set of all unprimed variables in P;
3: F_p := the set of all indivisible sub-functions in P;
4: C_p := the set of all constraints in P;
5: repeat
6:     V_s := new Set(); //create a set of unprimed variables for a sub-problem
7:     F_s := new Set(); //create a set of indivisible sub-functions for a sub-problem
8:     C_s := new Set(); //create a set of constraints for a sub-problem
9:     pick a variable t in V_p and remove t from V_p;
10:    add t to V_s;
11:    repeat
12:        V'_s := V_s;
13:        for each variable u in V_p do
14:            for each variable v in V_s do
15:                for each indivisible sub-function f in F_p do
16:                    if u and v are in the integral limits of f then
17:                        add u to V_s and remove u from V_p;
18:                        add f to F_s;
19:                    end if
20:                end for
21:                for each constraint c in C_p do
22:                    if u and v are in c then
23:                        add u to V_s and remove u from V_p;
24:                        add c to C_s;
25:                    end if
26:                end for
27:            end for
28:        end for
29:    until V'_s == V_s;
30:    p_s := new Problem(); //Problem is a class
31:    p_s.f :=F_s;
32:    p_s.c :=C_s;
33:    add p_s to S;
34: until V_p == ∅;
35: return S
```

Indicator function elimination. Heretofore, the objective function of each optimization sub-problem has been provided with indicator functions,

which will increase computational cost when directly handled in solving the optimization problem. To save the cost, we propose to remove the indicator function and transform the objective function to an equivalent form for easier computation. As mentioned earlier, the indicator function $\mathbb{1}_C(\mathbf{v}')$ ($\mathbf{v}' = (v'_1, v'_2, \ldots, v'_n)$) returns 1 when $C(\mathbf{v}')$ is satisfied and 0 otherwise, which means that to remove the indicator function we just need to find the value range of each $v'_i (1 \leq i \leq n)$ that satisfies $C(\mathbf{v}')$. We propose to obtain the ranges by using the interval constraint solver Realpaver [41], which can compute value ranges of variables that satisfy a constraint efficiently. Then we make the integral upper and lower limits fall in the value ranges so that the constraints are always true and therefore the indicator function can be removed. More specifically, suppose the integral range of the integral variable v'_i appeared in the constraints are $[v_i + a, v_i + b]$; and value ranges of v'_i satisfying the constraints are acquired. The integral upper and lower limits that satisfy the constraints can be obtained by computing all the potential overlapping ranges of $[v_i + a, v_i + b]$ and the acquired value ranges of v'_i, which is a well-defined simple linear programming problem. The result is a piecewise function without the indicator function. Let us illustrate this process by an example function $\int_{v-5}^{v+5} \mathbb{1}_C(v') \cdot p(v')\mathrm{d}v'$, where $C(v')$ is ${v'}^2 - 16v' \geq 0$. For this constraint, we can have two satisfying value ranges $(-\infty, 0]$ and $[16, +\infty)$. There are two overlapping ranges between $[16, +\infty)$ and $[v - 5, v + 5]$. When $v \in [21, +\infty)$, the entire range of $[v-5, v+5]$ is within $[16, +\infty)$, so the function can be simplified as shown in Figure 4, condition (1). When $v \in [11, 21)$, the two ranges are only partially overlapped and the overlapped range is $[16, v+5]$, so we substitute the integral limits, as shown by condition (2). For the other satisfying range $(-\infty, 0]$, there are also two overlapping ranges that can be computed in a similar manner, and the complete result is shown in Figure 4.

Problem solving with genetic algorithm. The constrained optimization problem can be nontrivial when the objective function or constraints are complex. Considering that the objective function usually involves complex nonlinear expressions, even the state-of-the-art solvers cannot analytically solve the problem effectively. As an outstanding instance of intelligent algorithms, ge-

$$\int_{v-5}^{v+5} \mathbb{1}_C(v') \cdot p(v')\mathrm{d}v' = \begin{cases} \int_{v-5}^{v+5} p(v')\mathrm{d}v', & v \in [21, +\infty) \quad (1) \\ \int_{16}^{v+5} p(v')\mathrm{d}v', & v \in [11, 21) \quad (2) \\ 0, & v \in (5, 11) \quad (3) \\ \int_{v-5}^{0} p(v')\mathrm{d}v', & v \in (-5, 5] \quad (4) \\ \int_{v-5}^{v+5} p(v')\mathrm{d}v', & v \in (-\infty, -5] \quad (5) \end{cases}$$

Figure 4: Indicator function elimination.

netic algorithm has been successfully applied in solving complex optimization problems. A genetic algorithm is a metaheuristic random search technique that simulates the process of natural selection. Since genetic algorithm does not require a derivative, it is very suitable for solving relatively complicated constrained optimization problems [54]. Thus, we leverage the genetic algorithm to solve our optimization problem. Before solving, we first try to simplify the objective function by transforming the variable limit integral functions to an expression free of integrals, since generally the computation cost of the expression without integrals is lower than the variable limit integral function. The transformation is achieved by finding the antiderivatives of the integrands in the integral functions. For example, suppose that we have an objective function $\int_{v-5}^{v+5} f(v')\mathrm{d}v'$, which is a variable limit integral function. Then if we can find a function $F(v')$ such that $F'(v') = f(v')$, then $F(v')$ is the antiderivative of $f(v')$ and the objective function can be transformed to $F(v+5) - F(v-5)$. The search of an antiderivative of an integrand can be computed by MATLAB [1]. This process is optional, and for any variable limit integral function that cannot be computed by symbolic integral, we can still obtain the function value by numerical integral. For our problem, the concerned variable limit integrals are mostly probability density functions of various distributions, e.g., Gaussian distribution and uniform distribution, etc., whose antiderivatives can be efficiently computed by MATLAB.

We perform the optimization problem solving with genetic algorithm in MATLAB, and the process is shown as Algorithm 2. The genetic algorithm we use has an additional elitist selection scheme compared to the standard one,

similar to Charbonneau's genetic algorithm PIKAIA [18]. The input consists of the optimization (sub-)problem P to be solved, and an integer n and a small real value ϵ serving as the termination condition of the main iteration in the algorithm. Since P is a constrained optimization problem, the objective function f and the set of constraints C of P are first extracted for subsequent fitness evaluation (Line 1-2). The algorithm then generates an initial population of a fixed size. Each element in the population is called an individual (candidate solution). In the initial population, it contains a randomly picked value for each input variable in the objective function f from the variable's value range and is encoded to a binary form. The core part of the algorithm is the iteration process (Line 6-15), in which metaheuristic uses the evaluation values to make the population evolve and fitter individuals (i.e., better solutions) to be chosen. The evaluation values specify the fitness degrees of the individuals, which are basically generated by a fitness function, together with some other functions if the problem to be solved is complex. Since P is a constrained optimization problem, the goal of the search is to find a solution that optimizes the objective function f, while satisfying the set of constraints C. Therefore, we directly use f as the fitness function. Besides, to handle the constrained optimization problem, the penalty function method is applied to cope with constraints. If the constraints are satisfied, the penalty value is 0; otherwise, the penalty function adds a negative number to the fitness value depending on the violation degrees of the constraints, by using the method in [27]. The sum of the values of the fitness function and the penalty functions serves as the evaluation value. The searching terminates when it reaches the iteration times n or the population's best evaluation value no longer changes significantly (i.e., the difference between best evaluation values of two consecutive iterations is smaller than ϵ). Then the most fitted individual is decoded to obtain the input variables' values and its corresponding objective function value. These variable values and the computed objective function value comprise the returned optimization solution for the problem. As is known that the genetic algorithm may not always return the optimal solution, however, with a small ϵ the solution can be approximately

optimal.

Algorithm 2 Genetic algorithm based optimization problem solving.

Input: the optimization (sub)-problem P, the iteration times n, and the difference ϵ between two consecutive evaluation values.

Output: the optimization solution s.

```
1: f := the objective function of P;
2: C := the set of constraints of P;
3: pop := generateInitialPopulation(); //pop is a vector
4: evalValue := 0;
5: k := 0;
6: repeat
7:     evalValue' := evalValue;
8:     fitnessValue := evalFitness(f,pop); //fitnessValue is a vector
9:     penaltyValue := evalPenalty(C,pop); //penaltyValue is a vector
10:    evalValue := max(fitnessValue+penaltyValue);
11:    pop := select(pop,fitnessValue+penaltyValue);
12:    pop := crossOver(pop);
13:    pop := mutate(pop);
14:    k++;
15: until k < n or evalValue − evalValue' < ε
16: individual := selectMostFitted(pop);
17: s.variableValues := decode(individual);
18: s.optimizationValue := f(s.variableValues);
19: return s;
```

During the iteration process, there are several key operations: evaluation(Line 8-9), selection (Line 11), crossover (Line 12) and mutation (Line 13). We introduce them briefly in the following:

- Evaluation. It evaluates the fitness degrees using the sum of the values of the fitness function and the penalty function for each individual in the population.
- Selection. It chooses parent individuals for the next children population based on the scaled values from the evaluation. We adopt a standard selection method, i.e., roulette wheel selection, where the probability of selecting individual i is $eval_i / \sum_{j=1}^{m} eval_j$, in which $eval_i$ is the evaluation value of individual i and m is the total number of individuals. We additionally use the elitist strategy to guarantee that a number of elites in

the population can survive to the next generation by specifying the elite count e. The selected individuals are in the selection pool.

- Crossover. It crosses two parent individuals to form two new children individuals for the next population. To determine the parents for crossover, given a crossover probability p_c, we randomly pick $p_c * m/2$ parents from the selection pool. In a specific crossover operation, single point strategy is applied to cross the parents. The parents are then replaced by their children in the selection pool.
- Mutation. It makes small random changes to the individuals of the population, which provides genetic diversity and enables the genetic algorithm to search a broader space. We choose the individual in the selection pool one by one, and randomly change its char (e.g., changing a binary form individual's char from 0 to 1) at a mutation probability p_m.

The setting of genetic algorithm's parameters (e.g., crossover probability p_c and mutation probability p_m) is problem-dependent, however, there are useful guidelines. The initial population size m is set to 50 when there are five or fewer variables in the fitness function, or 200 or more depending on the number of variables. It is not advised to set m with a small value, or the improvement per iteration in the fitness function will be low. The elite count e in the elite strategy should be set according to the population size, which is $0.05 \times m$. For the mutation and crossover probabilities, generally speaking, a low mutation probability and a high crossover probability can lead the algorithm to search in a relatively concentrated area over the solution space and converge to one of several local optima. However, a high mutation probability and a low crossover probability can cause the good candidate solutions perturbed and pushed into worse solutions, making the algorithm convergent slowly. To balance accuracy and efficiency, existing literature has summarized many valuable experiences for our reference [54]. As recommended, it is safe to set the crossover probability from $[0.4, 0.9]$ and mutation probability from $[0.0001, 0.1]$. Moreover, in our problem, we suggest to choose a high mutation probability (e.g. 0.1) and a low

crossover probability (e.g. 0.4) from the safe value ranges. This may cost more computation time, but produce a solution closer to the global optimum. Since a small increase in the probability can reduce many tries in field validation, much validation time can be saved, which makes the extra computation time worthwhile. Similarly, we advise that the real value ϵ which serves as one of the termination conditions of the main iteration to be a fairly small value from $[0.00001, 0.0001]$, so as to trade computation time for a more accurate solution. The iteration times n works as an auxiliary termination condition when the difference of the fitness function values between two consecutive iterations is not smaller than ϵ. It should be set with a large value if there is an enough time budget. For example, in our robot-car application the iteration times n is set to 1000, which makes the maximum computation time to be about 10 minutes.

The solution includes the values of the unprimed variables which represent the actual environment, and the objective function's value which indicates the approximately optimal probability. The values of the primed variables are not necessary for validation, because the construction of the environment settings only requires the values reflecting the actual environment, which are already stored in the unprimed variables. Moreover, if the values of the primed variables are needed for some reason, e.g. to form a complete path-equivalent counterexample, one can simply use the condition checked for the original counterexample and substitute the unprimed variables with obtained concrete values, and solve the condition to get the values of the primed variables with a SMT solver.

4.6. Discussions

The validity of our work is based on the fact that the SMT solvers used in the verification process cannot guarantee to return a high-probability counterexample. When exploiting SMT solvers such as Z3 to solve a constraint, just one specific solution will be returned by most solvers if the constraint can be satisfied. This is reasonable for most cases because one solution is enough to demonstrate the satisfiability of a constraint. However, for our problem it is not sufficient since users expect counterexamples with higher probabilities for

easier validation. One way to get around the problem is to obtain a new solution by augmenting the original constraint set with the negation of already obtained but rejected solutions due to their low probabilities. However, there is no guarantee that a new solution would be better. Another more ambitious idea is to directly formulate such probability requirements into the constraint set before the verification process. There is indeed some work that shares the same idea to find optimal solutions for SMT problems. For example, in [22, 57] each clause of an SMT formula is associated with some weight or cost. The task is to find a feasible assignment such that the total weight of satisfied clauses are maximized, or a given cost function is minimized. Our problem is different since the probability of a solution (a.k.a. counterexample) is dependent on the probability of each clause's satisfaction, which is unknown before the solution. Thus, we postpone our optimization until a counterexample has been acquired.

In our approach, the problem of obtaining a path-equivalent counterexample with a higher probability is transformed into a constrained optimization problem. The objective function of the optimization problem relates to the modeled uncertainty, so our approach's effectiveness might be affected by uncertainty models. Although the information about uncertainty may not be stringently precise, it serves already well for validation under real deployment. With higher probabilities, counterexamples require less times of validation to witness their occurrences. Therefore, on the one hand, counterexamples can be validated (confirmed) more efficiently provided that the uncertainty modeling is precise enough. On the other hand, if the uncertainty modeling is imprecise, the inconsistency between the calculated and experimental probabilities can be discovered earlier, resulting in that imprecise uncertainty modeling are exposed faster. Considering the goal of this work is to efficiently validate verification results, the quick discovery of imprecise modeling which is the key source of false counterexamples is also one of the contributions made in this work. This is supported by our evaluation in the next section.

Our approach depends on the power of the interval constraint solver and optimization toolbox we used. Given a limited amount of time, it is possible

the solution of the constrained optimization problem is not globally optimal but approximately optimal. One option is to use this approximate solution, since it guarantees to generate a path-equivalent counterexample with higher probability, which is effective in speeding up the validation process. The other option is to allow more time for problem solving, since our work can greatly reduce the number of tries in validating the verification results in real environment. Each try in executing a self-adaptive application can be costly since it requires a heavy setup for the environment and time-consuming waiting for the completion of the application's execution. Thus, the cost spent on solving optimization problems makes a good return since it saves a lot of cost that could have been spent on real environment validation.

It is necessary to mention that our work deals with the uncertainty that arises from the self-adaptive applications' environmental interactions (e.g., unreliable sensing). This type of uncertainty is aleatory and modeled as distributions in our work. Nevertheless, there are other epistemic uncertainties (e.g., user's uncertainty in specifying requirements) that cannot be modeled as distributions. They are not considered in this work. However, those uncertainties also deserve research efforts to be investigated in verifying and validating self-adaptive applications, and we plan to study more types of uncertainty in future.

The uncertainty in our approach has a continuous distribution. We observed that most environmental interactions suffering uncertainty from unreliable sensing could be modeled by continuous variables. We have surveyed existing literatures [39, 56, 58, 52, 42, 8] and learned that almost all sensors, including location sensors, radios sensors, photogrammetry sensors, ultrasonic sensors, temperature sensors, sonar and GPS, have continuous uncertainty. Nevertheless, our approach is not restricted to continuous distributed uncertainty. For the few sensing uncertainty that is discretely distributed, a few adjustments can make our approach support discrete uncertainty. Actually, uncertainty being discrete could make the modeling and analysis even easier. To model discrete uncertainty, its error range would be a set of values enumerating all possible uncertainty values in the range, and each value is assigned with a probability.

Since there are only a finite number of uncertainty values in the range, for the probability function introduced in Equation 1, the number of values for vector $\mathbf{X}$ is also finite. Therefore, one can enumerate all values of $\mathbf{X}$, and for each value of $\mathbf{X}$ calculate the value of $\mathbb{1}_{PC} \cdot p(\mathbf{X})$. Then the result of the probability function would be the sum of all the calculated values. Besides, the process of maximizing the probability function would be simpler. Clearly, the technique of indicator function elimination is no longer necessary, as it is designed for continuous uncertainty. The techniques of optimization problem separation can still be applied, and the problem solving can use genetic algorithms as well.

5. Experimental Evaluation

We implemented our approach as a prototype, and our evaluation addresses the following research questions:

- **RQ1:** *Can our approach effectively improve the efficiency of validating counterexamples for self-adaptive applications?*
- **RQ2:** *How does the precision of uncertainty model affect the effectiveness of our approach?*
- **RQ3:** *How does our genetic algorithm perform in solving the optimization problem compared with other algorithms?*

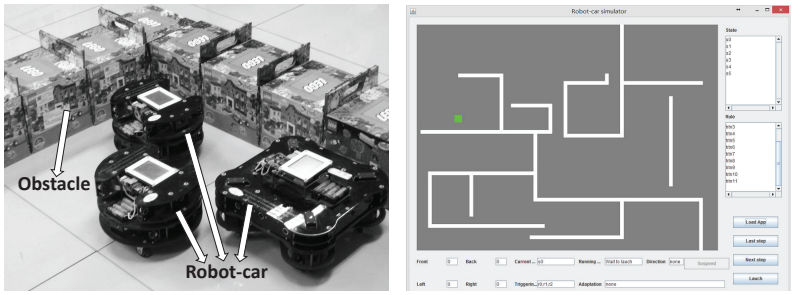
5.1. Experimental Setup

We select self-adaptive applications running in manageable environment for controlled experimentation purposes. There can be many self-adaptive applications that interact with physical environments and adapt their behavior based on sensed environmental changes, such as autopilot cars, drones and robotic arms. However, they are not all suitable for real-world experiments. To conduct our evaluation, we require both the selected applications and their running environments to be manageable. Specifically, an application's adaptation strategies should be open to us, so that its verification can be performed to obtain

corresponding counterexamples. Besides, we prefer an application's running environment to be manageable in a laboratory setting. This is the reason why we chose a group of robot-car applications as our experimental subjects, since their adaptation strategies are open and it is easy for us to place obstacles for these cars in our lab environment. To alleviate the limitation of our application selection, we asked different researchers and students in our university to independently develop various robot-car applications implementing different adaptation strategies during the past four years. We finally selected 20 different applications (with diverse adaptation strategies) as our experimental subjects [76]. Our approach requires counterexamples for these applications as its inputs. So we first applied the state-of-the-art verification approach [76] to derive a set of counterexamples for each of these applications. Each counterexample is accompanied with an initial probability. These counterexamples were used to find the high-probability path-equivalent counterexamples with our approach. For the part of genetic algorithm search, the initial population, crossover probability, mutation probability, ϵ value and iteration times are set to 200, 0.4, 0.1, 0.00001 and 1000, respectively. The optimization ran on a desktop PC with an Intel Core i7 CPU @3.4GHz and 8GB RAM, and the version of MATLAB in use was R2014a. Then the counterexamples with and without our approach were compared for validation efficiency. We set up a real environment in the lab (Figure 5(a)) and developed a simulator (Figure 5(b)) for validating robot-car applications. For field tests, we ran robot-car applications and monitored their behavior affected by environment-interacting uncertainty. For simulation, we simulated noisy environments by their error ranges and distributions.

5.2. RQ1: Overall Effectiveness

To answer RQ1, we first randomly selected 10 counterexamples from each subject application for validation, which do not include all the counterexamples. In our experiments, it takes quite some time to set up the environment for each counterexample. For practical consideration, we can only sample a subset of counterexamples. Besides, we believe that the validation results of the randomly



(a) (b)

Figure 5: Real environment (a) and simulator (b).

selected counterexamples can well illustrate the effectiveness of our approach. Then, our approach was applied to each selected counterexample to find a path-equivalent counterexample with a higher probability. Compared to the original counterexamples, these new counterexamples are supposed to be validated more efficiently in the evaluation. Here, efficiency is measured by the times of executing an application before triggering the concerned counterexample, since all path-equivalent counterexamples followed the same path in execution and thus take almost the same time. Lastly, we evaluated efficiency improvement by comparing execution times with and without our approach. Therefore, we validated both the original counterexamples and their path-equivalent counterexamples in real environment. Specifically, for each counterexample we ran its application under the corresponding environment setting for a fixed times to observe whether the application would fail following the path specified in the counterexample. Meanwhile, we recorded the first time and the total times that each counterexample was triggered. The former information is used to measure validation efficiency, and the latter information serves as a ground truth to assess the accuracy of our calculated probability for each counterexample. The times of running the application for each counterexample was set to 100 since we found that the lowest probability of the counterexamples after optimization is about 0.01. Conducting repeated experiments is a good practice to eliminate the bias of evaluation, and thus we also did the above experiments in the simulation as

a complement to refine our experimental results by providing more sampling data. In the simulation, the times of running for each counterexample is set to 1,000.

Results. For all the selected counterexamples, our approach has successfully found their path-equivalent counterexamples. As expected, those path-equivalent counterexamples' probabilities are all higher than their original ones: those having more than 10 times of increase take up 80% of the total counterexamples, and those having more than 100 times of increase take up 28.5%. On the average, the optimized counterexamples' probabilities are 126 times higher than the ones of the original counterexamples. These optimized counterexamples are supposed to be validated more efficiently, and the experimental results confirm it. Figure 6 shows all the experimental data, and Table 1 lists the results in the real environment experiments of Application 16 which is randomly chosen. In Table 1, we compare three types of data between the original and the optimized counterexamples: the calculated probability (Column 2 and 3), the experimental probability (Column 4 and 5) and the time cost in validation (Column 6 and 7). For each counterexample, the calculated probability has increased by 4 (Counterexample #5) to 82 times (Counterexample #8). This increase of probabilities has also appeared in the real environment experiments. For the original counterexamples, we observed very few occurrences in field experiments so the probability is extremely low as shown in Column 4. But for the optimized counterexamples, the times of occurrence can go from 2 up to 33 out of 100 times of executions, which results in considerably higher probabilities as shown in Column 5 suggesting that our approach has a higher chance to validate real counterexamples. Column 6 and Column 7 list the first time of occurrence and time cost for the original and optimized counterexample. Clearly, the first time of counterexample occurrence after the optimization is much smaller. This results in the time cost being significantly reduced accordingly (from 30 up to 119 minutes, averagely 77 minutes). This gain is acquired at a very low cost, as shown in Column 8. The extra time to compute the higher-probability counterexamples is just about 3 to 9 minutes (averagely 5 minutes). Therefore, our

Table 1: Validation results of original and optimized counterexamples for one application

Counter-example	Orig. cal. P	Opt. cal. P	Orig. exp. P	Opt. exp. P	Orig. first occur.	Opt. first occur.	Comp. time
# 1	0.00442	0.223	0.02	0.22	50 (92min)	3 (6min)	321s
# 2	0.00898	0.107	0.01	0.10	73 (84min)	4 (5min)	214s
# 3	0.000998	0.038	0	0.04	N (114min)	17 (19min)	358s
# 4	0.000443	0.091	0	0.07	N (82min)	65 (40min)	207s
# 5	0.00833	0.035	0.01	0.04	85 (104min)	24 (26min)	472s
# 6	0.00112	0.015	0	0.02	N (130min)	21 (25min)	182s
# 7	0.0767	0.124	0.01	0.12	80 (73min)	4 (3min)	219s
# 8	0.00126	0.178	0	0.17	N (129min)	7 (10min)	544s
# 9	0.00113	0.063	0	0.06	N (78min)	16 (12min)	331s
# 10	0.0193	0.324	0.03	0.33	23 (38min)	6 (8min)	263s

* Symbol "N" stands for no counterexample occurrence in the validation.

approach can indeed reduce the validation time.

The above conclusion is consistent with the complete experimental data set, which is shown in Figure 6. In this figure, (a) and (b) are the results of the real environment experiments, while (c) and (d) are the results from simulation. Figure 6 (b) and (d) shows the first occurrence times of the original and optimized counterexamples. If there is no witness of occurrence, we set the first time to the maximum time conducted in the experiments (there are 106 and 82 such points in (b) and (d), respectively). We can see from the two figures that the first times of counterexample occurrence for the original ones (blue points) have been significantly reduced after the optimization (green points). The other two figures, (a) and (c), compare the calculated probabilities with the experimental ones. They are shown in the logarithmic coordinates. The results indicate that the calculated probability is close to the counterexample's probability of occurrence in real cases. This means that our approach not only improves the validation efficiency, but can also provide information about the extent of improvement.

5.3. RQ2: Impact of Uncertainty Model

Error range and distribution are commonly used in physics to specify uncertainty. Our work introduces this practice to model environment-interacting

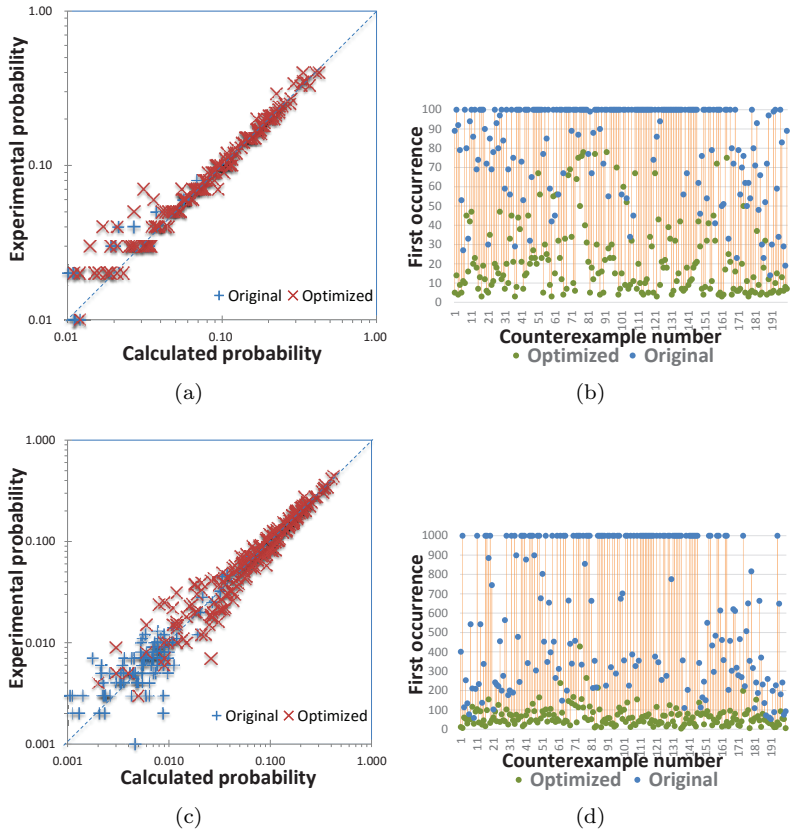


Figure 6: Complete data set of probability and the first time of counterexample occurrence. Subfigure (a) and (c) are comparisons of calculated and experimental probability in real experiments and simulation, respectively. Subfigure (b) and (d) are comparisons of first occurrence of the original and optimized counterexamples in real experiments and simulation, respectively.

uncertainty and can support different distributions. The uncertainty model is obtained from field studies and experiments with statistical analysis, however, it is not guaranteed that the uncertainty model will exactly coincide with the real cases. RQ2 evaluates how the precision of uncertainty model affects the effectiveness of our validation approach. The precision of uncertainty model is determined by the precision of error ranges and distributions' relevant parameters (e.g., the mean and variance in our experiment). In the experiments for RQ1, we found that the probabilities of counterexample occurrences in real environment and simulation were close to the calculated ones, which indicates our

uncertainty model is considerably precise. We used this model as a baseline, and applied various changes to the model to observe impacts. Specifically, we identified three controlled variables: the error range, the mean and the variance of the distribution. The change of these variables was made in a controlled way that each time we changed one variable and see whether it impacts the validation. To evaluate the degree of the impact, we tried both increasing and decreasing the variables by $\pm 20\%$ and $\pm 40\%$. We randomly chose Application 1, and conducted experiments on the 10 counterexamples used in RQ1. The mean or the variance of the distribution was not used in the verification and thus would not invalidate existing counterexamples. Thus, we can repeat our optimization approach to get the path-equivalent counterexamples with the changed uncertainty model, and then validate the results in both real environment and simulation. Yet when the error range is changed, we need to first repeat the verification process on the counterexample's path since the changed error range will affect verification results. Then we can conduct the experiments to check the impact of the change.

Results. When each variable of the uncertainty model was altered, we conducted validation experiments to observe the impact on the results. Each counterexample has been validated 200 times in real environment experiment, and 1,000 times in simulation. All data demonstrate the same impact consistently, so we just use the real environment experiment results of Application 1 for explanation, which are shown in Table 2. As we can see, the change of any uncertainty model parameter would have an impact on both the calculated and experimental probabilities, and most changes demonstrate patterns. The increase of the error range would result in the increase of the calculated probability, which makes sense since error ranges can cause unexpected behavior that might be application failures. However, the experimental probability goes in the opposite direction. The more the error range deviates from the real uncertainty model, the more likely a false counterexample is found which would not occur in reality. Therefore, we can see a striking contrast between calculated and experimental probabilities when altering error ranges. The parameter "mean"

also causes a clear difference on the value of the calculated and experimental probabilities. The parameter "variance" is more intriguing. In our experiment, we found that the values of variables in optimized counterexamples almost stay the same when changing the variance. The reason could be that the change of variance does not affect a variable's value that maximizes the counterexample's probability, while our approach always seeks for the maximum probability. However, with an imprecise variance, the calculated probability is not accurate, so again we see a difference between the calculated and experimental probabilities.

The reason that the experimental probability does not confirm the calculated one is that the calculation is based on imprecise uncertainty model. However, such inaccurate calculated probabilities of counterexamples are still important. As we know, the calculated and experimental probabilities should be close when the uncertainty model is precise. Then by observing the consistency between the two probabilities, our approach can effectively conclude that the uncertainty model is precise or not. Note that without our approach, one can still notice the imprecision by comparing the calculated and experimental probabilities. However, this requires a huge number of validations, since the probability without maximization is often very small. Our approach, on the contrary, provides a higher probability and can thus expose imprecise uncertainty model much faster. This makes the discovery of false counterexamples sooner, since imprecise uncertainty models would lead to false counterexamples.

5.4. RQ3: Algorithm Comparisons

Multiple techniques are proposed by researchers to tackle the challenge of solving constrained optimization problems, including deterministic analytic methods (e.g., gradient-based algorithms) and heuristic searching methods (e.g., genetic algorithms). Meanwhile, there are variants of genetic algorithms with different selection, crossover or mutation strategies and other evolutionary algorithms. It is of valuable reference to explore how other algorithms perform compared with our approach in solving the optimization problem. Therefore, in order to answer RQ3, we compared our algorithm with five other algorithms:

Table 2: Impact of uncertainty model precision on validation effectiveness

Counter-example	**Probability**	**Optimized**	**Altered error range**				**Altered mean**				**Altered variance**			
			-40%	-20%	+20%	+40%	-40%	-20%	+20%	+40%	-40%	-20%	+20%	+40%
# 1	Calculated	0.225	0.037	0.089	0.284	0.308	0.279	0.388	0.375	0.256	0.074	0.098	0.279	0.301
	Experimental	0.200	0.055	0.075	0.145	0	0.015	0.080	0.030	0	0.150	0.130	0.170	0.230
# 2	Calculated	0.094	0.027	0.053	0.173	0.212	0.281	0.132	0.051	0.066	0.037	0.071	0.136	0.244
	Experimental	0.090	0.040	0.065	0.015	0	0	0.085	0.100	0	0.065	0.105	0.080	0.085
# 3	Calculated	0.144	0.032	0.059	0.081	0.365	0.062	0.088	0.096	0.167	0.043	0.066	0.287	0.353
	Experimental	0.100	0.045	0.050	0.105	0	0	0.040	0.070	0	0.125	0.095	0.120	0.105
# 4	Calculated	0.336	0.029	0.202	0.259	0.382	0.228	0.107	0.326	0.074	0.033	0.052	0.406	0.486
	Experimental	0.410	0.075	0.180	0.150	0	0	0	0.008	0.015	0.290	0.305	0.295	0.320
# 5	Calculated	0.369	0.035	0.292	0.228	0.316	0.109	0.156	0.185	0.089	0.093	0.115	0.502	0.570
	Experimental	0.330	0.060	0.250	0	0	0.060	0.030	0.005	0	0.350	0.315	0.295	0.390
# 6	Calculated	0.143	0.028	0.078	0.167	0.218	0.201	0.118	0.095	0.104	0.062	0.094	0.196	0.224
	Experimental	0.150	0.075	0.085	0.015	0.005	0	0.050	0.065	0.030	0.125	0.115	0.125	0.140
# 7	Calculated	0.065	0.019	0.047	0.069	0.106	0.126	0.053	0.117	0.074	0.015	0.039	0.113	0.142
	Experimental	0.060	0.025	0.055	0.010	0.005	0.005	0.085	0	0	0.070	0.055	0.075	0.055
# 8	Calculated	0.088	0.039	0.054	0.114	0.158	0.043	0.155	0.078	0.083	0.023	0.046	0.126	0.139
	Experimental	0.100	0.065	0.060	0.085	0.005	0.010	0.095	0	0	0.070	0.065	0.075	0.050
# 9	Calculated	0.234	0.039	0.214	0.276	0.302	0.203	0.178	0.198	0.155	0.085	0.113	0.297	0.306
	Experimental	0.240	0.055	0.195	0.100	0	0.030	0.050	0.005	0	0.185	0.240	0.210	0.260
# 10	Calculated	0.053	0.013	0.035	0.074	0.085	0.106	0.065	0.087	0.101	0.019	0.026	0.096	0.132
	Experimental	0.050	0.020	0.025	0.020	0.005	0	0	0.065	0.030	0.040	0.045	0.065	0.040

a gradient-based algorithm (GBA) [13], an augmented Lagrangian genetic algorithm (ALGA) [23] and three evolutionary algorithm including (1+1) [30], Hill Climbing [64] and Alternating Variable Method (AVM) [47]. They are representatives of analytic algorithms, variants of genetic algorithms and evolutionary algorithms, respectively.

GBA is a widely used analytic algorithm that leverages the gradient of the objective function and nonlinear constraints to solve the constrained optimization problems, so the objective function and constraints need to be continuous and their first derivatives need to be continuous, too. ALGA, as a variant of genetic algorithm, can also solve a nonlinear optimization problem with nonlinear constraints, linear constraints and bounds. The objective function and nonlinear constraints are combined using the Lagrangian and the penalty parameters and approximately minimized using the genetic algorithm such that the linear constraints and bounds are satisfied. Algorithm (1+1) is an evolutionary algorithm, and the size of the population is just one individual represented as a bit string. It uses a bitwise mutation operator that flips each bit independently of the others with a probability p_m which depends on the length n of the bit string ($p_m = 1/n$). It replaces the current bit string with the new one if the fitness of the current bit string is not superior to the fitness of the new string. Hill Climbing is an iterative algorithm that starts with an arbitrary solution to a problem, then attempts to find a better solution by incrementally changing a single element of the solution. It is good for finding a local optimum, but not necessarily guaranteed to find the global optimum. AVM is also a local search algorithm that was originally applied to automatic test input generation problems by Korel [47]. The algorithm starts on a random search point, and then it considers modifications of the input variables, one at a time. It employs a pattern search that consists of applying increasingly larger changes to the chosen variable as long as a better solution is found.

The GBA and ALGA are implemented in MATLAB's optimization toolbox and we implemented the algorithm (1+1), the standard Hill Climbing and AVM algorithm introduced in [44]. We randomly selected 50 counterexamples

Table 3: Number of generated sub-problems after using each simplification technique

Technique	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**	**#9**	**#10**
Separation	6	7	4	6	8	5	5	4	8	4
Elimination	15	23	15	17	26	18	21	17	27	11

as subjects from the experiment addressing RQ1 to find their path-equivalent counterexamples with higher probabilities, and formulated the constrained optimization problems for these counterexamples. Before applying the five algorithms and our algorithm to get the results, we need first simplify the problems with the simplification techniques. Otherwise, the problems would have multiple integrals and indicator functions, and thus be too complex for any algorithm to solve. To show the impact of the simplification techniques on the problems, we randomly selected 10 counterexamples from the 50 ones, and in Table 3 listed the number of sub-problems generated by the techniques. The first technique of optimization problem separation divides a constrained optimization problem to several smaller sub-problems. The small sub-problems contain less constraints and variables, and simpler integral functions. As we can see from Table 3, our technique can successfully separate the big problem to 4-8 sub-problems in general. The second technique of indicator function elimination can transform the objective function to piecewise functions. The range of integration of each transformed function is guaranteed to satisfy the indicator function, which results in that the indicator function can be eliminated. The third row of Table 3 shows the number of sub-problems after this process. The third technique transforms the variable limit integral functions to expressions free of integrals. It does not change the number of sub-problems, but can significantly reduce the computation cost. For the 50 counterexamples, we simplified their corresponding constrained optimization problems, and recorded the optimized solutions and time costs for comparison.

For a more objective comparison of different algorithms' obtained results, we also conducted statistical tests. We randomly selected 10 counterexamples from the above 50 counterexamples. Every algorithm was run for 100 times

for each selected counterexample to account for random variations inherited in search algorithms. As suggested in the guidelines of using statistical tests to assess randomized algorithms [3], the Vargha and Delaney statistics and Mann-Whitney U test were adopted. The Vargha and Delaney statistics is used to calculate $\widehat{A}_{12}$, which is a non-parametric effect size measure [3]. In our context, given the calculated probability *cal*, $\widehat{A}_{12}$ is used to compare the the probability of yielding a higher value *cal* for two algorithms A and B. If $\widehat{A}_{12}$ is equal to 0.5, the two algorithms are equivalent. If $\widehat{A}_{12}$ is greater than 0.5, it indicates that the first algorithm A has higher chances of obtaining a higher *cal* value than B.

The non-parametric U-test (The Mann-Whitney U test) is used to calculate the p-value for deciding whether there is a significant difference between two algorithms. We chose the significance level of 0.05, which means that there is a significant difference if p-value is less than 0.05. When comparing the calculated probabilities of the selected algorithms, the Mann-Whitney U test is further performed together with Vargha and Delaney statistics for pair-wise comparisons between the algorithms. Based on the above description, we define that algorithm A has better performance than algorithm B, if the $\widehat{A}_{12}$ value is greater than 0.5. Moreover, algorithm A has significantly better performance than algorithm B, if the $\widehat{A}_{12}$ value is greater than 0.5 and the p-value is less than 0.05.

Results. Figure 7 shows the calculated probabilities by six algorithms, and Figure 8 shows their time costs. Since the purpose of the optimization is to find path-equivalent counterexamples with higher probabilities, the algorithm that can produce higher probabilities is considered to be more effective. If similar probabilities are produced, then the algorithm with less computation time is considered superior. As we can see from Figure 7, all algorithms can obtain higher probabilities for the counterexamples except for GBA. We explicitly show the results of GBA by purple triangles, since GBA could only give results for seven counterexamples out of the total 50 ones. Moreover, even for the seven counterexamples for which it could compute the results, GBA spent the most time on computation compared with the other five algorithms, as shown in

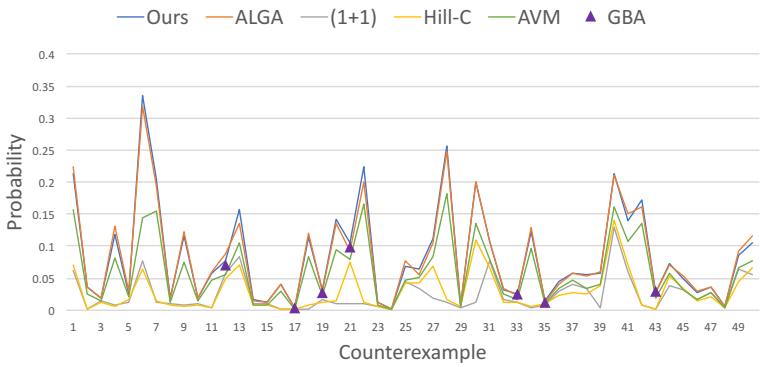


Figure 7: Optimized probabilities of 50 counterexamples by six different algorithms.

Figure 8. The results show that as an analytic algorithm, GBA cannot handle the degree of complexity of our optimization problems.

Among the five heuristic algorithms including ours, algorithm (1+1) and Hill Climbing demonstrated similar performance. Their time costs for computation are close and less than the other algorithms. However, their solutions are significantly inferior than the other four algorithms. AVM spent a bit more time to get slightly better solutions compared with algorithm (1+1) and Hill Climbing. However, its solutions are not fully optimized compared with our algorithm and ALGA. The top two algorithms in terms of producing higher probabilities are ours and ALGA, while in most cases our algorithm performed sightly better than ALGA. Besides, when considering the computation time, our algorithm outperformed ALGA for spending less time on all the counterexamples.

The evaluated results of statistical tests show a clearer comparison between our approach and other algorithms. As discussed above, since GBA could only give results for a few counterexamples, it was excluded from the pair-wise comparisons. Table 4 presents the evaluated results for each pair of the algorithms. When comparing our algorithm with (1+1), Hill Climbing and AVM, for all ten counterexamples, the $\widehat{A}_{12}$ values are far greater than 0.5 and the p-values are far less than 0.05. Thus, we can conclude that our algorithm has significantly better performance than algorithm (1+1), Hill Climbing and AVM. In compar-

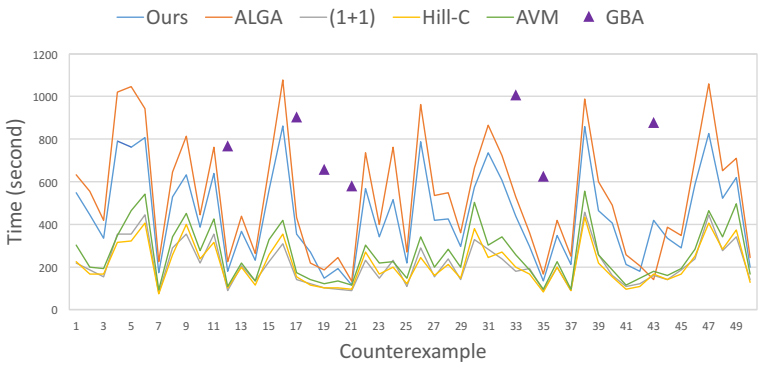


Figure 8: Time cost of optimizing 50 counterexamples by six different algorithms.

Table 4: Statistics of comparing different algorithms

Counter-example	Ours vs. ALGA $\widehat{A}_{12}$/p	Ours vs. (1+1) $\widehat{A}_{12}$/p	Ours vs. Hill-C $\widehat{A}_{12}$/p	Ours vs. AVM $\widehat{A}_{12}$/p
# 8	0.5896/6.32E-5	1/2.2e-16	1/2.2e-16	0.9978/1.1e-15
# 10	0.4327/6.48E-1	1/2.2e-16	1/2.2e-16	0.8992/7.6e-14
# 14	0.6013/4.45E-3	1/2.2e-16	1/2.2e-16	1/2.2e-16
# 19	0.5732/2.67E-3	1/2.2e-16	1/2.2e-16	1/2.2e-16
# 26	0.5007/6.18E-2	1/2.2e-16	1/2.2e-16	0.9554/4.1e-14
# 27	0.4964/7.32E-2	1/2.2e-16	1/2.2e-16	1/2.2e-16
# 30	0.4321/3.32E-1	1/2.2e-16	1/2.2e-16	1/2.2e-16
# 36	0.5101/4.08E-2	1/2.2e-16	1/2.2e-16	1/2.2e-16
# 41	0.7133/5.67E-6	1/2.2e-16	1/2.2e-16	1/2.2e-16
# 43	0.5449/5.89e-2	1/2.2e-16	1/2.2e-16	1/2.2e-16

ison of our algorithm with ALGA, for seven out of ten counterexamples, the $\widehat{A}_{12}$ values are greater 0.5 and among them 5 p-values are less than 0.05. For the rest three counterexamples of which the $\widehat{A}_{12}$ values are less than 0.5, all the p-values are greater than 0.05. The results show that for half of the counterexamples, our algorithm is significantly better than ALGA. For the other half five counterexamples, our algorithm has better performance in two cases and ALGA has better performance in three cases. However, none of the better-performance cases is significant. Therefore, our algorithm is a better choice to obtain higher probabilities compared with ALGA.

6. Related Work

In this section, we discuss some closely related work concerning quality assurance for self-adaptive applications, uncertainty handling in self-adaptive applications and probabilistic analysis.

Quality assurance. Developing high-quality self-adaptive applications is confronted with stiff challenges [75, 19], and many research efforts are made to assure their quality. Some studies address the problem of testing self-adaptive applications [72, 70, 16, 59, 69]. However, the testing approach is generally infeasible to predict and enumerate all possible environmental conditions that an application can encounter at runtime [53, 61]. Lots of efforts are then spent on developing certifiable verification methods [71, 79]. There are lots of studies about verification of self-adaptive applications against properties including safety [29], liveness and reachability [50, 66], reliability[15, 35], and stability [66, 9]. Verification techniques are also used to devise advanced adaptation strategies for self-adaptive applications to achieve quality requirements [55, 36, 14]. In those pieces of work, probabilistic models of self-adaptive applications are exploited to determine a strategy that drives the application to satisfy the non-functional properties specified by probabilities.

As we know, interaction uncertainty affects the precision of the verification results and is difficult to precisely specify, so the validation of verification results is necessary. Despite its importance, related work of self-adaptive application validation is very limited. Therefore, we discuss some related validation techniques in general applications. Simulation is one of the validation approach which models the application's execution environment [5, 4]. However, it can be hard for simulation to consider not only the environment and hardware, but also the uncertainty, which often lacks a precise specification. Another kind of work of validation focuses on studying abstractions of system behavior [24, 17]. Different from these studies, our approach focuses on increasing the efficiency in validating counterexamples of self-adaptive applications in real environment by finding counterexamples of higher probabilities with the optimization the-

ory. From this point of view, there is a series of studies [63, 74] that focus on rare events simulation and optimization from system engineering, sharing a similar purpose with our work. Simulation optimization is proposed to solve optimization problems of complex real systems that usually cannot be modeled by clear functional representations, typically via Monte Carlo simulation methods [68]. However, it usually takes long time to simulate rare events using traditional Monte Carlo methods because of their low probabilities. To address this challenge, importance sampling techniques are proposed to find a different distribution than the original distribution of interest, so as to improve simulation efficiency. The most important step in the importance sampling method is to find the optimal importance sampling distribution function. For this problem, in work [63, 74], researchers proposed two different approaches by minimizing the variance of importance sampling estimator and minimizing cross entropy, respectively. Rare events simulation and optimization can also be applied to our problem, as the occurrence of a counterexample with a low probability can be considered as a rare event. This method does not require an expression modeling the system structure, but requires the construction of a simulated system and its corresponding simulation process. Compared with this method, our approach first explicitly derives a probability function. Then, an optimal solution that maximizes the counterexample's probability is calculated with such a probability function, and the simulation process is no longer needed.

Uncertainty handling. Uncertainty has always been a bone of contestation in self-adaptive applications [60, 33] and even the whole software engineering community [31, 40]. Ramirez et al. [60] 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. Cheng et al. [21] presented a requirement language RELAX to address uncertainty explicitly in application requirements. Esfahani et al. [32] proposed an approach to handling uncertainty by assessing both positive and negative consequences of uncertainty. Garlan et al. [37] proposed to mitigate uncertainty in Rainbow framework by comparing running average in monitoring

with architectural descriptions that are augmented with probabilistic information to detect trend of behavior. Once the problem is detected, a strategy is selected to resolve the problem. A recent article by Moreno et al. [55] presented an approach for latency aware self-adaptive applications under uncertainty that uses probabilistic model checking for adaptation decision. Our previous work [76] focused on a different kind of uncertainty that arises from the self-adaptive applications' environmental interaction, and handled its effects during the verification process. However, the possible false counterexamples obtained from the verification because of the imprecise specification of uncertainty were not dealt with in that work. Therefore, we studied the problem of validating verification results and focused on improving the validation efficiency.

Probabilistic analysis. Our work is related to a collection of work about probabilistic software analysis [38, 34, 10, 67] and probabilistic model checking [43, 49, 45]. Probabilistic software analysis aims at quantifying the probability of a target event to occur during a program execution[38, 51, 11], or computing interval bounds on the probability from an adequate set of paths [67, 2]. Our work differs from it mainly in two ways. First, probabilistic software analysis does not directly apply to self-adaptive applications, as sensing and adaptation operations typically require invoking library calls from device manufacturers, while our work considers modeling constraints for the effects of such library calls. Second, our work aims to obtain one path-equivalent counterexample with a higher probability than a given one by exploiting optimization theory, so it finds a specific input solution; but probabilistic software analysis would find an input space that triggers all path-equivalent counterexamples and give an overall probability. Probabilistic model checking takes on different goals with our work, whereas it generally considers counterexample generation for probabilistic formulas and its input is a system with the probabilities for each transition already provided.

7. Conclusion

In this paper, we introduce a novel approach to improving the efficiency of validating counterexamples for self-adaptive applications by finding path-equivalent counterexamples of higher probabilities with respect to original ones. To achieve so, a counterexample's probability is formulated into a probability function, and acts as the objective function to be maximized in a constrained optimization problem. With the proposed multiple optimization techniques, the optimization problem can be efficiently solved. In an evaluation on real-world applications, the path-equivalent counterexamples acquired by our approach have their probabilities significantly increased by 126 times averagely. These counterexamples were validated in real environment and simulation, and the results consistently confirmed the large improvement of validation efficiency.

8. Acknowledgments

This work was supported in part by National Key R&D Program (Grant No. 2017YFB1001801), National Natural Science Foundation (Grant Nos. 61472174, 61690204), and Jiangsu Natural Science Foundation (Grant No. BK20150589) of China. The authors thank the anonymous reviewers for valuable comments and suggestions for improving this article. The authors would also like to thank the support from the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China.

References

[1] MATLAB. http://www.mathworks.com/.

[2] A. Adje, O. Bouissou, J. Goubault-Larrecq, E. Goubault, and S. Putot. Static analysis of programs with imprecise probabilistic inputs. In *Verified Software: Theories, Tools, Experiments*, volume 8164 of *LNCS*, pages 22–47. Springer Berlin Heidelberg, 2014.

[3] A. Arcuri and L. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1–10, New York, NY, USA, 2011. ACM.

[4] A. Arrieta, G. Sagardui, and L. Etxeberria. A configurable test architecture for the automatic validation of variability-intensive cyber-physical systems. In *Proc. VALID '14*, pages 79–83, Nice, France, 2014.

[5] A. Arrieta, G. Sagardui, and L. Etxeberria. Test control algorithms for the validation of cyber-physical systems product lines. In *Proc. SPLC '15*, pages 273–282, Nashville, Tennessee, 2015.

[6] J. M. Aughenbaugh. *Managing uncertainty in engineering design using imprecise probabilities and principles of information economics*. PhD thesis, Georgia Institute of Technology, 2006.

[7] C. Baier, J.-P. Katoen, et al. *Principles of model checking*, volume 26202649. MIT press Cambridge, 2008.

[8] B. Barshan and R. Kuc. Active sonar for obstacle localization using envelope shape information. In *[Proceedings] ICASSP 91: 1991 International Conference on Acoustics, Speech, and Signal Processing*, pages 1273–1276 vol.2, Apr 1991.

[9] 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, 2011.

[10] M. Borges, A. Filieri, M. d'Amorim, and et al. Compositional solution space quantification for probabilistic software analysis. In *Proc. PLDI '14*, pages 123–132, Edinburgh, United Kingdom, 2014.

[11] M. Borges, A. Filieri, M. d'Amorim, and C. S. Păsăreanu. Iterative distribution-aware sampling for probabilistic symbolic execution. In *Proc. Joint ESEC/FSE '15*, pages 866–877, Bergamo, Italy, 2015.

[12] Y. Brun, G. Di Marzo Serugendo, C. Gacek, and et al. Engineering self-adaptive systems through feedback loops. In *Software Engineering for Self-Adaptive Systems*, volume 5525 of *LNCS*, pages 48–70. Springer Berlin Heidelberg, 2009.

[13] H. R. Byrd, C. J. Gilbert, and J. Nocedal. A trust region method based on interior point techniques for nonlinear programming. *Mathematical Programming*, 89(1):149–185, 2000.

[14] R. Calinescu, C. Ghezzi, M. Kwiatkowska, and R. Mirandola. Self-adaptive software needs quantitative verification at runtime. *Commun. ACM*, 55(9):69–77, Sept. 2012.

[15] 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, 2012.

[16] 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.

[17] G. D. Caso, V. Braberman, D. Garbervetsky, and S. Uchitel. Enabledness-based program abstractions for behavior validation. *ACM Trans. Softw. Eng. Methodol.*, 22(3):25:1–25:46, July 2013.

[18] P. Charbonneau. Genetic algorithms in astronomy and astrophysics. *The Astrophysical Journal Supplement Series*, 101:309, 1995.

[19] B. Cheng, K. Eder, M. Gogolla, and et al. Using models at runtime to address assurance for self-adaptive systems. In *Models@run.time*, volume 8378 of *LNCS*, pages 101–136. Springer International Publishing, 2014.

[20] B. H. C. Cheng, R. de Lemos, H. Giese, and et al. Software engineering for self-adaptive systems: A research roadmap. In *Software Engineering for Self-Adaptive Systems*, volume 5525 of *LNCS*, pages 1–26. Springer Berlin Heidelberg, 2009.

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

[22] A. Cimatti, A. Franzén, A. Griggio, R. Sebastiani, and C. Stenico. Satisfiability modulo the theory of costs: Foundations and applications. In *Proc. TACAS '10*, pages 99–113. Paphos, Cyprus, 2010.

[23] A. R. Conn, N. Gould, and P. L. Toint. A globally convergent lagrangian barrier algorithm for optimization with general inequality constraints and simple bounds. *Math. Comput.*, 66(217):261–288, Jan. 1997.

[24] G. de Caso, V. Braberman, D. Garbervetsky, and S. Uchitel. Automated abstractions for contract validation. *IEEE Trans. Softw. Eng.*, 38(1):141–162, 2012.

[25] R. de Lemos, H. Giese, H. Müller, 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.

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

[27] K. Deb. An efficient constraint handling method for genetic algorithms. *Computer Methods in Applied Mechanics and Engineering*, 186(24):311 – 338, 2000.

[28] N. D'Ippolito, V. Braberman, J. Kramer, J. Magee, D. Sykes, and S. Uchitel. Hope for the best, prepare for the worst: Multi-tier control for adaptive systems. In *Proc. ICSE '14*, pages 688–699, Hyderabad, India, 2014.

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

[30] S. Droste, T. Jansen, and I. Wegener. On the analysis of the (1+1) evolutionary algorithm. *Theoretical Computer Science*, 276(1):51 – 81, 2002.

[31] S. Elbaum and D. S. Rosenblum. Known unknowns: Testing in the presence of uncertainty. In *Proc. FSE '14*, pages 833–836, Hong Kong, China, 2014.

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

[33] N. Esfahani and S. Malek. Uncertainty in self-adaptive software systems. In *Software Engineering for Self-Adaptive Systems II*, volume 7475 of *LNCS*, pages 214–238. Springer Berlin Heidelberg, 2013.

[34] A. Filieri, C. S. Păsăreanu, and W. Visser. Reliability analysis in symbolic pathfinder. In *Proc. ICSE '13*, pages 622–631, San Francisco, CA, USA, 2013.

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

[36] J. a. M. Franco, F. Correia, R. Barbosa, M. Zenha-Rela, B. Schmerl, and D. Garlan. Improving self-adaptation planning through software architecture-based stochastic modeling. *J. Syst. Softw.*, 115(C):42–60, May 2016.

[37] D. Garlan, S. Cheng, A. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer*, 37(10):46–54, Oct. 2004.

[38] J. Geldenhuys, M. B. Dwyer, and W. Visser. Probabilistic symbolic execution. In *Proc. ISSTA '12*, pages 166–176, Minneapolis, MN, USA, 2012.

[39] S. Gezici, Z. Tian, G. B. Giannakis, H. Kobayashi, A. F. Molisch, H. V. Poor, and Z. Sahinoglu. Localization via ultra-wideband radios: a look

at positioning aspects for future sensor networks. *IEEE Signal Processing Magazine*, 22(4):70–84, July 2005.

[40] H. Giese, N. Bencomo, L. Pasquale, and et al. Living with uncertainty in the age of runtime models. In *Models@run.time*, volume 8378 of *LNCS*, pages 47–100. Springer International Publishing, 2014.

[41] L. Granvilliers and F. Benhamou. Algorithm 852: Realpaver: An interval solver using constraint satisfaction techniques. *ACM Trans. Math. Softw.*, 32(1):138–156, Mar. 2006.

[42] R. Gutierrez-Osuna, J. A. Janet, and R. C. Luo. Modeling of ultrasonic range sensors for localization of autonomous mobile robots. *IEEE Transactions on Industrial Electronics*, 45(4):654–662, Aug 1998.

[43] T. Han, J.-P. Katoen, and D. Berteun. Counterexample generation in probabilistic model checking. *IEEE Trans. Softw. Eng.*, 35(2):241–257, 2009.

[44] M. Harman and P. McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Transactions on Software Engineering*, 36(2):226–247, March 2010.

[45] H. Hermanns, B. Wachter, and L. Zhang. Probabilistic cegar. In *CAV*, volume 5123 of *LNCS*, pages 162–175. Springer Berlin Heidelberg, 2008.

[46] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[47] B. Korel. Automated software test data generation. *IEEE Trans. Softw. Eng.*, 16(8):870–879, Aug. 1990.

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

[49] M. Kwiatkowska, G. Norman, and D. Parker. Prism 4.0: Verification of probabilistic real-time systems. In *CAV*, volume 6806 of *LNCS*, pages 585–591. Springer Berlin Heidelberg, 2011.

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

[51] K. Luckow, C. S. Păsăreanu, M. B. Dwyer, and et al. Exact and approximate probabilistic symbolic execution for nondeterministic programs. In *Proc. ASE '14*, pages 575–586, Vasteras, Sweden, 2014.

[52] L. Matthies and S. A. Shafer. *Error Modeling in Stereo Navigation*, pages 135–144. Springer New York, New York, NY, 1990.

[53] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. Cheng. Composing adaptive software. *Computer*, 37(7):56–64, 2004.

[54] G. Mitsuo and R. Cheng. *Genetic algorithms and engineering optimization*. John Wiley & Sons, 2000.

[55] G. A. Moreno, J. Camara, D. Garlan, and B. Schmerl. Proactive self-adaptation under uncertainty: a probabilistic model checking approach. In *Proc. Joint ESEC/FSE '15*, pages 1–12, Bergamo, Italy, 2015.

[56] R. Morton and E. Olson. Robust sensor characterization via max-mixture models: Gps sensors. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 528–533, Nov 2013.

[57] R. Nieuwenhuis and A. Oliveras. On sat modulo theories and optimization problems. In *SAT*, volume 4121 of *LNCS*, pages 156–169. Springer Berlin Heidelberg, 2006.

[58] L. Y. Pao and L. Trailovic. The optimal order of processing sensor information in sequential multisensor fusion algorithms. *IEEE Transactions on Automatic Control*, 45(8):1532–1536, Aug 2000.

[59] 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, 2005.

[60] 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, 2012.

[61] 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, Lawrence, Kan, USA, 2011.

[62] S. S. Rao and S. Rao. *Engineering optimization: theory and practice*. John Wiley & Sons, 2009.

[63] R. Y. Rubinstein. Optimization of computer simulation models with rare events. *European Journal of Operational Research*, 99(1):89 – 112, 1997.

[64] S. J. Russell, P. Norvig, J. F. Canny, J. M. Malik, and D. D. Edwards. *Artificial intelligence: a modern approach*, volume 2. Prentice hall Upper Saddle River, 2003.

[65] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):14:1–14:42, 2009.

[66] 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.

[67] S. Sankaranarayanan, A. Chakarov, and S. Gulwani. Static analysis for probabilistic programs: Inferring whole program properties from finitely many paths. In *Proc. PLDI '13*, pages 447–458, Seattle, Washington, USA, 2013.

[68] E. Tekin and I. Sabuncuoglu. Simulation optimization: A comprehensive review on theory and applications. *IIE transactions*, 36(11):1067–1081, 2004.

[69] 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, 2004.

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

[71] 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, 2012.

[72] 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.

[73] 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, 2013.

[74] R. Y. Rubinstein. Combinatorial optimization, cross-entropy, ants and rare events. 01 2001.

[75] W. Yang, Y. Liu, C. Xu, and S. Cheung. A survey on dependability improvement techniques for pervasive computing systems. *Science China Information Sciences*, 58(5):1–14, 2015.

[76] W. Yang, C. Xu, Y. Liu, C. Cao, X. Ma, and J. Lu. Verifying self-adaptive applications suffering uncertainty. In *Proc. ASE '14*, pages 199–210, Vasteras, Sweden, 2014.

[77] 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, 2013.

[78] M. V. Zelkowitz and I. Rus. Understanding iv & v in a safety critical and complex evolutionary environment: The nasa space shuttle program. In *Proc. ICSE '01*, pages 349–357, Toronto, Ontario, Canada, 2001.

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