Environment Rematching:

Toward Dependability Improvement for Self-Adaptive Applications

Chang Xu[1,2], Wenhua Yang[1,2], Xiaoxing Ma[1,2], Chun Cao[1,2], and Jian Lü[1,2]
[1]State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China
[2]Department of Computer Science and Technology, Nanjing University, Nanjing, Jiangsu, China
changxu@nju.edu.cn, ihope1990@126.com, {xxm, caochun, lj}@nju.edu.cn

***Abstract*—Self-adaptive applications can easily contain faults. Existing approaches detect faults, but can still leave some undetected and manifesting into failures at runtime. In this paper, we study the correlation between occurrences of application failure and those of consistency failure. We propose fixing consistency failure to reduce application failure at runtime. We name this environment rematching, which can systematically reconnect a self-adaptive application to its environment in a consistent way. We also propose enforcing atomicity for application semantics during the rematching to avoid its side effect. We evaluated our approach using 12 self-adaptive robot-car applications by both simulated and real experiments. The experimental results confirmed our approach's effectiveness in improving dependability for all applications by 12.5–52.5%.**

***Index Terms*—Consistency failure, environment rematching.**

I. Introduction

Self-adaptive applications adapt their behaviors based on environmental changes. They use *states* to represent their varying understanding to environments, and associate different application logics with these states to fulfill functionalities in different ways. When an application encounters new environmental attribute values, it transits to a corresponding state and switches to new application logic associated with that state. Thus each state implies a range of environmental attribute values that can be handled by its associated application logic. We name this implication *assumption on environment*.

Self-adaptive applications keep collecting environmental attribute values, transiting to corresponding states, and evolving to new application logics. This is known as *adaptation*. The correctness of application logic at one state is often guaranteed by traditional model checking or testing approaches [12][13][16][22]. The correctness of adaptation across states is more difficult to guarantee. Our previous study [22] on real-world self-adaptive applications reported that 58.6–77.5% application failures relate to environment-sensing or self-adaptation code. This indicates extra challenges in developing such applications.

Many self-adaptive applications define adaptation by *adaptation rules*. These rules structurally specify *conditions* that trigger adaptation and *actions* that execute adaptation. We call them *m̲odel-based s̲elf-adaptive a̲pplications* (MSAs). Existing static analysis work [12][16] used model checking to explore an MSA's state space to detect failures. They require an MSA's rule conditions to be specified by propositional logic, with action semantics simplified. Our previous work [22] relaxed these assumptions. It specifies an MSA's rule conditions by first-order logic, and explicitly models action semantics. For this increased expressive power, failures are detected dynamically.

Such detected failures are called *model failures*, as they manifest at a model level (with respect to states and rules). Model failures can grow into observable *application failures* (with certain application semantics). Much existing work detected model failures, including non-determinism failure [4][12][16][22], stability failure [12][16][22], reachability and liveness failure [16][22], and consistency failure [9][11][22]. Our previous study [22] observed that *consistency failure often occurs together with other model failures*. This suggests a possible correlation between occurrences of consistency failure and those of other model failures, as well as a further correlation with those of application failure.

Consistency failure means an application's understanding to its environment (by current state) not matching its actual environment (by collected environmental attribute values). Consistency failure behaves as a state's assumption on environment being violated by environmental attribute values. We conducted a study of 12 self-adaptive robot-car applications to investigate the correlation between occurrences of consistency failure and those of application failure. We observed that 27.7–99.2% consistency failures led to application failures, and 51.0–95.2% application failures were accompanied by consistency failures (details given later). This confirms a strong correlation between occurrences of consistency failure and those of application failure. This also inspires us to fix consistency failure so as to reduce application failure.

In this paper, we aim to recover an MSA from consistency failure. We name this *environment rematching*, i.e., making an application's current state and follow-up adaptation matched again with its actual environment. Developers can hardly consider every exceptional case for a self-adaptive application. Our environment rematching can release developers from considering numerous or even infinite exceptional cases. It allows them to focus on normal functionalities, and exceptional cases are taken care of by our approach once they cause any consistency failure. Thus our approach can practically improve an MSA's dependability. Our later evaluation confirmed 12.5–52.5% dependability improvement for these robot-car applications.

The remainder of this paper is organized as follows. Section II introduces MSA modeling. Section III reports our correlation study on consistency failure and application failure. Section IV

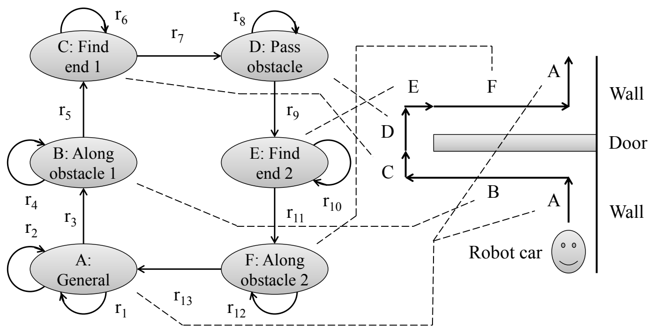

Fig. 1. An example robot-car MSA and its obstacle-avoiding scenario.

presents our environment rematching approach. Section V experimentally evaluates our approach. Section VI discusses related work, and finally Section VII concludes this paper.

II. Preliminaries

We model an MSA as $M := (S, R, s_0)$. S is the set of all states and R is the set of all rules. State $s_0 \in S$ is the initial state, with which the MSA starts. Each state $s \in S$ is associated with a subset of rules $R_s \subseteq R$. When the MSA resides at a state (called *current state*), the state's associated rules are *enabled*, while others are disabled.

When new environmental attribute values are collected, each enabled rule's condition is checked. If satisfied, the rule is *triggered*. If multiple rules are triggered, priority mechanism [16][19][20] is applied for resolution. One rule is finally selected for execution by taking its associated actions. We thus model a rule r as: r := (*condition*, *actions*, *owner*).

Here, *condition* is a logical formula for checking; *actions* is what to take when the rule is executed; *owner* is the rule's associated state. We assume each rule's owner to be unique. Variables in a rule's condition take values from environmental attributes. When new environmental attribute values are collected, a rule's condition's truth value is subject to change. A rule's actions include *cyber actions* (e.g., updating an MSA's current state) and *physical actions* (e.g., controlling an MSA for environmental interactions).

We take a self-adaptive robot-car [25] MSA for example. Fig. 1 (left) illustrates the MSA, which controls a robot car to explore an unknown area. The car should not bump into any obstacle detected by its ultrasonic sensors. Fig. 1 (right) illustrates how the MSA controls the car to avoid an opened door.

The MSA is defined as: $M := (S, R, s_0)$. S := {A, B, C, D, E, F}, $R := \{r_1, r_2, \ldots, r_{13}\}$, and s_0 := A. Initial state A assumes the car walking safely, i.e., in an open area, or along a long obstacle by keeping a constant distance to it. Three rules (r_1, r_2, r_3) are associated with State A:

r_1 := ("$dis_F > 50$ && $dis_B > 50$ && $dis_L > 50$ && $dis_R > 50$", {walk$_F$}, A).
r_2 := ("$dis_F > 50$ && $40 \le dis_R \le 50$", {walk$_A$}, A).
r_3 := ("$dis_F < 40$", {turn$_L$, update(B)}, A).

Four variables (dis_F, dis_B, dis_L, dis_R) represent measured distances between the car and nearby obstacles at four orientations (front, back, left, right), respectively. "> 50" means a *safe* distance larger than 50cm. Action "walk$_F$" means walking forward by a unit of distance. Rule r_1 keeps the car walking forward if no obstacle is detected nearby. If the car finds its distance to a right-hand obstacle in [40, 50], it would walk forward and maintain this distance by slight orientation adjustments. This is what Rules r_2's action "walk$_A$" does. Finally, Rule r_3 turns the car left by 90 degrees if any front obstacle is detected ("$dis_F < 40$"), and transits the MSA to a new state B. We do not elaborate on other states or rules due to space limit. This application serves as a running example in the paper.

III. Consistency Failure

A. Failure Formulation

Consistency failure represents a situation in which an MSA's current state has its assumption on environment violated by collected environmental attribute values. To specify assumptions, we associate each state *s* with an invariant *s.invariant*. If an MSA resides at a state but the state's invariant is violated, a *consistency failure* occurs.

The invariant can be any constraint or condition formulated for this purpose. It can be a context consistency constraint [19] [21] that asserts about environmental attribute values to be conflict-free. It can also be a guard condition [14][22] for a state to be set as current state.

Consider our robot-car MSA. Its state B is associated with two rules (r_4, r_5). Rule r_4 keeps the car walking along an obstacle. Rule r_5 turns the car right once it detects the obstacle's end. Then State B represents a situation in which the car walks along a long obstacle, and wishes to bypass it after some distance. It is thus assumed that the car's front orientation should be open, and there is an obstacle on right hand during this distance. This assumption can be formulated as State B's invariant: "$dis_F > 50$ && $dis_R \le 50$". If it is violated (e.g., $dis_F \le 50$), the car is unlikely still walking along the long obstacle, and a consistency failure occurs.

Consider State C, which is associated with two other rules (r_6, r_7). Rule r_6 keeps the car walking forward and monitors its right-hand space. Rule r_7 confirms an obstacle previously detected on the car's right hand. Then State C represents a situation in which the car tries to locate an obstacle on its right hand. This process should not take too long (≤ 50 cm). Let dis_T measures how far the car has walked since State C. Then this expectation can be formulated as State C's invariant: "$dis_T < 50$". If it is violated, the car is unlikely still avoiding the original obstacle (it may have disappeared or been missed), and a consistency failure occurs.

If the two consistency failures are left unattended, the car may not be able to make correct adaptation in future. At State B, the car may crash into a front obstacle by executing Rule r_4, if it ignores the violation of invariant "$dis_F > 50$ && $dis_R \le 50$". At State C, it may keep locating its right-hand obstacle by repeatedly executing Rule r6 and ignoring the violation of invariant "$dis_T < 50$", until crashing into another front obstacle.

B. Failure Correlation

We conducted a study of 12 self-adaptive robot-car applications. They were independently developed by different research staffs and students in our university during past three years. We

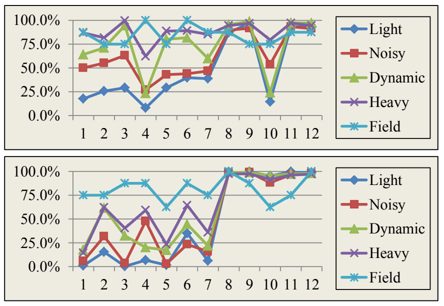

Fig. 2. Application failure rate (top) and consistency failure rate (bottom).

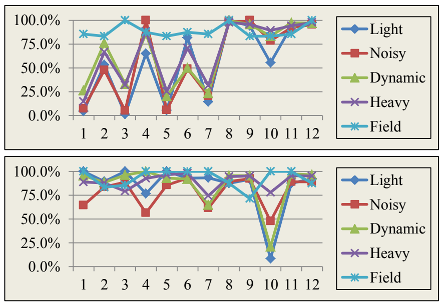

Fig. 3. Both-failure rate (top: against application failure; bottom: against consistency failure).

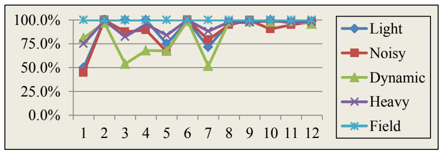

Fig. 4. Close-failure rate (against both-failure).

ran these applications on different hardware platforms (we give data for only one platform, *tricycle car*, due to space limit). We conducted experiments under five configurations. Four of them are simulation and one is field test, namely, *Light*, *Noisy*, *Dynamic*, *Heavy*, and *Field*, respectively. The first four means "ideal scenario without noise or dynamics", "scenario with sensing noise", "scenario with environmental dynamics", and "scenario with both noise and dynamics", respectively. Environmental dynamics behaves as unpredicted object movement an application may not anticipate. *Field* means "field test with real hardware and scenario", with natural noise and dynamics. We used state invariants provided by developers.

We ran each application 250 times for each simulation configuration and 8 times for field test. The total number (12,096 runs) is adequate for alleviating random error. Fig. 2 (top) compares application failure rate for 12 applications, which is defined as the ratio of failed runs against all runs. "Failed" means the car violating safety requirements (e.g., bumping into obstacle). 12 applications have greatly varying application failure rates. This reflects their different quality levels. Fig. 2 (bottom) compares consistency failure rate, which is defined as the ratio of runs in which consistency failure occurred against all runs. For some applications (1, 3, 5, 7), their consistency failure rates are lower, as compared to relative extents of their application failure rates. This suggests that their state invariants are weaker (looser conditions). For some applications (2, 4, 6, 10), their state invariants are stronger (stricter conditions). Other applications (8, 9, 11, 12) have their application failure rates and consistency failure rates both very high.

Fig. 3 (top) compares the ratio of runs with both failures against runs with application failure. Fig. 3 (bottom) compares the ratio of runs with both failures against runs with consistency failure. The averaged ratio (over five configurations) is at least 78.0% in at least one figure for each application. This suggests a correlation between occurrences of application failure and those of consistency failure. When a run had consistency failure, it also had application failure in 27.7–99.2% (mean is 67.1%) cases. When a run had application failure, it also had consistency failure in 51.0–95.2% cases (mean is 86.5%).

We investigate when a run had both failures occurred, whether they occurred temporally closely. "Closely" is defined as at most one triggering of adaptation rule. Fig. 4 compares the ratio of runs with both failures occurred closely against runs with both failures. The averaged ratio (over five configurations) falls in the range of 70.1–99.6% (mean is 90.9%), which is significant. This confirms that application failure and consistency failure did occur in a correlated way.

IV. Environment Rematching Methodology

A. Environment Rematching

When a consistency failure occurs, the concerned MSA's current state is inappropriate as its associated invariant is violated. To fix it, the MSA can transit to another state whose associated invariant is satisfied. However, arbitrary change of current state can bring unexpected side effect. We aim to keep an MSA's application semantics *atomic* in environment rematching, i.e., all cyber and physical actions taken in adaptation are eventually completed or totally cancelled (known as "all-or-nothing" semantics in transactional systems [17][24]).

Consider an MSA's execution trace: $s_0\ r_1\ s_1\ \ldots\ r_n\ s_n$, where s_0, s_1, …, s_n are states and r_1, …, r_n are rules. At state s_n, the MSA has its state invariant violated. Now the MSA can:

Backward rematching. Transit back to its earlier state $s_{n\text{-}1}$, provided that all actions taken in executing rule r_n can be totally cancelled, and then restart executing rule r_n.

Forward rematching. Transit forward to a future state s_{n+1}, provided that all actions taken in executing rule r_{n+1} (connecting state s_n to s_{n+1}) can guarantee to complete.

If state $s_{n\text{-}1}$'s or s_{n+1}'s invariant is also violated, the MSA can transit back to an earlier state, say $s_{n\text{-}2}$, $s_{n\text{-}3}$, …, until s_0, or transit forward to a more future state, say s_{n+2}, s_{n+3}, …, until that state's associated invariant is satisfied.

We associate each action with two attributes, *compensable* and *retriable*. The former specifies whether the action can be compensated without side effect (i.e., totally cancelled). The latter specifies whether the action can be retried via different ways until its completion (i.e., guarantee to complete). The two attributes were borrowed from transactional systems [17][24]. We use them to analyze whether a rule can be totally cancelled or guarantee to complete. Our key insights are as follows:

Backward rematching. Backward rematching aims to undo and then redo the adaptation from state $s_{n\text{-}1}$ to s_n. It handles consistency failure caused by random environmental noise. For example, our robot car's right-hand door may be missed in its walking due to sensing noise, but this may not happen next time. The consistency failure caused by missed sensing can be fixed by backward rematching.

Forward rematching. Forward rematching aims to skip current situation that is not supported by the MSA, and transits

```
Algorithm: Rematching ability analysis algorithm
Input: $M$ (MSA)
Output: $M$ (MSA, annotated with rematching ability values)
 1: for each state $s$ in $M.S$ do  // Analyze 1-step rematching ability
 2:   let $R'$ be the set of rules connecting to state $s$;
 3:   if $\forall r\in R'$ ($r.compensable$ == true) then $s.br := 1$ else $s.br := 0$ end if
 4:   let $R'$ be the set of rules connected from state $s$ but not to $s$;
 5:   if $\exists r\in R'$ ($r.retriable$ == true) then $s.fr := 1$ else $s.fr := 0$ end if
 6: end for
 7: // Analyze $n$-step backward rematching ability
 8: while any change to any $s.br$ do
 9:   for each state $s$ in $M.S$ do
10:     if $s.br > 0$ then
11:       let $S'$ be the set of other states connecting to state $s$;
12:       $s.br := \min\{\min\{s_i.br \mid s_i\in S'\} + 1, M.S.size\}$
13:     end if
14:   end for
15: end while
16: // Analyze $n$-step forward rematching ability
17: while any change to any $s.fr$ do
18:   for each state $s$ in $M.S$ do
19:     if $s.fr > 0$ then
20:       let $S'$ be the set of other states connected from state $s$;
21:       $s.fr := \min\{\max\{s_i.fr \mid s_i\in S'\} + 1, M.S.size\}$
22:     end if
23:   end for
24: end while
25: return $M$
```

Fig. 5. Rematching ability analysis algorithm.

it to another state s_{n+1}, where it can handle other adaptations. For example, our robot car's right-hand door may be closed unexpectedly, and this would cause a consistency failure at State C. Handling this case is not supported by the MSA, but it can be skipped as if the car already bypassed the door.

B. Multi-step Rematching

If an MSA can perform backward or forward rematching for multiple steps, it has a better chance in finding a state consistent with environment. We name this *multi-step rematching*.

We present an algorithm to analyze rematching ability for each state in an MSA (Fig. 5). It first analyzes 1-step rematching ability (Lines 1–6). It then analyzes n-step rematching ability by propagation (Lines 7–24). Analysis results are restricted by the number of total states in the MSA. When forward rematching is performed, its future states are selected from those that carry the largest forward rematching ability values.

Consider two hardware platforms for our robot-car MSA: *track car* and *tricycle car*. All cyber actions are both compensable and retriable for both platforms. Physical actions are supported differently and thus have different compensability and retriability values. Normal "walk forward" action (walk$_F$) is both compensable and retriable for both platforms. Special "walk forward" action with automated orientation adjustments (walk$_A$) is not compensable but retriable for both platforms. Turning-orientation actions (turn$_L$ and turn$_R$) are both compensable and retriable for the track car, but neither compensable nor retriable for the tricycle car. Based on this information, one can analyze compensability and retriability for the MSA's rules on the two platforms. He can further analyze backward and forward rematching abilities for each state in the MSA. We omit analysis results due to space limit.

Generally in an MSA of n states, if a state is n-step backward/forward rematchable, the MSA can transit from this state to any state by backward/forward rematching, and keep its application semantics atomic at the same time.

C. Discussion

Our environment rematching prefers backward rematching to forward rematching. This is because the former is conservative (repeating the last adaptation), while the latter is aggressive (skipping the current situation). Therefore, our environment rematching would try backward rematching first if possible.

Cancelling or completing an action needs underlying support from hardware platforms. For example, our track car uses backward/turning-right/turning-left movement to compensate forward/turning-left/turning-right movement (for compensability). It uses a speed sensor and digital compass to guarantee movement actions to complete successfully (for retriability).

V. EVALUATION

We experimentally evaluate our environment rematching approach. We used the aforementioned 12 self-adaptive robot-car applications as our subjects. They were deployed on two hardware platforms: *track car* and *tricycle car*. We conducted experiments under the aforementioned five configurations: *Light*, *Noisy*, *Dynamic*, *Heavy*, and *Field*. They correspond to a total of 24,192 application runs, and each run took two minutes. We compared four approaches: *Original* (no rematching), *Rematched* (with environment rematching), *Jumping* (directly jumping to a state whose invariant holds), and *Skipped* (with environment rematching but ignoring its rematching ability checking). The last two are for justifying why we have to guarantee application semantics atomic in environment rematching. For comparison, we conducted 96,768 application runs totally.

A. Dependability Improvement

Table I lists how 12 applications' dependability is changed after applying *Rematched* (only track car data are given due to space limit). "Dependability" is defined as the ratio of runs in which no application failure occurred against all runs. Each datum takes the form of "X (Y)". X is original dependability and Y is its value change after applying *Rematched*. We observe that all value changes are positive, and each application's dependability is improved by 12.5–52.5% on average. This shows that our environment rematching is generally helpful for improving MSA dependability.

The averaged dependability improvement (over 12 applications) is 29.0%, 18.1%, 35.9%, 17.7%, and 51.0% for *Light*, *Noisy*, *Dynamic*, *Heavy*, and *Field*, respectively. These values are absolute percentages. If calculated relatively (against original percentage), the improvement is up to 46.2%, 42.5%, 96.7%, 119.3%, and 188.5%. From *Light*, *Noisy*, *Dynamic*, to *Heavy*, scenarios became increasingly tougher, but *Rematched* became more effective. When deployed in *Field*, *Rematched* even achieved over 180% relative dependability improvement. These values confirm the effectiveness of our environment rematching in improving MSA dependability, especially for tough environments.

TABLE I. APPLICATION DEPENDABILITY IMPROVEMENT FOR 12 SELF-ADAPTIVE ROBOT-CAR MSAS.

Configuration	App 1	App 2	App 3	App 4	App 5	App 6	App 7	App 8	App 9	App 10	App 11	App 12	Average	Relative
Light	89.6% (+2.0%)	87.6% (+4.0%)	90.0% (+5.2%)	96.8% (+1.2%)	77.6% (+12.8%)	83.6% (+13.6%)	92.0% (+3.2%)	16.4% (+78.0%)	7.2% (+68.8%)	96.8% (+3.2%)	7.2% (+88.0%)	8.0% (+68.0%)	62.7% **(+29.0%)**	**+46.2%**
Noisy	37.6% (+16.8%)	41.2% (+24.8%)	50.8% (+5.2%)	77.2% (+11.6%)	50.8% (+12.4%)	70.4% (+14.8%)	88.0% (+4.8%)	12.8% (+23.6%)	6.4% (+21.6%)	58.4% (+24.4%)	7.2% (+30.4%)	9.6% (+26.4%)	42.5% **(+18.1%)**	**+42.5%**
Dynamic	39.2% (+18.4%)	26.0% (+57.2%)	83.2% (+10.8%)	66.4% (+24.8%)	22.0% (+20.4%)	42.4% (+21.2%)	74.0% (+10.8%)	4.8% (+81.2%)	2.8% (+32.8%)	78.8% (+21.2%)	4.0% (+84.0%)	2.0% (+48.0%)	37.1% **(+35.9%)**	**+96.7%**
Heavy	4.0% (+20.8%)	12.0% (+18.4%)	28.0% (+16.4%)	18.4% (+30.4%)	7.2% (+14.0%)	23.6% (+10.0%)	44.8% (+14.8%)	4.4% (+5.6%)	2.0% (+10.0%)	26.0% (+48.4%)	4.0% (+10.0%)	3.6% (+13.6%)	14.8% **(+17.7%)**	**+119.3%**
Field	25.0% (+37.5%)	37.5% (+37.5%)	37.5% (+25.0%)	12.5% (+62.5%)	37.5% (+50.0%)	12.5% (+62.5%)	25.0% (+62.5%)	25.0% (+62.5%)	25.0% (+50.0%)	37.5% (+50.0%)	25.0% (+50.0%)	25.0% (+62.5%)	27.1% **(+51.0%)**	**+188.5%**
Average	39.1% **(+19.1%)**	40.9% **(+28.4%)**	57.9% **(+12.5%)**	54.3% **(+26.1%)**	39.0% **(+21.9%)**	46.5% **(+24.4%)**	64.8% **(+19.2%)**	12.7% **(+50.2%)**	8.7% **(+36.6%)**	59.5% **(+29.4%)**	9.5% **(+52.5%)**	9.6% **(+43.7%)**		

TABLE II. DEPENDABILITY IMPROVEMENT COMPARISON FOR 4 APPROACHES.

Configuration	*Original*	*Rematched*	*Jumping*	*Skipped*
Light	62.7%	91.7% (+29.0%)	66.5% (+3.8%)	66.5% (+3.8%)
Noisy	42.5%	60.6% (+18.1%)	45.9% (+3.3%)	48.0% (+5.4%)
Dynamic	37.1%	73.0% (+35.9%)	43.5 (+6.3%)	46.8 (+9.6%)
Heavy	14.8%	32.5% (+17.7%)	18.9% (+4.1%)	21.4% (+6.6%)
Field	27.1%	78.1% (+51.0%)	27.1% (+0.0%)	36.5 (+9.4%)
Average	36.9%	67.2% (**+30.3%**)	40.4% (**+3.5%**)	43.8 (**+7.0%**)

As mentioned earlier, the 12 applications' state invariants are specified in a stricter (2, 4, 6, 10) or looser (1, 3, 5, 7) way. Or, their application failure and consistency failure rates are both very high (8, 9, 11, 12). We thus partition the applications into three groups: *Strict*, *Loose*, and *High*. The averaged dependability improvement (over applications in each group) is 27.1%, 18.2%, and 45.8%, respectively. *Strict* applications have higher dependability improvement than *Loose* ones. This is because *Strict* applications had their state invariants violated, causing consistency failures, more easily. This triggered more environment rematching. This is echoed by *High* applications, which have even higher dependability improvement because they triggered even more environment rematching as a result of their high consistency failure rates.

B. Effectiveness Comparison

Table II compares averaged dependability improvement (over 12 applications) among four approaches. The "*Original*" column provides original dependability values. Other data take the form of "*X* (*Y*)". *X* is new dependability value after applying an approach and *Y* is value change. We observe that *Rematched* consistently performs better than *Jumping* and *Skipped* in improving MSA dependability.

Jumping is the idea of directly transiting an application to a state whose invariant holds. It reflects rematching effort on the cyber part (i.e., inside software), but the physical part (e.g., the car's orientation, location, and its distances to nearby obstacles) is ignored. Thus *Jumping* cannot fulfill environment rematching, and its averaged dependability improvement is only 3.5%.

Skipped is the idea of ignoring whether it is safe to start environment rematching on certain states (i.e., ignoring rematching ability analysis results). *Skipped* achieved slightly higher averaged dependability improvement of 7.0%. It is significantly lower than that of *Rematched* (30.3%). A closer study discloses that *Rematched* successfully rematched an application with environment in 98.4% cases, but *Skipped* did it in 79.0%. This explains why *Skipped* performed worse than *Rematched*.

VI. RELATED WORK

We discuss related work in recent years.

Self-adaptive applications often understand environments imperfectly, because sensing technologies are subject to noises. Incomplete, inaccurate, or conflicting sensory data are common for such applications, which are thus uncertain about whether they can fulfill functionalities [7]. For this, Ramirez et al. [15] presented a taxonomy of uncertain factors that can affect the dependability of self-adaptive applications.

Self-adaptive applications need to handle uncertainty. This can be supported by feedback-based control [3], exception-monitored framework [11], or reflective middleware [4]. For such applications, developers need to consider how to adequately and properly model adaptation. Andersson et al. [1] discussed a set of modeling dimensions that should be considered. Cheng et al. [5] presented a research roadmap for related engineering issues. Kramer and Magee [10] discussed architectural challenges and potential solutions.

Self-adaptive applications may fail at runtime. Existing work can be grouped into two categories: *model-based and code-based fault detection*. In the first category, Sama et al. [16] used model checking to search an application's state space to detect faults. Our previous work [22] used error patterns to track and analyze responsible faults. Liu et al. [12] used machine learning to derive deterministic and likely constraints to prune false warnings and prioritize true faults. In the second category, Lu et al. [13] used new coverage criteria to test data flows in self-adaptive applications. Wang et al. [18] strengthened test cases by focusing on context switching points. These pieces of work detect faults before application deployment.

More work relates to handling runtime errors for self-adaptive applications. It can be grouped into four categories: *data-level, application-level, service-level, and environment-level error handling*. The first category fixed data errors, not considering side effect. Khoussainova et al. [9] fixed sensory data errors probabilistically with as many integrity constraints satisfied as possible. Demsky and Rinard [6] fixed data structure errors based on specifications. In the second category, Garlan et al. [8] improved application dependability through architecture-based self-repairing. Our previous work [20] fixed context errors and prevented application semantics from being affected. The third category considers applications composed from service components. Schuldt et al. [17] required atomicity of transactional systems be protected from composition errors.

Ye et al. [24] studied how atomicity of composed services may be violated globally. In this work, we extend atomicity to self-adaptive applications. In the last category, Boos et al. [2] checked assertions for self-adaptive applications. This echoes our idea of using state invariants to decide whether an application is inconsistent with environment. Yang et al. [23] constructed isolated context views for different applications. It isolates an application from environment rather than matching it with environment, while the latter is our focus in this work.

VII. Conclusion

In this paper, we aim to improve dependability for self-adaptive applications. We propose, before application failure manifests, fixing its correlated model failure. This idea is based on our study of 12 self-adaptive robot-car applications about the correlation between application failure and consistency failure. The correlation is leveraged to guide a systematic environment rematching process, which brings an application back to a state consistent with its environment. This has greatly improved dependability for these robot-car applications. We plan to validate this work on more other self-adaptive applications.

Acknowledgment

This research was partially funded by National Basic Research 973 Program (No. 2009CB320702), National High-tech R&D 863 Program (No. 2012AA011205), National Natural Science Foundation (Nos. 61100038, 61021062, 91318301, 61361120097) of China, and Research Grants Council (No. 612210) of Hong Kong. Chang Xu was also partially supported by Program for New Century Excellent Talents in University, China (No. NCET-10-0486).

References

[1] J. Andersson, R. D. Lemos, S. Malek, and D. Weyns, "Modeling dimensions of self-adaptive software systems", *Software Engineering for Self-Adaptive Systems*, LNCS 5525, pp. 27–47, Springer-Verlag Berlin, Heidelberg, 2009.

[2] K. Boos, C. L. Fok, C. Julien, and M. Kim, "BRACE: an assertion framework for debugging cyber-physical systems", In *Proc. the 34th ICSE*, pp. 1341–1344, Zurich, Switzerland, Jun 2012.

[3] Y. Brun, G. D. M. Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, et al, "Engineering self-adaptive systems through feedback loops", *Software Engineering for Self-Adaptive Systems*, LNCS 5525, pp. 48–70, Springer-Verlag Berlin, Heidelberg, 2009.

[4] L. Capra, W. Emmerich, and C. Mascolo, "CARISMA: context-aware reflective middleware system for mobile applications", *IEEE TSE*, 29(10), pp. 929–945, Oct 2003.

[5] B. H. C. Cheng, R. D. Lemos, H. Giese, P. Inverardi, J. Magee, "Software engineering for self-adaptive systems: a research roadmap", *Software Engineering for Self-Adaptive Systems*, LNCS 5525, pp. 1–26, Springer-Verlag Berlin, Heidelberg, 2009.

[6] B. Demsky and M. C. Rinard, "Goal-directed reasoning for specification-based data structure repair", *IEEE TSE*, 32(12), pp. 931–951, Dec 2006.

[7] N. Esfahani, E. Kouroshfar, and S. Malek, "Taming uncertainty in self-adaptive software", In *Proc. the 8th joint meeting of the ESEC/FSE*, pp. 234–244, Szeged, Hungary, Sep 2011.

[8] D. Garlan, S. W. Cheng, and B. Schmerl, "Increasing system dependability through architecture-based self-repair", *Architecting Dependable Systems*, LNCS 2677, pp. 61–89, 2003.

[9] N. Khoussainova, M. Balazinska, and D. Suciu, "Towards correcting input data errors probabilistically using integrity constraints", In *Proc. the 5th ACM Inter. Workshop on Data Engineering for Wireless and Mobile Access*, pp. 43–50, Chicago, Illinois, USA, Jun 2006.

[10] J. Kramer and J. Magee, "Self-managed systems: an architectural challenge", In *Proc. Future of Software Engineering, the 29th ICSE*, pp. 259–268, Minneapolis, MN, USA, May 2007.

[11] D. Kulkarni and A. Tripathi, "A framework for programming robust context-aware applications", *IEEE TSE*, 36(2), pp. 184–197, Mar/Apr 2010.

[12] Y. Liu, C. Xu, and S. C. Cheung, "AFChecker: effective model checking for context-aware adaptive applications", *JSS*, 86(3), pp. 854–867, Mar 2013.

[13] H. Lu, W. K. Chan, and T. H. Tse, "Testing context-aware middleware-centric programs: a data flow approach and an RFID-based experimentation", In *Proc. the 14th FSE*, pp. 242–252, Portland, Oregon, USA, Nov 2006.

[14] A. L. Murphy, G. P. Picco, and G. C. Roman, "LIME: a coordination model and middleware supporting mobility of hosts and agents", *ACM TOSEM*, 15(3), pp. 279–328, Jul 2006.

[15] A. J. Ramirez, A. C. Jensen, and B. H. C. Cheng, "A taxonomy of uncertainty for dynamically adaptive systems", In *Proc. the 7th SEAMS*, pp. 99–108, Zurich, Switzerland, Jun 2012.

[16] M. Sama, S. Elbaum, F. Raimondi, D. S. Rosenblum, and Z. Wang, "Context-aware adaptive applications: fault patterns and their automated identification", *IEEE TSE*, 36(5), pp. 644–661, Sep/Oct 2010.

[17] H. Schuldt, G. Alonso, C. Beeri, and H. J. Schek, "Atomicity and isolation for transactional processes", *ACM TODS*, 27(1), pp. 63–116, Mar 2002.

[18] Z. Wang, S. Elbaum, and D. S. Rosenblum, "Automated generation of context-aware tests", In *Proc. the 29th ICSE*, pp. 406–415, Minneapolis, MN, USA, May 2007.

[19] C. Xu and S. C. Cheung, "Inconsistency detection and resolution for context-aware middleware support", In *Proc. the Joint ESEC/FSE*, pp. 336–345, Lisbon, Portugal, Sep 2005.

[20] C. Xu, S. C. Cheung, W. K. Chan, and C. Ye, "On impact-oriented automatic resolution of pervasive context inconsistency", In *Proc. the 6th Joint Meeting of ESEC/FSE*, pp. 569–572, Dubrovnik, Croatia, Sep 2007.

[21] C. Xu, S.C. Cheung, W.K. Chan, and C. Ye, "Partial constraint checking for context consistency in pervasive computing", *ACM TOSEM*, 19(3), Article 9, pp. 1–61, Jan 2010.

[22] C. Xu, S. C. Cheung, X. Ma X., C. Cao, and J. Lu, "ADAM: identifying defects in context-aware adaptation", *JSS*, 85(12), pp. 2812–2828, Dec 2012.

[23] H. Yang, C. Xu, X. Ma, L. Zhang, C. Cao, and J. Lu, "ConsView: towards application-specific consistent context views", In *Proc. the 36th COMPSAC*, pp. 632–637, Izmir, Turkey, Jul 2012.

[24] C. Ye, S. C. Cheung, W. K. Chan, and C. Xu, "Atomicity analysis of service composition across organizations", *IEEE TSE*, 35(1), pp. 2–28, Jan/Feb 2009.

[25] L. Zhang, C. Xu, X. Ma, T. Gu, X. Hong, C. Cao, and J. Lu, "Resynchronizing model-based self-adaptive systems with environments", In *Proc. the 19th APSEC*, pp. 184–193, Hong Kong, China, Dec 2012.