IDEA: Improving Dependability for Self-Adaptive Applications

Wenhua Yang, Chang Xu[*] , Linghao Zhang
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China
Department of Computer Science and Technology, Nanjing University, Nanjing, Jiangsu, China
ihope1024@gmail.com, changxu@nju.edu.cn, zlh.nju@gmail.com

ABSTRACT

Self-adaptive applications are becoming popular since they are able to adapt their behavior based on changes of environments. However, possible faults in these applications may result in runtime failures, which reduce their dependability. We propose a novel approach to improving the dependability of self-adaptive applications. The approach uses a rematching process to make self-adaptive applications consistent with their environments. In the rematching process, consistency failures are automatically detected and fixed at runtime to reduce application failures. The strategy for fixing consistency failures includes backward rematching and forward rematching. Proper strategies can be selected according to rematching ability analysis results for concerned applications. As a result, applications can thus achieve consistency with their environments and failures can be significantly avoided. We developed a tool named IDEA to support this process and the experimental results confirmed the effectiveness of our IDEA.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Algorithms, Performance, Design, Experimentation

Keywords

Self-adaptive Application, Failure, Rematching

1. INTRODUCTION

Self-adaptive applications modify their own behavior in response to changes in their environments. They use sensors and actuators, which could be imprecise and imperfect, to interact with the complex and dynamic physical world. Meanwhile, faults often exist in applications. As a result, such applications may be subject to runtime failures. In this paper, we focus on model-based self-adaptive applications which are being widely-used.

Model-based self-adaptive applications are typically constructed using states and transition rules. For each state in such an application, it is associated with some transition rules. These transition rules describe the *conditions*, which should be satisfied when the rules are triggered, and *actions* that will thus be executed. Each state specifies its own understanding to environment, which means that each state and its transition rules are designed to deal with different environmental attribute values. The understanding to environment is called *assumption* on environment, which is specified as an invariant. Traditional model checking or testing approaches [13, 14, 17, 20] can be used to guarantee the correctness of transition rules at one state. However, the correctness of transition rules between states is hard to guarantee. Much work pays attention to the detection of errors at model level, which includes non-determinism [13, 24, 20, 17], stability [17, 13, 20], reachability and liveness [17, 20], and consistency errors [12, 10, 20].

The errors at model level may result in observable *application failures* with certain application semantics at runtime. The consequences of application failures are severe, which we should try to avoid. A consistency failure emerges when the *state*'s assumption on environment is violated, which means the *state*'s understanding to environment is inconsistent with the real physical world. Our previous work [24] studied the correlation between consistency failures and application failures, and found that there is a strong correlation between them. Our approach can take certain adaptive actions to account for violations of state invariants.

We developed a tool named IDEA (Improving Dependability for Self-Adaptive Applications) to detect and fix consistency failures at runtime for model-based self-adaptive applications. The input of IDEA is an application model specified in XML files with specific format including states, transition rules and states' invariants. IDEA has an execution engine to support running the applications, and an error checker checking invariants with environmental attribute values. When consistency failures are detected, the rematching module will be triggered to rematch an application with its environment.

The goal of this paper is to introduce our approach to recovering model-based self-adaptive applications from consistency failures, as well as the details of our IDEA implementation. We would also outline follow-up research steps

[*]Corresponding Author.

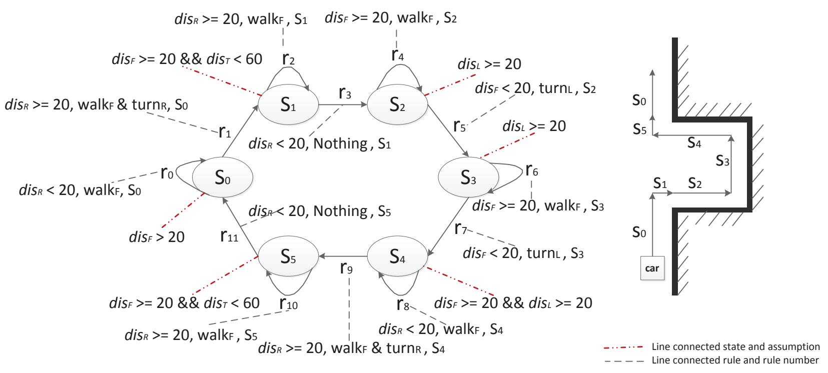


Figure 1: An example robot car application and its scenario.

to take, especially for an extended experimentation.

The remainder of this paper is organized as follows. Section 2 introduces model-based self-adaptive applications, consistency failures and application failures using an illustrating example. Section 3 presents the design and implementation of tool IDEA. Section 4 discusses the experiments and future work. Related work is given in Section 5 and a summary is drawn in the last section.

2. SELF-ADAPTIVE APPLICATIONS

We discuss the model-based self-adaptive applications and the challenges that they confront in the following parts.

2.1 Model-Based Applications

There are plenty of approaches to modeling self-adaptive applications, such as Petri Net, Hybrid Automata, UML, and FSM. Here we follow a popular A-FSM approach [17] proposed in recent years, and extend it to fulfill our requirement.

A model-based self-adaptive application can be modeled as $M := (S, R, s_0)$. S is the set of state, and R is the set of transition rules. A transition rule $r \in R$ is modeled as $r := (condition, actions, state)$, where *condition* is a logical formula that will be checked; *actions* are serially executed (if more than one action are contained) when the transition rule is triggered; and *state* is the owner of the transition rule. State $s_0 \in S$ is the initial state, from which the model-based self-adaptive application starts. For each state $s \in S$, s is associated with a part of rules $R_s \in R$.

An *adaptation* can be described as follows. Firstly, new environmental attribute values are obtained. Secondly, the transition rules associated with the current state are checked for their conditions. If a condition is satisfied, the corresponding transition rule is triggered. Nevertheless, more than one transition rule's condition may be satisfied at one time. For such a case, priority or resolution mechanism [17, 21, 22] can be applied to select one transition rule. Thirdly, the triggered transition rule's actions are taken. Here, the current state may transit to another state in accordance with the actions. Continuous adaptations form a model-based self-adaptive application's execution process.

We take a robot car model-based self-adaptive application as an illustrative example to introduce the way of modeling model-based self-adaptive applications. The application is designed for controlling a robot car to automatically explore unknown area by using ultrasonic sensors without bumping into any obstacles. Figure 1 (left) illustrates a robot car application designed for controlling a robot car. Figure 1 (right) illustrates one scenario in which the application controls the car to explore the area when it walks along a wall.

As discussed before, $M := (S, R, s_0)$, where $S := S_0, \ldots, S_5$ (6 states), $R := r_0, \ldots, r_{11}$ (13 transition rules), and the initial state s_0 is S_0. At the very beginning, the car is in state S_0, which is safe, i.e., no obstacle is nearby or the car walks along an obstacle (like a wall) by keeping a constant distance. We take state S_0 as an example to explain the robot car application, and two transition rules are associated with S_0. They are:

$r_0 := (''disR < 20'', walkF, S_0)$.

$r_1 := (''disR \geq 20'', walkF, turnR, S_0)$.

The variables ($disF$, $disB$, $disL$, $disR$) represent distances sensed by ultrasonic sensors in four directions (front, back, left and right) between the car and its nearby obstacles. 20 is a safe distance, and action walkF means walking forward by a unit of distance. Actions (turnL, turnR, turnB) represent turning the car left, right and back by 90, 90, and 180 degrees, respectively. Transition rule r_0 will keep the car walking along the right obstacle. Transition rule r_1 makes the car walk forward for a unit and then turn right if it finds its distance to its right obstacle bigger than 20, and at the same time, the application transits to a new state S_1 from S_0. Other states and transition rules are elaborated in Figure 1 in which each simplified transition rule (actions of updating states are omitted) is connected to the corresponding rule number. This application serves as a running example for illustrating how consistency failures occur and how they are fixed to rematch the application with the environment in a consistent way.

2.2 Consistency Failures

A consistency failure means that the assumption on the environment of a state, which the application is currently in, is violated. As mentioned before, a state's assumption is specified as an invariant. Given an model-based self-adaptive application $M := (S, R, s_0)$, we associate each state $s \in S$ with an invariant $s.invariant$. The invariant is a logical formula, which models the state's assumption on the environment. When a state's invariant is violated, it means that the application's understanding to the environment is no longer consistent with the actual environment. Then a consistency failure occurs.

An invariant can be any constraint or condition formulated for modeling a state's assumption on the environment. For example, it can be a context consistency constraint [21, 23] that ensures contexts to be conflict-free for a state. It can be a guard condition [15, 20] that serves as the precondition for a state to be set as the current state, too. The invariant associated with a state allows explicit specification of the state's assumption on its environment, and enables automatic detection of consistency failures. The specifying and detecting process is essentially independent of the rematching strategy.

Take the preceding mentioned robot car application for example. At state S_1, the car walks forward to find the obstacle on its right-hand. Rule r_2 keeps the car walking forward when its right-hand area is open, and once $disR$ is smaller than 20, the car transits to the new state S_2 according to r_3. It is assumed that the car's front area should be open at state S_1, and the process to find the right-hand obstacle should not take too long (the distance should be smaller than 60). Let $disT$ measures how far the car has travelled since it transits from previous state to the current state. Then this assumption can be formulated as an invariant of state $''disF \geq 20\&\&disT < 60''$. Other states' invariants are elaborated in Figure 1 in which each state is connected with the invariant by a dotted line. If this invariant is violated (e.g., $disT > 60$), the car is unlikely still walking in an open area to find the right-hand obstacle. It may already pass the open area and its right-hand obstacle may be taken away, but it does not know the situation itself. In this case, a consistency failure occurs.

The preceding example illustrates how consistency failures occur by violating invariants. Consistency failures indicate that the car's understandings of their environments no longer match the actual environments. If consistency failures are ignored, the car is unlikely to continue to make correct adaptation in the future. Take our preceding consistency failure for instance. The application is at S_1, and the car may keep looking for its right-hand obstacle, until it bumps into its front obstacle by executing r_2 (since it thinks its right-hand obstacle has not been found yet).

2.3 Application Failures

Application failures are always observable and may result in severe consequences with certain application semantics at runtime. Design faults may lead to failures, and some faults are hard to identify and find. However, it is easy to recognize failures, such as crash, freezing, and stack overflow.

Application failures, which include some specific failures with certain application semantics like safety and time requirements, are more specific than failures. Consider the robot car application. It is unacceptable for the car bumping into any obstacles, which is a safety requirement. Meanwhile, turning round and round in one fixed place is unacceptable for the car, too. The above examples are two application failures for the robot car application. Different applications may have different application failures. The application failure is the last thing we want to see, and should be avoided as far as possible.

We studied the correlation in [24], and got interesting results that 27.7-99.2% consistency failures led to application failures, and 51.0-95.2% application failures were accompanied by consistency failures, through studying 12 different robot car model-based self-adaptive applications in both simulated and real environment. It means that there is a strong correlation between consistency failures and application failures.

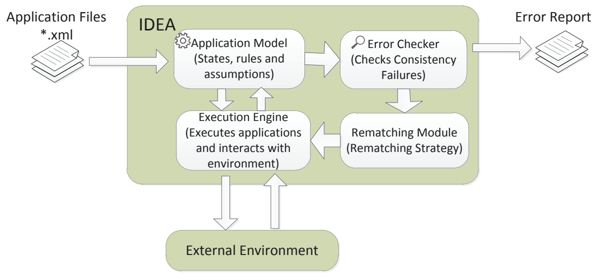

Application Files
*.xml
IDEA
Application Model (States, rules and assumptions)
Error Checker (Checks Consistency Failures)
Error Report
Execution Engine (Executes applications and interacts with environment)
Rematching Module (Rematching Strategy)
External Environment

Figure 2: Tool architecture.

3. IMPLEMENTATION

In this section, we will talk about the implementation of IDEA. First, we describe the functionality, the design, and the rematching strategy used in the rematching module. Then we describe the implementation details.

3.1 Functionality and Design

The tool IDEA is used to support detecting and fixing consistency failures at runtime for model-based self-adaptive applications, so as to prevent applications falling into application failures while in running. It also has an execution engine to support running the model-based self-adaptive applications.

IDEA takes an application's model specified in XML files with specific format as input. The execution engine interacts with the external environment. It obtains environmental attribute values from the environment and controls devices (e.g., robot cars) to influence the environment in return. And an error checker will check the assumptions' invariants of states with new environmental attribute values continuously. Furthermore, if a consistency failure is detected, the rematching module will be triggered to adopt necessary measures based on the rematching strategy to rematch the application with the environment. At the same time, the consistency failure will be recorded in the error report. Figure2 illustrates the architecture of our tool IDEA.

The transition rules' conditions and states' assumptions in applications' models can be expressed in first order logic. The execution engine will check the conditions associated with the current state to select a rule to execute. The error checker will check the invariant of the current state before executing transition rules. If a consistency failure is detected, the rematching module will be called.

3.2 Rematching Strategy

If a consistency failure is detected, the current state of the concerned model-based self-adaptive application is problematic, for its associated invariant has been violated, which means its understanding to the environment is inconsistent with the actual environment. In order to fix this consistency failure, it is intuitive to transit the application to another proper state whose invariant is not violated. However, an application's current state cannot be changed by force arbitrarily. It will cause side effects that we do not want to see. Therefore, the application should be rematched to the environment reasonably both logically and physically. In

Algorithm 1 Rematching strategy algorithm.

```
Input:
    $M$ (Model-based self-adaptive application)
Output:
    $rr$ (rematching result)
 1: while ($s.invariant$) do
 2:   if ($s.br > 0$ and !firstState($t$, $s$)) then
 3:     //Backward rematchable and not rematching the
        beginning yet, $t$(history execution trace),$s$(the cur-
        rent state)
 4:     $t$ := cancelOne($M$, $t$) //Cancel the last step
 5:     $s$ := lastState(t) //May still be itself
 6:   else if ($s.fr > 0$ ) then
 7:     //Forward rematchable
 8:     find $r$, $s'$ such that connect($r$, $s$, $s'$) and $s'.fr ==$
        $max\{s''.fr \mid connect(?, s, s'')\}$
 9:     $t$ := completeOne($M$, $t$, $r$) //Complete a new step
10:     $s := s'$
11:   else
12:     $rr$ := false
13:     break
14:   end if
15:   $rr$ := true
16: end while
17: return rr
```

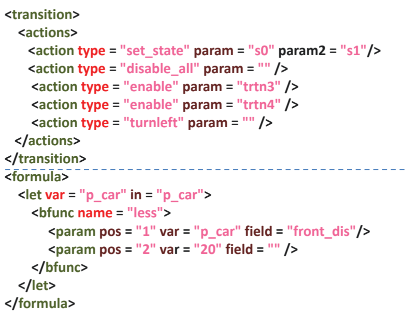
[27] and [24], we introduced the model-based self-adaptive application's atomic semantics of actions in the rematching strategy to achieve the above goal. Actions' atomicity means that actions taken in adaption should be totally completed or cancelled, aka "all-or-nothing" semantics in transactional processes [18, 26].

Consider a model-based self-adaptive application's execution trace: $s_0 r_0 s_1 \ldots r_n s_n$ ($s_0, s_1, \ldots, s_n$ are states, and $r_0, ..., r_n$ are rules), either the forward rematching or the backward rematching can be brought in for s_n, whose invariant is violated. The application can transit to its future state s_{n+1} by forward rematching provided that all actions taken in executing rule r_{n+1} can be guaranteed to complete, and can transit back to its earlier state s_{n-1} by backward rematching provided that all actions taken in executing rule r_n can be totally cancelled. If the application transits to state s_{n+1} or s_{n-1}, and the states' invariants are not hold, the application can consider transiting back to a earlier state or a future state until the state's invariant is satisfied. The rematching strategy algorithm is given in Algorithm3.1 to describe this process. The algorithm draws support from the rematching ability analysis algorithm in [24]. The rematching ability analysis algorithm analyzes states' forward and backward rematching abilities (results are expressed by $s.fr$ and $s.br$). The algorithm of rematching strategy prefers backward rematching (Lines 3-5) to forward rematching (Lines 7-10) in a conservative way. Function cancelOne compensates all actions in the last adaptation. Function completeOne completes all actions in the selected adaptation.

3.3 Implementation Details

Implementation details of tool IDEA are presented in this part. IDEA (developed in JAVA) mainly consists of four function modules (application model resolving module, execution engine, error checker and rematching module).

The execution engine is a core module in IDEA, which drives the whole process. The application model resolving module is called by the execution engine to receive and resolve the application files. Specific format XML files are supported, and the format is easy to understand and use.



```
<transition>
  <actions>
    <action type = "set_state" param = "s0" param2 = "s1"/>
    <action type = "disable_all" param = "" />
     <action type = "enable" param = "trtn3" />
     <action type = "enable" param = "trtn4" />
     <action type = "turnleft" param = "" />
  </actions>
</transition>
<formula>
  <let var = "p_car" in = "p_car">
    <bfunc name = "less">
       <param pos = "1" var = "p_car" field = "front_dis"/>
       <param pos = "2" var = "20" field = "" />
    </bfunc>
  </let>
</formula>
```

Figure 3: A snapshot of the application's XML files.

Figure 3 shows a snapshot of the files describing a robot car application's transition rule. The actions are illustrated in the top of Figure 3, and four types of actions (update states, disable rules, enable rules and turn left) are showed. Figure 3 (bottom) illustrates the condition, which expresses that the distance between the car and its front obstacles is less 20. The input files will be checked for the format and then we use dom4j [1] to resolve the XML files. After the application model is understood by IDEA, the execution engine can execute the application by interacting with the environment.

Note that the interaction needs support from underlying hardware platforms or other middleware to collect and manage the environmental attribute values (contexts). Different model-based self-adaptive applications may involve different hardware platforms. In Section 4, we state the experiment plan, and then introduce both certain hardware platforms and simulated environments about the robot car application.

The invariant of the current state is checked before executing every selected transition rule. If invariants are unsatisfied, consistency failures will occur and then the rematching module will be called. At the same time, this information is recorded in the error report. Proper rematching decisions will be made by the rematching module, and then the rematching actions will be executed by the execution engine according to the decisions. In order to complete the rematching actions, the execution engine needs to interact with the environment and guarantee the rematching actions can be totally cancelled or complete. It also needs hardware platforms' support. For example, our aforementioned robot car application's hardware is based on Lego NXT car [2]. And the application uses backward/turning-right/turning-left movements to totally cancel forward/turning-left/turning-right movements. It uses a speed sensor and a digital compass to guarantee forward/turning-left/turning-right movements to be totally completed. To support the robot car application, we realize a function in the execution engine to collect and manage environmental attribute values related to the robot car. The execution engine can support some applications (e.g., the robot car application) now, and we will extend it to support more model-based self-adaptive applications. Other middleware could be used to achieve this aim like CABOT [21].

Table 1: Failure rate reduction on 5 robot-car apps.

Config.	App1	App2	App3	App4	App5
Ideal	16.8% (−1.6%)	25.6% (−15.2%)	29.2% (−6.0%)	2.4% (−2.0%)	28.0% (−2.8%)
Noise	46.8% (−9.6%)	47.2% (−12.0%)	52.0% (−6.4%)	20.0% (−10.4%)	37.0% (−5.6%)
Dynamic	64.0% (−15.6%)	71.2% (−30.8%)	84.4% (−25.6%)	12.4% (−4.0%)	64.8% (−17.6%)
Severe	81.6% (−7.6%)	80.8% (−13.2%)	95.6% (−18.0%)	55.6% (−8.0%)	82.4% (−11.6%)
Real	88.0% (−38.0%)	75.0% (−25.0%)	75.0% (−50.0%)	100.0% (−50.0%)	75.0% (−50.0%)

4. EXPERIMENTS AND FUTURE WORK

We are going to conduct experiments to evaluate our approach's performance and cost, on the robot car application and other model-based self-adaptive applications. Detailed experiment plan will be elaborated in this section.

The subjects of our experiments are model-based self-adaptive applications, mainly the robot car applications here. We have 12 different robot car applications which are independently developed by different research staffs and students in our university during the past three years. The total number of states and transition rules in each application varies from 17 to 40. We prepare to compare our approach with three other strategies: no rematching, directly jumping to a state whose invariant holds, and rematching but ignoring its rematching ability analysis result (arbitrary forward rematching or backward rematching). The goal of the last two is to justify why we have to guarantee application semantics atomic in rematching.

The environment of the experiments for the robot car application can either be simulated or real. We simulated four different scenarios by introducing noise and dynamics. The scenarios are *Ideal* (scenario without noise or dynamics), *Noise* (scenario with noise), *Dynamic* (scenario with dynamics), and *Severe* (scenario with noise and dynamics). The dynamics are simulated as unpredicted object movements for the applications. The real environment uses real hardware and scenario.

In total, we set five configurations for the experiment. We are going to run these robot car applications on four different hardware platforms including tank car, tricycle and other two different custom-made cars. The differences of those cars reflect in the success rate to complete actions and hardware failure rate. For example, tank car can be guaranteed to turn right and turn left precisely, but tricycle may generate some relatively large offset compared to the tank car's result. For each configuration in the simulation, we are ready to run each application for 250 times in every different hardware platform. It takes 2 minutes for each run in simulation. And for the real environment, we will conduct 8 times of runs of each application in every different hardware platform. And four strategies will be compared. To sum up, 61,920 application runs will be executed.

We conducted some preliminary experiments, and Table 1 illustrates the results. They are about five applications running in five configurations on the tricycle platform. We compared no rematching and our approach for the application failure rate, which is defined as the ratio of runs in which application failures occurred against all runs. Each datum takes the form of $''X(Y)''$. X is the original application failure rate and Y is the change of the rate after applying our approach. The results show that our rematching strategy is generally effective for reducing the failure rate.

The next research steps to be taken include completing the above experiments, and evaluating our approach in other respects. First, we will compare the cost of our rematching strategy with other approaches. Secondly, the scalability of our approach is going to be evaluated. Thirdly, we will study the influence on the rematching effects exerted by the quality of the invariants, and whether the users could define the invariants for the states. Furthermore, we want to extend our tool IDEA to support more model-based self-adaptive applications.

5. RELATED WORK

Self-adaptive applications interact with the environment frequently by sensors. But sensing technology is prone to errors. As a result, incomplete, imprecise, or even conflicting sensory data may be collected by the self-adaptive applications. It is uncertain to know whether those sensory data can fulfill functionalities [8]. Ramirez et al. [16] sorted the uncertain factors that can affect the dependability of self-adaptive applications. Feedback-based control [5], exception-monitored framework [9], or reflective middleware [7] can be used to handle uncertainty in self-adaptive applications. And developers need to consider how to adequately and properly model adaptation in self-adaptive applications. Andersson et al. [3] discussed a set of modeling dimensions. A research roadmap is introduced by Cheng et al. [6] for related engineering issues. Kramer and Magee [11] discussed architectural challenges and potential solutions.

Existing work about fault detection in self-adaptive applications focuses on different levels (model level and code level). Sama et al. [17] made use of model checking to search applications' state space to detect faults. Xu et al. [20] used error patterns to track and analyze responsible faults and Liu et al. [13] used machine learning to derive deterministic and likely constraints to prune false warnings and prioritize true faults in model level. In code level, new coverage criteria was imported by Lu et al. [14] to test self-adaptive applications, and Wang et al. [19] strengthened test cases by focusing on context switching points.

There has been much work about runtime error handling for self-adaptive applications. [7] and [10] paid attention to data. The first fixed sensory data errors probabilistically with as many integrity constraints satisfied as possible, and the latter fixed data structure errors based on specifications. To improve application dependability, Garlan et al. [9] used architecture-based self-repairing. Xu et al. [22] fixed context errors and prevented application semantics from being affected. Some work considers applications composed from service components. Schuldt et al. [18] required atomicity of transactional systems to be protected from composition errors. Ye et al. [26] discussed the atomicity of composed services. In our previous work [24, 27], we extended atomicity to self-adaptive applications. Boos et al. [4] checked assertions for self-adaptive applications, which echoes our idea to use invariants to check whether an application's state is consistent with environment. Yang et al. [25] isolated an application from environment rather than matching it with environment.

6. SUMMARY

Self-adaptive applications may fail at runtime because their understandings to the environments do not match the actual environments. This may happen when sensors do not work properly, or when the environment conditions change rapidly. We propose an approach to reducing application failures by fixing consistency failures, since there is a high correlation between consistency and application failures. In this approach, the application's states whose invariant are violated can be matched again to the actual environment by forward or backward rematching. A tool named IDEA was developed to realize the approach, and was evaluated on a set of 12 model-based self-adaptive robot car applications with different configurations. The results demonstrate that the failure rate of such applications can be significantly reduced. In the future, we are going to evaluate the approach from more aspects.

7. ACKNOWLEDGMENTS

This research was partially funded by National High-tech R&D 863 Program (No. 2012AA011205) and National Natural Science Foundation (Nos. 61100038, 91318301, 6136112-0097) of China. Chang Xu was also partially supported by Program for New Century Excellent Talents in University, China (No. NCET-10-0486).

8. REFERENCES

[1] Dom4j. http://dom4j.sourceforge.net/.

[2] Lego. http://mindstorms.lego.com/en-us/default.aspx.

[3] J. Andersson, R. Lemos, S. Malek, and D. Weyns. Modeling dimensions of self-adaptive software systems. In *Software Engineering for Self-Adaptive Systems*, pages 27–47. Springer Berlin Heidelberg, 2009.

[4] K. Boos, C. L. Fok, C. Julien, and M. Kim. Brace: An assertion framework for debugging cyber-physical systems. In *Proc. ICSE' 12*, pages 1341–1344, 2012.

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

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

[7] B. Demsky and M. Rinard. Goal-directed reasoning for specification-based data structure repair. *IEEE Trans. Softw. Eng.*, 32(12):931–951, 2006.

[8] N. Esfahani, E. Kouroshfar, and S. Malek. Taming uncertainty in self-adaptive software. In *Proc. ESEC/FSE' 11*, pages 234–244, New York, USA, 2011.

[9] D. Garlan, S.-W. Cheng, and B. Schmerl. Increasing system dependability through architecture-based self-repair. In *Architecting Dependable Systems*, pages 61–89. Springer Berlin Heidelberg, 2003.

[10] N. Khoussainova, M. Balazinska, and D. Suciu. Towards correcting input data errors probabilistically using integrity constraints. In *Proceedings of the 5th ACM international workshop on Data engineering for wireless and mobile access*, pages 43–50, New York, USA, 2006.

[11] J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In *Proc. Future of Software Engineering, ICSE' 07*, pages 259–268, 2007.

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

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

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

[15] A. L. Murphy, G. P. Picco, and G.-C. Roman. Lime: A coordination model and middleware supporting mobility of hosts and agents. *ACM Trans. Softw. Eng. Methodol.*, 15(3):279–328, July 2006.

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

[17] 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.

[18] H. Schuldt, G. Alonso, C. Beeri, and H.-J. Schek. Atomicity and isolation for transactional processes. *ACM Trans. Database Syst.*, 27(1):63–116, Mar. 2002.

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

[20] 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.

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

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

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

[24] C. Xu, W. Yang, X. Ma, C. Cao, and J. Lu. Environment rematching: toward dependability improvement for self-adaptive applications. In *Proc. ASE' 13*, 2013, forthcoming.

[25] H. Yang, C. Xu, X. Ma, L. Zhang, C. Cao, and J. Lu. Consview: Towards application-specific consistent context views. In *Proc. COMPSAC' 12*, pages 632–637, 2012.

[26] C. Ye, S. Cheung, W. Chan, and C. Xu. Atomicity analysis of service composition across organizations. *IEEE Trans. Softw. Eng.*, 35(1):2–28, 2009.

[27] 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. APSEC' 12*, pages 184–193, 2012.