Requirements-Driven Self-Repairing against Environmental Failures

Rui-Zhi Dong[1] *, Xin Peng[1] , Yi-Jun Yu[2], Wen-Yun Zhao[1]
[1] School of Computer Science, Fudan University, Shanghai 201203, China
[2] Department of Computing, The Open University, Milton Keynes MK7, United Kingdom
Email: {09110240009, pengxin, wyzhao}@fudan.edu.cn; y.yu@open.ac.uk

Abstract— **Self-repairing approaches have been proposed to alleviate the runtime requirements satisfaction problem by switching to appropriate alternative solutions according to the feedback monitored. However, little has been done formally on analyzing the relations between specific environmental failures and corresponding repairing decisions, making it a challenge to derive a set of alternative solutions to withstand possible environmental failures at runtime. To address these challenges, we propose a requirements-driven self-repairing approach against environmental failures, which combines both development-time and runtime techniques. At the development phase, in a stepwise manner, we formally analyze the issue of self-repairing against environmental failures with the support of the model checking technique, and then design a sufficient and necessary set of alternative solutions to withstand possible environmental failures. The runtime part is a runtime self-repairing mechanism that monitors the operating environment for unsatisfiable situations, and makes self-repairing decisions among alternative solutions in response to the detected environmental failures.**

I. Introduction

The satisfaction of a requirement of software-intensive systems relies on the assumptions about the behaviors of both the software components and their environment [1], [2]. Accordingly, a system specification must be based on these assumptions: specifically, *environmental failures* (i.e., environmental components failing to meet relevant assumptions) may cause either system failures or quality degradation. For example, the performance of the service may degrade when the available network bandwidth is less than what is expected.

To alleviate the problem of such failures, self-repairing [3], [4] systems that automatically detect, diagnose, and repair localized software and hardware problems have been proposed as an important capability of self-management. Approaches have been proposed to achieve the purpose of self-repairing using requirements-driven reconfiguration [4], pre-defined Event-Condition-Action (ECA) rules [5], [6] etc. Assume the availability of alternative solutions for various potential failures, however, such approaches lack a systematic analysis of the relationship between environmental failures and derivation of the corresponding alternative solutions.

To derive alternative solutions from the analysis of failure conditions, in this paper, we propose a requirements-driven self-repairing approach against environmental failures, applying both development-time and runtime techniques. The development-time part stepwise derives alternative solutions by applying formal analysis with the support of a model checker. The runtime part is a runtime self-repairing mechanism that monitors the operating environment for unsatisfiable situations, analyzes the aftereffects of any detected failures, and makes switching decisions among alternative solutions in response to environmental failures.

II. Preliminaries

A. Problem Frames

In our approach, we use the Problem Frames (PF) approach as our underlying modeling framework. The PF approach [1] provides a basis for analyzing the structure of software problems and their context. In this approach, a software problem comprises three set of descriptions: (1) a description of the context in which the problem resides in term of known domain properties of the world, denoted by W; (2) a description of the required properties of the world expressing the requirements of the stakeholders, denoted by R; and (3) a description specifying the software-to-be (namely machines) must do so as to meet the requirements, denoted by S. The approach defines requirements engineering as the problem of finding the specification of the software-to-be to ensure the given requirements in the given context.

The PF approach provides a series of tools to analyze the structure of a software problem, involving context diagram and problem diagram. A context diagram captures the environment by decomposing a software problem intr physically connected domains, whilst a problem diagram which specialises the settings for the context helps to structure and analyze the requirement.

After structuring a software problem, we must seek for a solution to ensure the desired requirement under the given context. Typically, we can use *problem progression* [1] to derive a solution for a problem. Now several approaches have been proposed to achieve problem progression, such as requirements progression [7] etc.

B. Dependability Arguments with Trusted Bases

Based the PF approach, Kang et al [2] propose the concept of *trusted base* to represent the subset of components in the context which the satisfaction of a requirement relies on. A *trusted base* for a requirement is a set of domains and machines that are crucial to establish the requirements regardless of how the domains outside the trusted base behave.

With the introduction of trusted base, the argument of the requirements in the problem frames approach is transformed into dependability arguments with trusted bases. The dependability argument with trusted bases [2] refers to determining whether or not the elements in the trusted base is sufficient to establish the desired requirement.

Typically, the trusted base for a requirement can be automatically derived by formalization tools (such as model checkers and theorem proofers) which are capable of finding the unsatisfiable core for a specific proposition. The unsatisfiable core for a proposition [8] is the set of constraints crucial to the satisfaction of the preferred proposition. Kang et al [2] investigate the relationship between the unsatisfiable core and trusted base, and find that the trusted base for a requirement is the set of elements (namely, domains and machines) whose constraints belong to the unsatisfiable core for the proposition expressing the argument of the specified requirement.

III. Rationale

Considering the issue of runtime environmental failures, to ensure the continuous satisfaction of the critical requirement denoted by F, an intuitive self-repairing policy is to switch to an alternative solution that can tolerate the detected environmental failures and implement the same requirements with possibly degraded quality. Therefore, the adaptation knowledge base shall involve different ways to ensure F. Each mean to meet F refers to a software problem, and involves different machine specifications which depends on different environmental components.

Definition 1: *(Rephrased Requirement) Given P_i denoting a software problem to find a solution to ensure F, $Domains(P_i)$ denoting the set of domains involved in P_i, $DA(P_i)$ denoting domain properties of elements in $Domains(P_i)$, the rephrased requirement denoted by R_i represents the rephrased expressions of F in terms of domain properties of components within $Domains(P_i)$.*

Next, we can develop a set of alternative solutions to ensure F. Each solution depends on a different set of domains. Therefore, considering the definition of *trusted base*, the trusted base of F has multiple candidates.

Although the means to define alternative solution for a critical requirement has been determined, the criteria to evaluate whether or not the current set of alternative solutions are sufficient so as to withstand possible environmental failures is kept unexplored. To deal with such issue, we introduce the definition of *rephrased model* and try to adopt formal analysis.

Definition 2: *(Rephrased Model) Suppose a set of alternative solutions S (S= { S_1, ..., S_n }) to meet F have been developed. Given P_i denoting a software problem to find solution S_i, a rephrase model RM may be constructed by composing the descriptions of every variant problems at present, together with the rephrased requirementRF of F in terms of the domain phenomena within RM.*

After a rephrase model is developed, we update $TDA(RM, RF)$ and then evaluate its acceptance by domain experts. If the current $TDA(RM, RF)$ is acceptable, the current set of alternative solutions is sufficient to ensure F. Otherwise, more solutions are to be introduced. In this paper, we adopt model checking technique to complete this task.

In order to define the monitoring requirements specifying how to monitor environmental components, we introduce the definition of *Trusted Domain Assumption* together with *Corollary 3*.

Definition 3: *(Trusted Domain Assumption) Given P_i denoting a software problem to find a solution to ensure F, R_i denoting the rephrased requirement of F, S_i denoting a solution to meet R_i, $DA(P_i)$ representing the domain assumptions relevant to P_i, the trusted domain assumptions $TDA(P_i, R_i)$ are domain assumptions crucial to the satisfaction of S_i.*

Considering the definitions of unsatisfiable core, $TDA(P_i, R_i)$ only involves a subset of constraints within the unsatisfiable core of the argument of requirement R_i. As a result, whenever the unsatisfiable core for the argument of R_i is identified, $TDA(P_i, R_i)$ can be determined.

Theorem 1: *(Monitoring Condition) Given S_i denoting a solution to ensure the rephrased requirement R_i of F, P_i denoting the software problem to find S_i, $\forall\ d \in Domains(P_i) \bigcap TB(P_i, R_i)$ are domains to be monitored, and domain assumptions in the set of $TDA(P_i, R_i)$ are constraints which shall be involved in the definition of the monitoring requirement.*

Next, to make preparation for runtime adaptation against environmental failures, we define the rules concerning adaption conditions and switching conditions. Here, the adaptation conditions are determined by evaluating whether or not any trusted domain assumptions become invalidated so as to determine whether or not an adaptation is required, whilst the switching conditions refer to find an alternative solution that can tolerate the detected environmental failures.

Theorem 2: *(Adaptation Condition) Given S_i denoting a solution currently adopted, FDA denoting the set of domain assumptions which are detected to be invalidated during the runtime, P_i representing the software problem to find S_i, R_i representing the rephrased requirement of F, an adaptation is needed if and only if $FDA \cap TDA(P_i, R_i) \neq \Phi$.*

When an adaptation is required, we may transfer to another solution which can tolerate the detected environmental failures.

Theorem 3: *(Switching Condition) Suppose that the detected environmental failures make the currently adopted solution S_i. Given i $\neq$ j, $S_i \neq S_j$, S_j denoting a candidate solution to ensure F, R_i representing the rephrased requirements of F in the software problem P_i concerning finding S_i, R_j representing the rephrased requirements of F in the software problem P_j concerning finding S_j, $FDomains(P_i, R_i)$ denoting the set of domains which are detected to misbehave during the runtime, a shift from S_i to S_j is applicable if $FDomains(P_i, R_i) \notin TB(P_j, R_j)$.*

Proof. Because $FDomains(P_i, R_i) \notin TB(P_j, R_j)$, those misbehaved domains do not belong to $TB(P_j, R_j)$. Then, according to the definition of trusted base, the invalided domains have no aftereffect to the satisfaction of R_i. Therefore, a shift from S_i to S_j will ensure the continuous satisfaction of F. $\square$

At runtime, there may be conditions that multiple alternative solutions can be adopted. In such cases, we shall select a solution with the best quality satisfaction guided by an applicable utility function. Therefore, we shall investigate the contribution links between quality requirements and a solution in advance. Following the method of Pent et al [9], we may investigate and specify the contribution relationships between a solution and the preferred quality requirements.

IV. Our approach

In this section, we first present an overview of our approach, and then introduce the two key techniques of our method, i.e., the derivation of alternative solutions at development time and self-repairing decision making at runtime.

A. Overview

To achieve self-repairing against environmental failures, we propose a requirements-driven approach as shown in Figure 1. Our approach involves both development and runtime phases. The objective of the development phase is to derive a series of alternative solutions that can tolerate a comprehensive set of environmental failures. At runtime, when environmental failures are detected, the self-repairing mechanism analyzes the impact and makes self-repairing decisions by switching to an applicable solution accordingly.

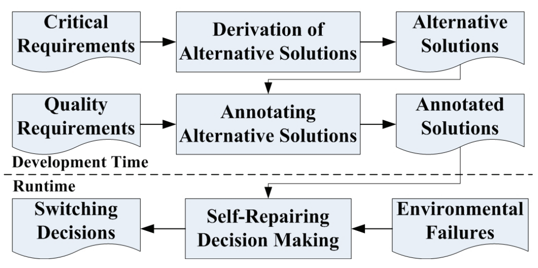

Fig. 1. An overview of our approach

The development part of our approach begins with the identification of the critical requirement F and its related quality attributes. To ensure the satisfaction of F under different kinds of environmental failures at runtime, we take an iterative process (as shown in Figure 2) to derive a set of alternative solutions. Within each iteration, a variant problem, which incorporates a new alternative solution for F, is derived from the descriptions of the motivating problem for F. Based on the description of all variant problems at present, we update update trusted domain assumptions $TDA(RM, RF)$ for F. And then, we evaluate the acceptance of the current $TDA(F)$. After evaluation, if the current set of $TDA(RM, RF)$ are deemed to be acceptable or no more variant problems can be derived, the iterative process for F ends with a set of alternative solutions. Finally, for each of identified alternative solution, we analyze its contributions to the preferred quality attributes. The set of alternative solutions together with their trusted domain assumptions and the annotated quality contributions are then used as the knowledge base for runtime self-repairing.

The runtime part of our approach is a self-repairing mechanism with a planning algorithm. When environmental failures are detected at runtime, the planning process evaluates whether or not the detected environmental failures have influences on the solution currently adopted, and then makes switching decisions by choosing an applicable solution if necessary.

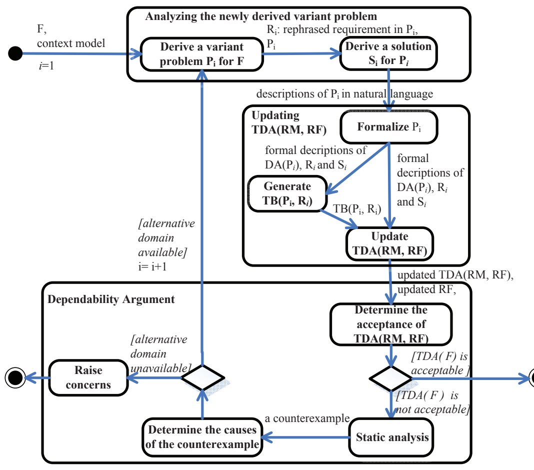
Analyzing the newly derived variant problem
F, context model
i=1
Derive a variant problem P_i for F
R_i: rephrased requirement in P_i, P_i
Derive a solution S_i for P_i
descriptions of P_i in natural language
Updating TDA(RM, RF)
Formalize P_i
formal decriptions of DA(P_i), R_i and S_i
formal decriptions of DA(P_i), R_i and S_i
Generate TB(P_i, R_i)
TB(P_i, R_i)
Update TDA(RM, RF)
[alternative domain available] i= i+1
updated TDA(RM, RF), updated RF,
Dependability Argument
Determine the acceptance of TDA(RM, RF)
[alternative domain unavailable]
Raise concerns
[TDA(F) is acceptable]
[TDA(F) is not acceptable]
a counterexample
Determine the causes of the counterexample
Static analysis

Fig. 2. Derivation of alterative solutions

B. Derivation of Alternative Solutions

1) Variant Problem Analysis

In the first iteration, based on the context model of the target system, an initial variant problem is usually identified by considering the most typical context settings for F. In subsequent iteration, when a new variant problem P_i is identified, we rephrase the critical requirement F in terms of the phenomena within P_i, resulting in a variant requirement R_i. Next, we use the technique of requirements progression [7] to derive a solution S_i for P_i and the corresponding domain assumptions $DA(P_i, R_i)$.

2) Updating Trusted Domain Assumptions

To make preparation for the update of $TDA(RM, RF)$, we firstly formalize the descriptions of all variant problems at present. In this paper, we choose Alloy [10] as our modeling language. Alloy is a formal language based on first-order relational logic with an analysis engine, Alloy Analyzer, which allows bounded model checking and the discovery to finding unsatisfiable core.

Next, based on the formal descriptions of available variant problems, we construct the rephrased model RM for F, and update FM. Next, following the method described in paper [11], we update the trusted domain assumptions $TDA(RM, RF)$.

3) Dependability Arguments

After that, we argument the acceptance of the trusted domain assumptions $TDA(RM, RF)$ with domain experts. If the newly derived trusted domain assumptions are acceptable, we are convinced that current set of alternative solutions are sufficient to ensure F, and thus dependability arguments are completed. Otherwise, we shall analyze the aftereffects of those suspicious domain assumptions, and try to try to introduce new environmental components taking place of the domains which are relevant to those suspicious domain assumptions.

For example, we may construct and run an assert in Alloy expressing the situation that those suspicious domain assumptions become invalid, and thus a counterexample is returned. The analysis of the derived counterexample will help determine what the substitute for domains concerning suspicious domain assumptions should behave.

Considering that the characteristics of different domains may differ a lot, we identify two patterns to cope with such issue: (1)if the domains involving suspicious domain assumptions have no substitutes due to some limitation, we have to tolerate the failures of the domain; (2)otherwise, alternatives for those domains are introduced.

C. Self-Repairing Decision Making at Runtime

In order to cope with such issue, a *Runtime planning algorithm*(as shown in Figure 3) is proposed. In the control loop, we firstly communicate with deployed sensors and/or monitors to collect information about the operating environment, and thus the occurrence of environmental failures are determined (Line 1). Secondly, we evaluate the aftereffects of the detected environmental failures by accessing whether or not any elements in the set of invalidated environmental components belong to the trusted base of the solution currently adopted (Line 2). If the set of invalidated environmental components do not intersect that of the trusted base, no adaptation is required and thus the current solution is maintained (Line 3); otherwise, an adaptation action is required, and then an applicable alternative solution is to be selected (Line 4-20). To make preparation for finding an appropriate solution, we firstly select a subset of applicable alternative solutions that are free of detected environmental failures, and thus candidates are filtered (Line 5-8). If there exist multiple competing alternative solutions, we compare computed values of those candidates across domain-specific utility function, and thus the alternative with highest scores is chosen (Line 11-20). But if there is no elements in the set of the computed candidates, we have to tolerate detected failures (Line 9-10), and thus the current solution is maintained.

V. Conclusion and Future Work

To have a software system withstand environmental failures, a requirements-driven self-repairing method has been proposed. Furthermore, we demonstrate our method on a case study of a mobile application. Results of simulation experiments show the effectiveness of our approach.

Since environmental failures are often a consequence of accumulating of ever-changing resources, we plan to investigate the relationships between resource changes and environmental failures, and to exploit the relationships between them by sensitivity analysis with the support of the system dynamics technique.

```
Input:  F denoting the preferred critical requirement,
        $S_0$ denoting the solution currrently adopted,
        $P_0$ denoting the software problem to seek $S_0$,
        $R_0$ denoting the rephrased requirement of F in $P_0$,
        $S=\{S_1,...,S_n\}$ denoting a set of candidate solutions for F,
        $P_i$ denoting the rephrased requirement of F in $P_i$,
        Q representing the set of preferred quality factors,
        alternatives representing the set of filtered solutions
Output: candidate representing an applicable solution to ensure F
1  invalidDomains = getFailedEnvComponent();
2  if(invalidDomains $\cap$ TB($P_0$, $R_0$) == null)
3       candidate = $S_0$;
4  else{
5       for( each $S_i \in S$ ){
6           if( invalidDomains $\cap$ TB($P_i$, $R_0$) == null)
7              let $S_i \in$ alternatives;
8       }
9       if( #(alternatives) ==0){
10        candidate = $S_0$
11      }else{
12             int i, max =0;
13             for(each $S_j \in$ alternatives){
14                 i=getUtilityValue($S_j$, Q);
15                 if(i >max)
16                     max=i
17             }
18        }
19      candidate = getSolution(max, alternatives);
20 }
21 return candidate;
```

Fig. 3. Runtime adaptation algorithm

References

[1] M. Jackson, *Problem frames: analysing and structuring software development problems*. Addison-Wesley, 2001.

[2] E. Kang and D. Jackson, "Dependability arguments with trusted bases," in *Requirements Engineering Conference (RE), 2010 18th IEEE International*. IEEE, 2010, pp. 262–271.

[3] J. Kephart and D. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[4] Y. Wang and J. Mylopoulos, "Self-repair through reconfiguration: A requirements engineering approach," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2009, pp. 257–268.

[5] J. Keeney and V. Cahill, "Chisel: A policy-driven, context-aware, dynamic adaptation framework," in *Policies for Distributed Systems and Networks, 2003. Proceedings. POLICY 2003. IEEE 4th International Workshop on*. IEEE, 2003, pp. 3–14.

[6] D. Garlan, S. Cheng, A. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-based self-adaptation with reusable infrastructure," *Computer*, vol. 37, no. 10, pp. 46–54, 2004.

[7] R. Seater, D. Jackson, and R. Gheyi, "Requirement progression in problem frames: deriving specifications from requirements," *Requirements Engineering*, vol. 12, no. 2, pp. 77–102, 2007.

[8] E. Torlak, F. Chang, and D. Jackson, "Finding minimal unsatisfiable cores of declarative specifications," *FM 2008: Formal Methods*, pp. 326–341, 2008.

[9] X. Peng, Y. Yu, and W. Zhao, "Analyzing evolution of variability in a software product line: from contexts and requirements to features," *Information and Software Technology*, vol. 10, no. 7, pp. 707–721, 2011.

[10] D. Jackson, *Software Abstractions: logic, language and analysis*. MIT Press (MA), 2012.

[11] E. Kang, "A framework for dependability analysis of software systems with trusted bases," M. Eng. thesis, Massachusetts Institute of Technology, Cambridge, USA, Feb. 2010.