Adaptive Runtime Verification for Autonomic Communication Infrastructures

Giovanni Denaro, Leonardo Mariani, Mauro Pezzè, Davide Tosi
Universitá degli Studi di Milano Bicocca
Dipartimento di Informatica, Sistemistica e Comunicazione
Via Bicocca degli Arcimboldi 8
I-20126, Milano - Italy
{denaro|mariani|pezze|tosi}@disco.unimib.it

Abstract

Autonomic communication and autonomic computing can solve many problems in managing complex network and computer systems, as well as network applications, where computing and networking co-exist. Autonomic applications must be able to automatically diagnose and repair their own faults. In particular they must be able to monitor the execution state, understand the behavior of the application and of the executing environment, and interpret monitored data to identify faults and select a repairing strategy.

Assertions have been extensively studied in software engineering for identifying deviations from the expected behaviors and thus signal anomalous outcomes. Unfortunately, classic assertions are defined statically at development time and cannot capture unpredictable changes and evolutions in the execution environment. Thus they do not easily adapt to autonomic applications.

This paper proposes a method for the automatic synthesis and adaptation of assertions from the observed behavior of an application, aimed at achieving adaptive application monitoring. We believe that this represents an important basis to derive autonomic mechanisms that can deal with unpredictable situations.

1 Introduction

The networking and the software engineering communities are increasingly studying autonomic communication and autonomic computing as possible solutions to the problems involved in managing networks and software that grow beyond the control of single development team. Autonomic networks and software systems will be able to autonomously adapt to unpredictable environmental changes, by exploiting embedded self-managing capabilities such as self-organization, self-configuration, self-regulation, self-adaptation, self-healing and self-protection [11].

Autonomic web applications merge autonomic networking and software features to overcome many limitations of current communication technology, and thus meet the increasingly demands of the Internet of the future.

An important characteristic of autonomic applications is the ability to automatically diagnose and repair their own faults. Fault diagnosing requires (1) monitoring the execution state, to understand the behavior of the application and of the executing environment, and (2) interpreting the monitored data, to identify faults and select a repairing strategy.

Common monitoring mechanisms instrument the software and the environment with probes or gauges to capture runtime data, e.g., control and dataflow traversals, changes of values, performance indexes, and changes of context information. As for data interpretation, many approaches rely on embedded assertions, that is, constraints over the collected data at given execution points.

Assertions have been studied in software engineering for many purposes, e.g., debugging, testing and runtime verification [19], and are supported by many tools (e.g., [1]). The applicability of assertions to autonomic applications is limited by the need of specifying the assertions in advance, which prevents the possibility of dealing with unpredictable changes and problems.

This paper proposes the automatic synthesis of assertions from the observed behavior of the application to derive autonomic mechanisms that can deal with unpredictable situations. In previous work, we exploited dynamic analysis for verifying component-based systems and we developed and experimented a technique, called Behavior Capture and Test (BCT), to synthesize and use invariants that describe both input/output properties and interaction patterns of software components [15]. BCT automatically infers invariants that describe the runtime behavior of components, and then uses the inferred invariants to detect inconsistent behaviors in new versions or in new uses of the components. BCT works well for components that are reused without information about the source code and without precise specifica-

tions of the components' internals. In this paper we show that BCT adapts well to dynamically changing systems that present unpredictable behaviors, as in the case of autonomic applications.

This paper is organized as follows. Section 2 gives details on the original BCT technique. Section 3 describes our proposal of adapting BCT for autonomic communication systems. Section 4 discusses related work, and section 5 summarizes the contribution of the paper and outlines our research agenda for the future.

2 Behavior Capture and Test

BCT, which is the basis for our proposal, automatically identifies behavioral differences of evolving component-based systems. In particular, BCT can reveal incompatibilities of upgraded and reused components.

The technique is applied in two main phases: data collection and invariant checking. In the data collection phase, components are monitored, while used as part of running applications, and information about both the data exchanged and the interactions between components is recorded. The recorded information is then used to automatically distill invariants, which characterize the observed behavior. In the invariant checking phase, the invariants inferred in the previous phase are used to dynamically identify incompatibilities between the behavior represented by the invariants and new uses of components.

In the data collection phase, BCT derives two types of invariants: I/O invariants and interaction invariants. I/O invariants describe properties of services as Boolean expressions that can be evaluated over the input parameters and the results. Interaction invariants specify the interactions among components as finite state machines. To distill the invariants, we must be able to (1) deal with object references, which are difficult to analyze due to encapsulation, (2) derive I/O invariants from raw data, and (3) infer finite state machines from interaction samples.

BCT deals with object references by extracting information about the state of the object through heuristically identified inspectors, i.e., methods that return data about the state without modifying the state of the object. The heuristic consists of a set of syntactic rules that is used to check the signature of the methods. If a method satisfies a rule, it is selected as inspector (see [14] for details on the heuristic). Extraction recursively applies to objects returned as part of the state by inspectors until a primitive data value is collected, an already examined reference is returned, or the analysis reaches a given depth. For example, extracting information from a reference to an object `Person` produces the following output:

```
john.getFirstName = "John"
john.getSecondName = "Smith"
john.getAge = "12"
john.getAddress.getStreet = "street, 1"
john.getAddress.getCity = "New York"
john.getAddress.getCountry="US"
```

BCT distills I/O invariants from the traces obtained by analyzing the objects exchanged between components with the Daikon invariant inference engine [4], which generates Boolean expressions for a set of variables starting from a set of execution samples. The generated invariants define relations on objects' fields. For example, a set of executions that involved only `Persons` from US under the age of 15 is captured by the following I/O invariants:

```
john.getAddress.getAge <= 15
john.getAddress.getCountry == ''US''
```

BCT infers interaction invariants from traces that represent the sequences of services invoked by a component when a service is executed. For instance, a component `customerCare` may interact with an external library for sorting the customers, and with an external component for computing the unique Italian Social Security Number.[1] When the method `getListOfContacts` of `customerCare` is executed, `customerCare` interacts with both the external library to sort the customers, and with the external component to compute the Italian Social Security number for Italian customers. A possible set of traces for `getListOfContacts` is:

```
Sort SSN
Sort
Sort SSN SSN SSN SSN SSN SSN
Sort SSN SSN
...
```

where `Sort` indicates the invocation of method `Sort` and `SSN` indicates the invocation of the method that computes the Italian Social Security Number. A unique invocation of method `Sort` can be followed by many invocations of method `SSN`, one for each Italian customer.
Interaction invariants are captured with finite state machines that generalize the considered traces. For instance, BCT generates the FSM shown in Figure 1, from the above set of traces.

BCT infers finite state machines with an algorithm called kBehavior, which builds the FSM incrementally, considering each trace only once, thus traces do not need to be stored [13].

[1] The Italian Social Security Number, called Codice Fiscale, is required for commercial transactions and can be computed from personal data.

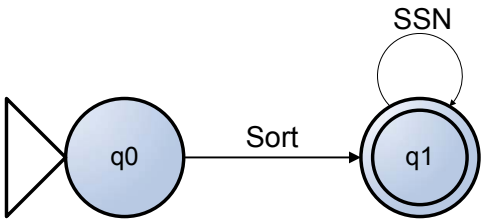

Figure 1. A FSM produced by BCT

In the invariant checking phase, BCT checks the behavior observed on a new system with respect to the invariants computed on an old system, which can be the system with an old version of an updated component, or a running system using a component that is now reused in the new system. In the case of component upgrade, BCT looks for violations of the invariants computed for the previous version of the component, and thus it signals anomalous behaviors that may either confirm expected upgrades, or indicate unexpected failures. In the case of components reused in new systems, BCT looks for violations of the invariants computed for the reused components when running as part of other systems, thus it signals new behaviors that may either indicate new legal behaviors in the different context or faulty interactions. BCT has been successfully validated with several case studies. Details can be found in [14].

3 Adaptive runtime verification

In modern networking systems, the execution environment of web applications changes dynamically and unpredictably, due to upgrades of the network protocols, modification of the network infrastructure, evolution of the set of available services, migration of software applications. As in the case of component-based software, changes often involve elements that have been already in use in other operative or testing contexts. Thus, as in the case of component-based software, we may take advantage of historic information about the components' behaviors to validate expected changes and identify potential faults. However, differently from the case of component-based software, networking systems evolve independently from the application developers, who cannot predict networking and environmental changes beforehand.

We argue that BCT can be successfully adapted to the new contexts, and in this section, we describe how to instantiate BCT to support autonomic communication infrastructures. In a nutshell, we propose to embed assertions in the communication infrastructures, to describe the legal interactions between the communicating entities. Such assertions are then checked at runtime to reveal misbehaviors, incompatibilities and unexpected interactions that may be due to hidden faults, changes in some components or malicious code. The main novel contribution of our approach is the ability of automatically synthesizing assertions that evolve over time and adapt to the new context-dependent interactions. The new technique, which we call *Adaptive Runtime Verification (ARV)* adapts BCT to the new context and is composed of three main phases:

Assertion synthesis: During the test of a new application, ARV uses BCT to monitor the execution of the application, and to produce invariants that describe the executed test cases. The collected invariants represent the interactions of the application with the testing environments, i.e., the interaction protocols that have been tested.

Assertion checking: At deployment time, ARV augments the applications with the computed invariants that are embedded in the underlying communication infrastructure in the form of assertions. The communication infrastructure checks the assertions at runtime, to signal interaction protocols not previously tested, and thus potentially dangerous. Notice that checking the assertions within the communication infrastructure (instead of within the application) allows the communication infrastructure to dynamically enable/disable the checks (this can be for example required when the overhead caused by the assertion checking risks to compromise performance requirements), and to integrate the checks with other advanced repairing strategies that can be built into the communication infrastructure itself. On the other hand, checking the assertions within the communication infrastructure can introduce new vulnerabilities, i.e., new opportunities to attack the communication infrastructure. How to tackle these emerging security issues and whether it is beneficial shifting these vulnerabilities from the application to the communication infrastructure, are still open problems and will be the subject of further investigations.

Assertion adaptation: When assertions are violated, ARV records the sequences of events that led to the violations and the status of the application after the violation. Violations may indicate illegal interactions or lack of testing, i.e., legal behaviors that have been not observed during testing. To distinguish between these two alternatives, we need diagnosis mechanisms that inspect the detected violations, identify either illegal interactions or lack of testing, and trigger the appropriate self-management mechanisms. Often, illegal interactions generate a set of assertion violations. Diagnosis mechanisms can attempt to correlate sets of violations to better identify the cases of illegal interaction.

For example, the violation of an invariant followed by an error message from the application can be safely diagnosed as an illegal interaction.

Diagnosis mechanisms may include two steps: an automatic attempt to be exploited first, and an operator driven attempt to complement the automatic mechanisms, when they cannot completely cope with the problem. Currently ARV relies on the operator driven approach, while we are investigating the feasibility of automatic diagnosis. ARV provides operators with information about the assertion violations to select an adequate repairing strategy. Our experience with BCT indicates that for adequately tested systems the information about assertion violations can be handled by human operators after some preprocessing. Moreover, the BCT information can be enhanced with additional data that indicate if the assertions have been violated while executing previously tested or new protocol sequences.

When the diagnosis mechanisms identify illegal interactions, they trigger proper self-management mechanisms to overcome the failure and/or repair the fault. In the case of lack of testing, self-management mechanisms may allow to run additional test cases to check for the correct interaction of the application with the new protocol. In this latter case, BCT can be instructed to update assertions to include the information of the newly tested interactions.

The possibility of automatically generating assertions that capture the set of tested interaction protocols and to automatically update the set of assertions while new interaction protocols are revealed and tested is particularly useful in the considered context, where the impossibility of controlling and predicting the evolution of the execution environment reduces the efficacy of classic testing techniques that focus on pre-deployment only.

4 Related work

ARV monitors the internal and external system behavior, records low-level data in term of input/output values and entities interactions, and uses such data to identify both untested and wrong interaction protocols. Several researchers proposed mechanisms for low-level monitoring of network applications in different contexts, e.g., Schmerl and Garlan [5], Schilit, Adams and Want [20], Hong and Landay [9]. This work focuses mostly on runtime information about performances and architectural modeling, while ARV focuses on automatic monitoring interaction protocols.

Diagnosing misbehaviors requires the ability of interpreting the monitored data. So far, research has focused on the use of embedded assertions [19, 16, 1, 17] to identify violations of formal constraints, the runtime check of event patterns [7, 8, 21] to validate temporal properties, and the replication of program blocks to compare and confirm the validity of the system outputs [10, 2]. We propose a novel approach based on the run time check and adaptation of automatically generated assertions that describe the set of successfully tested interactions.

Mechanisms for self-repairing runtime systems have proposed by some researchers. For example, Horning, Lauer, Melliar-Smith and Randell provide an interesting approach based on rollback and resume strategies [10]. Rollback and resume strategies are well investigated also in database research [6], to deal with problems such as concurrent accesses and incomplete transactions that are well-know sources of non-determinism. These approaches well complement our proposal that so far focused mostly on identifying faults, leaving the problem of runtime self-repairing in the future agenda.

Our approach integrates the DAIKON tool to capture and synthesize the behavior of application entities. DAIKON has been well used to infer likely invariants of software components at runtime [4, 22, 12, 18], but existing approaches focus mostly on analysis of interactions of known components, that works well in the case of component-based software, but not in the case of unpredictably changing environments as the ones considered in our work.

5 Conclusions and future work

The Internet infrastructure and its expected evolution offer the opportunity to develop ubiquitous and heterogeneous applications that will increase productivity and flexibility in many application domains, ranging from domestic systems to automotive, transportation, and medical control applications. These systems leverage technology from several domains, such as service oriented architectures, peer to peer systems, distributed middleware and wireless networks. In this scenario, many failures can likely stem from unexpected concurrent interaction patterns between independently developed and maintained applications, unpredictable stress and interferences of the environment, or unexpected malicious interactions.

In this position paper, we propose a new approach based on the automatic synthesis of assertions during testing of an application. The synthesized assertions capture the semantics of the observed application behaviors, and can be integrated in the networking infrastructure to detect violating interactions at runtime and trigger self-management mechanisms correspondingly. This new mechanism can provide a strong basis for the development of autonomic applications able to survive unpredictable changes and evolutions of the network environment.

Currently we are investigating the feasibility of automatic diagnosis to distinguish between violations that indicate illegal interactions and violations that indicate lack of testing. Our research agenda includes the integration of the mechanism briefly described in this paper with code transformation mechanisms for diagnosis purposes. Our experience with semantic mapping across notations suggests that graph transformations provide useful support for relating code to behaviors, thus supporting advanced diagnosis mechanism [3]. We are also working to a set of case studies to better understand potentialities and limitations of the ARV approach.

References

[1] P. Abercrombie and M. Karaorman. jContractor: Bytecode instrumentation techniques for implementing design by contract in java. In K. Havelund and G. Rosu, editors, *Electronic Notes in Theoretical Computer Science*, volume 70. Elsevier, 2002.

[2] A. Avizienis and L. Chen. On the implementation of n-version programming for software fault tolerance during execution. In *Proceedings of IEEE COMPSAC 77*, pages 149–155, Nov. 1977.

[3] L. Baresi and M. Pezzè. A toolbox for automating visual software engineering. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering*, volume 2306 of *Lectures Notes in Computer Science*, pages 189–202. Springer Verlag, 2002.

[4] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, Feb. 2001.

[5] D. Garlan, B. Schmerl, and J. Chang. Using gauges for architecture-based monitoring and adaptation. In *Proceeding of the Working Conference on Complex and Dynamic Systems Architecture*, Dec. 2001.

[6] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computer Surveys*, 15(4):287–317, 1983.

[7] K. Havelund and G. Rosu, editors. *First Workshop on Runtime Verification (RV'2001)*, volume 55(2) of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2001.

[8] K. Havelund and G. Rosu, editors. *Second Workshop on Runtime Verification (RV'2002)*, volume 70(4) of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.

[9] J. Hong and J. Landay. An infrastructure approach to context-aware computing. *Human-Computer Interaction*, 16(2/4):287–303, 2001.

[10] J. Horning, H. Lauer, P. Melliar-Smith, and B. Randell. A program structure for error detection and recovery. In *Lecture Notes in Computer Science*, volume 16, pages 177–193, 1974.

[11] J. Kephart and D. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, January 2003.

[12] L. Lin and M. D. Ernst. Improving adaptability via program steering. In *Proceedings of the 2004 International Symposium on Software Testing and Analysis*, pages 206–216, July 2004.

[13] L. Mariani. *Behavior Capture and Test: Dynamic Analysis of Component-based Systems*. Phd thesis, Università degli Studi di Milano Bicocca, 2004.

[14] L. Mariani. Capturing and synthesizing the behavior of component-based systems. Technical Report LTA:2004:01, DISCO, University of Milano Bicocca. LTA lab., Feb. 2004.

[15] L. Mariani and M. Pezzè. Behavior capture and test: Automated analysis of component integration. In *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)*, 2005 (to appear).

[16] B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.

[17] O. Raz, P. Koopman, and M. Shaw. Enabling automatic adaptation in systems with under-specified elements. In *Proceedings of the first workshop on Self-healing systems*, pages 55–60. ACM Press, 2002.

[18] O. Raz, P. Koopman, and M. Shaw. Semantic anomaly detection in online data sources. In *Proceedings of the 24th International Conference on Software Engineering*, pages 302–312, May 2002.

[19] D. Rosenblum. A practical approach to programming with assertions. In *IEEE Transactions on Software Engineering*, volume 21, pages 19–31, Jan. 1995.

[20] B. Schilit, N. Adams, and R. Want. Context-aware computing applications. In *Proceedings of the Workshop on Mobile Computing Systems and Applications*, pages 85–90, Dec. 1994.

[21] O. Sokolsky and M. Viswanathan, editors. *Third Workshop on Runtime Verification (RV'2003)*, volume 89(2) of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2003.

[22] T. N. Win, M. D. Ernst, S. J. Garland, D. Kirli, and N. Lynch. Using simulated execution in verifying distributed algorithms. *Software Tools for Technology Transfer*, 6(1):67–76, July 2004.