Behavior Capture and Test for Verifying Evolving Component-Based Systems

Leonardo Mariani
Università of Milano Bicocca
Dipartimento di Informatica, Sistemistica e Comunicazione
20126 Milano, Italy
mariani@disco.unimib.it

Research Area: Run-time verification of evolving component-based systems

1. Verification of Evolving Component-Based Systems

Component-Based System (CBS) technology supports rapid development of complex heterogeneous evolving systems by enhancing reuse and adaptability. CBSs can be extended and adapted by modifying one or more components. The same component can be used in several systems, and the same system can be deployed in many configurations that differ for some components. Traditional test and analysis techniques make little use of quality information about components and subsystems when testing the whole system. Thus, reusability for quality assessment and reduction of quality related costs are not fully exploited.

Moreover, verification of CBSs is hardened by the frequent lack of information about components that are provided by third parties without source code and with incomplete documentation. This framework reduces the applicability of many traditional testing and analysis techniques for CBSs. Main goal of my PhD research is the definition and experimentation of testing and analysis techniques that allow to efficiently test CBSs in presence of limited information about design and code by reusing behavioral information that can be gathered from previous usage of the components.

2. Behavior Capture and Test

My PhD research aims at using run-time information to automatically derive test cases for components when used in different contexts. I assume that the information recorded on the behavior of a component C in a specific system S can be used both for testing a replacing component, i.e., a new component that replaces C in S, and for testing component C when used in a new system. Our technique, hereafter called Behavior Capture and Test (BCT), first synthesizes component behavior by observing executions and inferring invariants, and then monitors executions of components to identify possible faults. We use the recorded behaviors also to create test suites and oracles to test components in different systems. The approach is based on five main steps: (1) *Deploy time*: Automatic generation and installation of recorders; (2) *Run-time*: Recording of executions (single behaviors); (3) *Run-time*: Distilling I/O and interaction invariants; (4) *Run-time*: Filtering single behaviors; (5) *Regression*: Automatic verification and testing.

Step 1: Generation of the Recorders We generate two kinds of recorders for a component: *Stimuli* and *Interaction Recorders*. Stimuli recorders are used to record requests and corresponding results, while interaction recorders are used to record patterns of interactions with other components. The behavior of the recorders is largely independent from the target components and they can be automatically generated from an interface or an interface specification. Installing recorders require altering the binding mechanism of the system. So far, we have manually developed several recorders. We are studying techniques for automatic generation and installation of recorders. In particular, we are investigating the possible degree of automatization for different technologies, e.g., Enterprise Java Beans.

Step 2: Registration of Single Behaviors Whenever a component issues a request, the I/O recorder stores parameters and results. Primitive data are recorded directly. Complex objects are processed before being recorded. Processing consists of extracting state information by recursive invocation of inspectors up to a given depth. Inspectors are methods of the class that return state information without altering the state. Inspectors can be manually identified by the users or automatically inferred by matching methods' signature with syntactic rules. For example, a method whose template signature is `<anyType> get<anyName> (void)` is assumed to be an inspector. We are currently refining the technique by adding validation

of the heuristically selection of inspectors.

Recording introduces computational overhead. To face this drawback we are investigating *recording policies* that limit the number of recorded single behaviors. We are currently studying optimal tradeoffs between the amount of recorded information and computational costs.

Step 3: Distilling Invariants We are distilling two kinds of invariants, I/O and interaction invariants, which summarize the behavior of the component. I/O invariants describe the behavior of a specific service and are computed with an extension of Daikon [4]; our extension allows considering complex objects as well as simple objects. Interaction invariants are computed by inferring a regular expression that resumes all observed pattern of interactions. The elements of the alphabet are `C.S()`, where `C` is the name of a component and `S` is the name of a service implemented by the component. The semantic of this element denotes the invocation of the service `S()` implemented by the component `C`. The invariant is inferred by merging observed behaviors and by applying *generalization rules*. Generalization rules modify the regular expression so that the new regular expression generates a language subsuming the older one. Example of generalization rules are: "`aaa` is mapped to `a*`" where `a` is an element of the alphabet and "`AAA` is mapped to `A*`" where `A` is a regular expression. The definition of a comprehensive set of rules is one of the goals of my PhD thesis.

In many work, behavioral information is synthesized from scenario specifications [11, 1]. A scenario specification describes all stimuli generated in the system during a specific execution. The synthesis of several scenario specifications produces a LTS describing the behavior of the whole system. In our approach, we incrementally refine the regular expression describing the behavior of each service by observing only local interactions. Information synthesized in the two approaches is very similar, but it differs on both the granularity and accuracy. Scenario synthesis describes the system- and component-level behavior, while interaction invariants describe the service-level behavior. Moreover, LTS may contain some undesired behaviors due to the synthesis process, e.g., implied scenarios [7]. In our case, we explicitly choice to introduce service-level generalization rules to both lessen the probability to exclude an admissible behavior and to reduce the size of the regular expressions. Next, we intend to compare service-level generalization rules with scenario synthesis to exploit benefits and drawbacks of both approaches.

Step 4: Filtering Single Behaviors Storing all single behaviors requires an enormous amount of memory, on the other hand single behaviors are very useful when used as test cases, and hence we are studying different *selection policies* for selecting relevant single behaviors, based on different coverage criteria, impact of the single behavior on inferred invariants and tester's interests.

Step 5: Automatic verification and testing We identified three very general updates that can be performed in a system and that enable the application of the BCT technique: (1) a component in use in a system is added to a different system; (2) an existing component is replaced with a new component; (3) an existing component is replaced with a component in use in a different system.

In the first case, it is possible to take advantage of information gathered from executing the component as part of a different system; in the second case it is possible to take advantage of information gathered from executing the replaced component as part of the target system; finally, in the third case, it is possible to take advantage of both kinds of information.

Single behaviors are used to generate both test suites and oracles. A test suite for component `C` can be generated by assembling `C`'s single behaviors, while a test suite for component `C'` (where `C'` is the component replacing `C`) can be generated by selecting `C`'s single behaviors that are still executable (some updates on `C` may have turned to unexecutable some single behaviors). We intend also to generate two kinds of oracles: strong oracles and weak oracles. Strong oracles require that the behavior observed while executing the corresponding test suite matches the behavior observed while executing the old version of the component. Weak oracles require that invariants are not violated while executing the test suite.

Both I/O and interaction invariants are used to monitor the executions of the target component. When an invariant is violated a warning is generated and interpreted by the user. We are still investigating all possible use of warnings; some examples are faults detection, derivation of user operational profiles and discovering of functionality never or rarely used.

When updating the system by replacing an existing component with another existing component, it is possible to formally match both interaction and I/O invariants. We are studying possible analysis based on invariants, preliminary ideas concern conflicts detection, derivation of component operational profiles and feedback for testing.

3. Related Approaches

Although testing and verification of CBSs are addressed by many authors, only few make use of in-field data. Existing testing techniques for CBSs do not take into account the specific way a system is used because testing is performed from the component assembler before the system is deployed. To overcome this inconvenience, the Perpetual Testing approach proposes to shift to the user-environments part of the test obligations that are not completely fulfilled

during the testing phase [9]. Our approach exploits the main principles underlying Perpetual Testing in the case of CBSs. Many run-time verification techniques for CBSs, e.g., [3, 2, 5], are based on some specifications. These approaches are not always applicable to CBS where specifications are not often available.

Existing approaches to verification and testing using in-field data require the instrumentation of the source code. BCT wraps components, thus requires no code alteration. Wrapping is harder to obtain with respect to code instrumentation, but it has two main benefits: it is applicable to components that are not been developed to support a particular kind of analysis and leaves the system designers free to install wrappers enabling the analysis they prefer.

For example, McCamant and Ernst infer pre- and post-conditions of the services implemented in a component by monitoring executions [6]. Formal matching of pre- and post-conditions can be used for predicting possible faults that can derive from updating components. If the source code is not available, the approach is applicable only to components that exchange scalar and structured data types, and considers only the interactions of the system with the component and not the interactions of the component with the system. On the contrary, the existing implementation of BCT computes invariants for complex objects and takes advantage of the object's internal structure to perform very accurate inference. Moreover, BCT invariants will be used for both generating monitors and matching synthesized behaviors, thus widening the spectrum of applicability.

Orso et al. gather field data both to predict impact of updates and to drive the regression testing [8] (regression test selection, test suite prioritization and test suite augmentation). Both BCT and Orso et al. approaches address management of side effects, but they focus on different techniques: BCT uses run-time verification, while Orso et al. use testing. For the moment, both test suite prioritization and selection are neglected in BCT, while automatic test suite augmentation (that is not possible in the Orso et al. approach) is one of the functionality foreseen for BCT.

Rodriguez monitors user activities to infer the way the system is used [10]. Rodriguez's approach has been used for navigability testing of web sites, but the idea can be extended to CBSs too. In BCT, both I/O and interaction invariants specify how a component is used but it could be difficult to rebuild the user operational profile from information about the usage of single components. Moreover, BCT neglects non-functional properties such as usability. In future activities we intend to further investigate both derivation of operational profiles and non-functional properties.

4. Conclusions and Future Work

This paper describes the main elements of BCT, the first attempt to combine in-field data extraction, run-time verification, invariant detection and testing in a unique framework for verifying CBSs. The technique is based on the derivation of invariants from information on in-field behavior of components. Such invariants are used to automatically derive test cases and monitors for new systems that include such components. The early experimental results demonstrate the feasibility of the approach. We are currently developing prototypes to verify the extent of automatization of the technique and to collect experimental data to verify our hypothesis and identify weakness of the approach. Our goal is to fully refine the technique, identify limits and advantages, and define the field of applicability.

References

[1] R. Alur, K. Etessami, and M. Yannakakis. Inference of Message Sequence Charts. In *proceedings of 22nd International Conference on Software Engineering*, pages 304–313, 2000.

[2] M. Barnett and W. Schulte. Spying on components: A runtime verification technique. In *proc. of Work. on Specification and Verification of Component-Based Systems*, 2001.

[3] S. H. Edwards. A framework for practical, automated black-box testing of component-based software. *Journal of Software Testing, Verification and Reliability*, 11(2), 2001.

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

[5] K. Havelund and G. Rosu. Synthesizing monitors for safety properties. In *proceedings of the Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *LNCS*, pages 324–356. Springer Verlag, 2002.

[6] S. McCamant and M. D. Ernst. Predicting problems caused by component upgrades. In *proc. of 9th European software engineering conf. and 10th ACM SIGSOFT int. symp. on Foundations of software engineering*, pages 287–296, 2003.

[7] H. Muccini. Detecting implied scenarios analyzing non-local branching choices. In *proc. of Int. Conference on Fundamental Approaches to Software Engineering*, 2003.

[8] A. Orso, T. Apiwattanapong, and M. J. Harrold. Leveraging field data for impact analysis and regression testing. In *proc. of the 9th European software engineering conf. and 10th ACM SIGSOFT international symp. on Foundations of software engineering*, pages 128–137, 2003.

[9] C. Pavlopoulou and M. Young. Residual test coverage monitoring. In *proceedings of the 21th International Conference on Software Engineering (ICSE'99)*, pages 277–284, 1999.

[10] M. G. Rodriguez. Automatic data-gathering agents for remote navigability testing. *IEEE Software*, 19(6):78–85, November/December 2002.

[11] S. Uchitel, J. Kramer, and J. Magee. Synthesis of behavioral models from scenarios. *IEEE Transactions on Software Engineerings*, 29(2):99–115, February 2003.