Behavior Capture and Test for Controlling the Quality of Component-Based Integrated Systems

Leonardo Mariani
Dipartimento di Informatica, Sistemistica e Comunicazione
Università degli Studi di Milano Bicocca
via Bicocca degli Arcimboldi, 8
20126 Milano, Italy
mariani@disco.unimib.it

Mauro Pezzè
Dipartimento di Informatica, Sistemistica e Comunicazione
Università degli Studi di Milano Bicocca
via Bicocca degli Arcimboldi, 8
20126 Milano, Italy
pezze@disco.unimib.it

1. INTRODUCTION

Complex software systems are seldom designed from scratch; rather they are often designed by assembling basic components. Software tools are not an exception: They are often obtained by assembling simple tools that provide basic functionality. From the verification viewpoint, basic tools do not differ from components: in both cases they can be seen as black box elements that provide a set of services for the embedding system or tool. Complex systems and tools are often provided in many different versions and configurations that are obtained by adding and replacing various components. Let us consider for example a complex CASE tool that runs on several platforms and is distributed in various versions for different user profiles. Many graphics and system components may be used for distinct platforms. For instance, the simple configurations distributed free-of-charge may include only a simple set of basic tools, while the educational and the professional editions may include a wider set of basic tools that provide extended functionality. Unfortunately, behavioral differences among components may cause subtile failures difficult to reveal and remove. On of the main goals of verification is to check the completeness and compatibility of the services provided by the components to reveal possible conflicts, thus supporting efficient verification of different tool configurations and versions.

The use of components in running systems produces a lot of useful information about the components' behavior that could be used to check for the compatibility between different components and between components and embedding systems. Unfortunately this information is normally lost. This paper proposes a new technique, called *Behavior Capture and Test (BCT)*, that takes advantage of this information. We first monitor the system execution and capture the essential characteristics of the behavior by means of automatic probes. We then use the collected information to verify the compatibility of the components when used as part of other systems. In this way, we can automatically reveal incongruences of components that either replace existing ones or are added to existing systems to extend their functionality.

The technique proposed in this paper is based on the construction of behavioral invariants that represent the interaction of the component with the system. A replacing component must satisfy the same invariants of the replaced component, while a new component must be compatible with the invariants that characterize the embedding system.

2. THE BCT APPROACH

Basic components can be added and/or replaced in four main ways that are summarized in Figure 1:

- a new component `B` replaces component `A` in system `S` (Figure 1a)
- a component `B` in use in system `S'` replaces component `A` in system `S` (Figure 1b)
- a component `A` in use in system `S` is added to system `S'` (Figure 1c)
- a new component `B` is added to system `S` (Figure 1d)

In this paper, we consider the first three ways, since they allow to gather information about the behavior of the components involved in the modifications by examining the behavior of the running systems.

BCT is based on three main steps:

- capture *single behaviors* of the target component. Single behaviors represent the interplay of the component with the embedding system,
- infer *behavioral invariants* of target component from single behaviors. Behavioral invariants summarize classes of homogeneous behaviors,
- use both invariants and selected single behaviors for checking the compatibility among components and between components and system.

The first step of BCT consists in capturing single behaviors. Single behaviors describe interplays of the target component with the system and are characterized by *requests*,

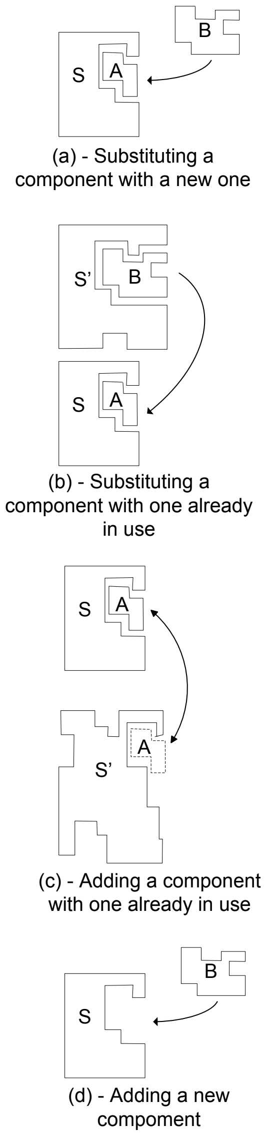
S
A
B
(a) - Substituting a component with a new one
S'
B
S
A
(b) - Substituting a component with one already in use
S
A
A
S'
(c) - Adding a component with one already in use
B
S
(d) - Adding a new compoment

Figure 1: Evolving systems through component addition or substitution

results, and *interactions*. The requests are the services requested by the system to the target component; the results are the values returned by the target component to the system; the interactions are the services requested by the target component to other components to complete the requested services.

The second step of BCT consists of summarizing component behaviors by computing behavioral invariants that describe the observed behaviors of the target component. Behavioral invariants include *interaction invariants* that describe interaction sequences, and *I/O invariants* that define relations between requests and results.

The third and last step of BCT consists of checking the validity of invariants on newly added components and executing selected single behaviors to verify the compatibility of the new components with the old ones.

When updating a system `S` by substituting a component `A` with a new component `B` (Figure 1a), we check for the compatibility of `B` with `S` by checking for the validity of both the interaction and I/O invariants of component `A` when `S` is executed with `B`. Invariants can be verified both during regression testing by replaying the single behaviors recorded for `A` on `B` and at run-time.

When component `B` is associated with a set of interaction invariants, e.g., derived from the use of `B` in a different context (Figure 1b), we can additionally check for the compatibility of the invariants of `A` and `B` by matching the languages generated from the regular expressions of `A` and `B`, that code the interaction invariants.

When the functionality of a system `S` is augmented adding a component `B` (Figure 1c), we can verify the compatibility of the interaction and I/O invariants of `B` with the specifications of interaction patterns for `S`.

3. CAPTURING SINGLE BEHAVIORS

Behavioral information can be automatically inferred by monitoring execution during test and normal operation of software. Figure 2 shows a conceptual framework for automatically capturing behaviors and computing behavioral invariants. A component is wrapped with a **`stimuli recorder`** that intercepts and stores requests and results, and with **`interaction recorders`** that intercept and store interactions with other components. Stimuli and interaction recorders collect all single behaviors.

Capturing single behaviors presents three main problems related to the observability of the requests, and the complexity and size of the exchanged parameters. The use of complex interaction mechanisms, e.g., dynamic binding or communications through system calls, may limit monitoring and recording capability. Capturing simple data is relatively simple, but capturing structured data, e.g., complex objects, may be difficult. In presence of complex data, stimuli recorders select “representative values”, i.e., values that can be extracted with non-intrusive methods, i.e., methods that do not alter the state of the class. When the objects are hierarchically composed of other objects, we recursively consider non-intrusive methods up to a given depth. Recording a single interaction requires a limited amount of resources, but indiscriminately recording all possible interactions quickly leads to request of resources that exceed any reasonable setting. The **`invariant distiller`** selects a subset of single behaviors and distills behavioral invariants.

Single behaviors can be selected with many criteria. So far we investigated the following approaches:

- coverage criteria: we select behaviors that augment some specified functional or structural coverage
- invariant impact: we select behaviors that modify the set of working invariants
- architecture constraints: we select only behaviors that are either generated by requests from a given set of

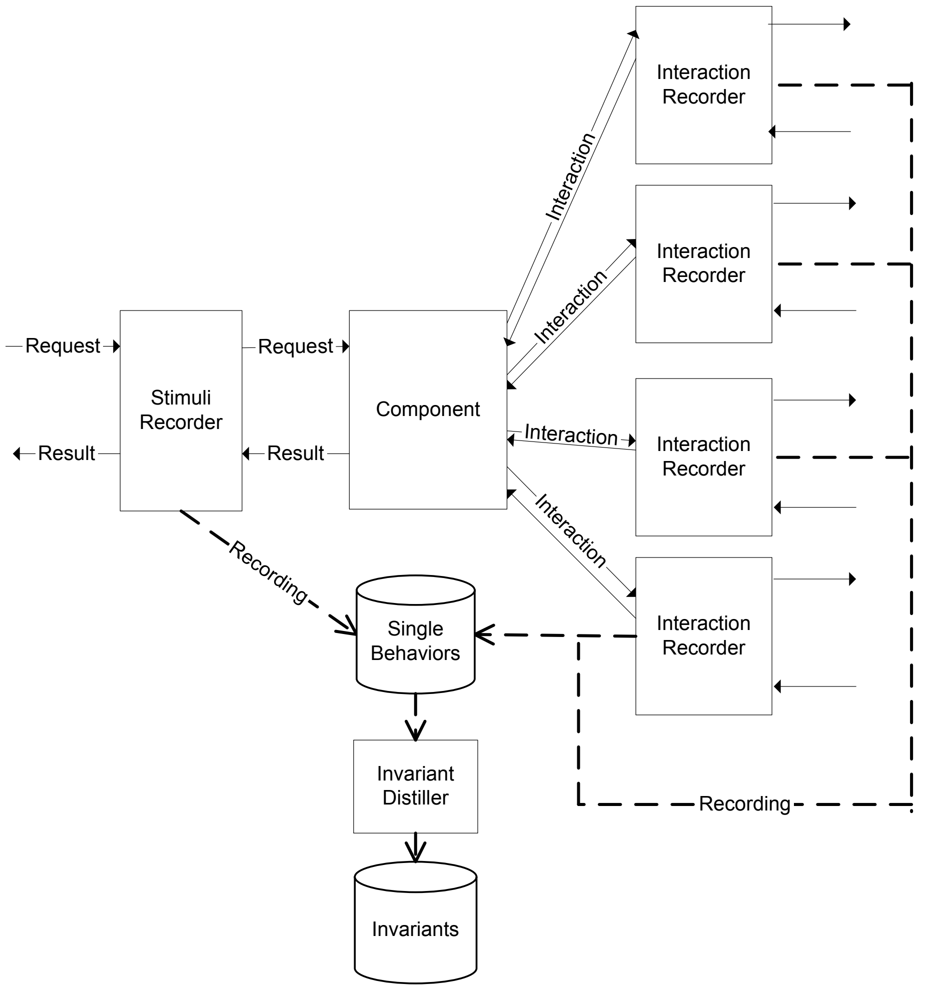

Figure 2: The BCT conceptual architecture

components or include interactions with a given set of components

- user preference: we select behaviors based on ad hoc criteria identified by the user, e.g., within a temporal window, if the user can identify a testing session particularly significant

The `stimuli recorder` can be automatically generated by using the monitored component's interface. Services published in the monitored component's interface are copied to the `stimuli recorder`'s interface; the behavior of each service consists of recording the request and then forwarding it to the monitored component.

The `interaction recorders` are generated using the interface of the wrapped components. The behavior of each service consists of recording the interaction, and then forwarding it to the corresponding component.

When an `interaction recorder` forwards a request to another monitored component, the request will be received and processed by the `stimuli recorder` of the receiving component. In this case, the interaction involves `interaction` and `stimuli recorders` that can exchange additional information to facilitate analysis. For example, the request of the service $\texttt{s(par}_1\texttt{,...par}_n\texttt{)}$ from the monitored component `A` to a monitored component `B` can result in the following sequence of events:

1. component `A` sends the request $\texttt{s(par}_1\texttt{,...par}_n\texttt{)}$ to component `B`.
2. the request is intercepted from `A`'s `interaction recorder`
3. the `interaction recorder` of `A` records the interaction and adds its identity to the request forwarded to `B`: $\texttt{s(par}_1\texttt{,...par}_n\texttt{, 'A')}$
4. the `stimuli recorder` of `B` receives the request and records the request together with the identity of the caller
5. the `stimuli recorder` of `B` forwards the request $\texttt{s(par}_1\texttt{,...par}_n\texttt{)}$ to `B`
6. component `B` receives the request $\texttt{s(par}_1\texttt{,...par}_n\texttt{)}$

In this way, the technique improves the recording and distilling phases by enabling additional information to travel among components.

The insertion of `stimuli recorders` and `interaction recorders` between components depends on the binding mechanism among the original components. For example, if the topology of a component-based system is defined by configuring XML files, we can include stimuli and interaction recorders by modifying the XML files; if components connect by accessing at run-time a discovering service, we need to modify the publishing and discovery procedures to include components for monitoring into the system.

4. COMPUTING BEHAVIORAL INVARIANTS

Recording all single behaviors of a component is impractical due to the large size of the information to be stored. The components' behaviors can be effectively stored by distilling behavioral invariants, i.e., interaction and I/O invariants.

Interaction invariants specify the set of interactions between the component and the system for the different stimuli. Interaction invariants can be described with regular expressions. The alphabet used to build the regular expressions is composed of elements of the form:

component_name.name_of_the_service

that indicate the request of the service *name_of_the_service* provided by component *component_name*. For example, let us consider a component *Login* that manages the login of an application. The component accesses both a component database to verify username and password, and a component user-profile to setup the profiles. If the username or the password do not satisfy standard constraints (e.g., the password is shorter than a given length), the *Login* component does not access the database component; if there is no profile associated to the user, it does not access the profile component. This interaction pattern can be described with the following interaction invariant:

$(UserDB.check + \epsilon)Profiles.setProperty^*$

Interaction invariants provide preliminary information about incompatible versions, admissible and unfeasible configurations, and conflicting components. When substituting or adding a component, we can perform a first set of compatibility checks by checking if the interaction invariants are satisfied.

Let us consider for example a Car Navigation System (CNS), i.e., a system that guides drivers along their travel providing information about current position, destination, distance, and so on. CNSs are built with several interacting components, each one available in several versions that differ in complexity, functionality and cost. CNSs are provided in several configurations for different car manufacturers and models. Different configurations are obtained by suitably replacing and adding components [9].

CNS includes a memory manager subsystem that retrieves maps by relying on the services of the DBEngine, Volatile memory, Non-volatile memory and the Data Drive components. We first computed the following regular expression for versions *Va* and *Vb* of the DBEngine component used in configurations $V.CNS.a$ and $V.CNS.b$ of the CNS, respectively: '

Va in $V.CNS.a$: *(VolMem.map)(DataDrive.loadMap+*ϵ*)*

Vb in $V.CNS.b$: *DataDrive.map*

The expressions indicate that version *Vb* of the DBEngine component always accesses only the Data Drive component, while version *Va* accesses both the Data Drive and the Volatile Memory components. The interaction invariants indicate that the two components behave in different way, to provide the same service.

If we want to update CNS *V.CNS.b* by substituting the DBEngine component *Vb* with *Va*, the simple comparison of the interaction invariants of the two components indicates that the system may fail or provide degraded services if not equipped with a suitable volatile memory. The interaction invariant indicates also that the Data Drive component must provide a *loadmap* service; a particular implementation of a Data Driver that does not provide such service would not be suited in the context of this system.

The interaction invariants capture only the interactions of the components with the system, but they do not record the behavior of the components themselves. The components' behavior can be described with I/O invariants, i.e., relations among input and output values.

Ernst et al. [4] proposed invariants for primitive types. We extended their use for complex data types in this new context by applying the invariants on the values extracted from the objects with non-intrusive methods, as described in the previous section. The computed invariants may not cover all possible behaviors, but the goal of this work is not to extract a complete specification from the code, but rather to automatically get information that can increase automatic testing capabilities.

I/O invariants are computed starting from an initial set of generic invariants that are selected according to the type of data exchanged in the service requests.

Reduction techniques over inferred invariants are specified in the invariant distiller, aiming at reducing the (initial) set of invariants inferred from single behaviors. A first general way to reduce invariants is to prefer used-defined and service-specific invariants to general invariants. This approach is based on the assumptions that user-defined invariants are more important than other invariants and that service-specific invariants are more accurate than general-purpose invariants.

The inference procedure for calculating the interaction invariants works by re-computing the shorter regular expression complying with all executions after each new execution. The algorithm monitors the set of interactions generated during the computation of a service s; if the expression corresponding to the generated interactions belongs to the language generated by the actual regular expression for s we ignore it, since it is already captured by the current set of invariants. Otherwise, the language is extended to include the new sentence.

I/O invariants and regular expressions depend from the use of the component. The recorded single behaviors provide information about the subset of the input domain that is actually used in the monitored context. When we match the behavior of two components used in different contexts or when we use invariants inferred for a component `C` in a system `S` for a new system `S'`, we can obtain abnormal results. If we are matching two components (see Figure 1b), the analysis can be "normalized" by replaying the single behaviors of the first component on the second one and vice versa. The final set of invariants computed for the two components will be obtained from the same set of requests. In the case of invariants obtained by executing a component `C` in system `S` and reused for system `S'`, violation of the invariants are reported to the developer as different ways to use the same component.

BCT can be generalized to (sub)systems, i.e., component aggregates. A first approach consists in applying the same analysis to stimuli entering and exiting the system rather then the single component. A finer approach consists in iteratively processing the set of generated interactions, i.e., if the service `s` of the component `A` generates the interaction $B.m1 * C.m1$, and the service $B.m1$ of the component `B` generates the interaction $(C.m1)D.m2$ the subsystem composed by `A` and `B` generates $((C.m1)D.m2) * C.m1$ when `s` is invoked. This kind of analysis enables the detection of inter-component faults, such as callback, re-entrance or distributed recursion [6].

5. RELATED WORK

The problem of test and analysis of complex component based systems has been addressed by many research groups ([1] and [10]). This paper presents a new technique that extends approaches proposed in different domains for obtaining an automatic technique for gathering information about component and subsystem behavior to check for compatibility among components integrated in different systems. The proposed technique is inspired by the work on Perpetual testing by Pavlopoulou and Young [8]. The computation of I/O invariants extends a technique proposed by Ernst et al. for simple data types [4].

Many researchers proposed techniques for monitoring executions at test at run-time. Martins et al.'s self-test components [7] can automatically test their services when deployed in a new system. The technique extends Binder's Object Oriented Design-for-Testability [2], where classes are augmented with tests, driver specifications, and oracle specifications. The techniques do not address effectively COTS components, because they require instrumentation of components, i.e., they require access to source code, which is seldom possible with COTS. Moreover, the component specifications required for deriving drivers and oracles must be provided by the component developer, which cannot be required for components developed by independent parties. Finally, self-test components are effective during integration testing, but are not particularly suitable for evolving systems and regression testing, because it is difficult to reuse drivers and oracles.

The Retrocomponents proposed by Liu and Richardson [5] record information about unit testing. This information must be associated to the component, and is used during integration testing. Retrocomponents do not infer the behavior of the component and are essentially tailored to support the tester during integration test.

Edwards' BIT wrappers [3] provide an abstract safe representation of the component internals that can be used to check for the validity of pre-conditions, post-conditions and invariants. Edwards proposes a semi-automatic generation of BIT wrappers, but the approach is not applicable to COTS components or in general to component that have not a specification. BIT wrappers effectively detect wrong component implementation, integration faults or violations on the interface, but do not address software evolution.

6. CONCLUSIONS

This paper proposes a new method to automatically generate tests for single components and for their integration in different systems. The generated framework facilitates the testing of systems and tools that are produced in many versions and configurations by substituting or adding one or more components.

We are currently experimenting with the technique on different case studies. The on going experiments are producing encouraging results: the data on the component behavior that can be gathered with the technique proposed in this paper allows for easily identifying anomalous behaviors due to incompatibility of different components, and help in removing subtile faults that can lead to unpredictable failures.

7. REFERENCES

[1] A. Bertolino and D. Richardson, editors. *Proceedings of International Workshop on the Role Of Software Architecture in Testing and Analysis of Complex Systems(ROSATEA)*, Marsala, Sicily, June/July 1998.

[2] R. Binder. Design for testability in object-oriented systems. *Communications of the ACM*, 37(9):87–101, 1994.

[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), June 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] C. Liu and D. Richardson. Software components with retrospectors. In *Proceedings of the International Workshop on the Role of Software Architecture in Testing and Analysis (ROSATEA)*, 1998.

[6] L. Mariani. A fault taxonomy for component-based software. In M. Pezzè, editor, *Proceedings of*

International Workshop on Test and Analysis of Component-Based Systems (TACoS), in conjunction with European Joint Conferences on Theory and Practice of Software (ETAPS), volume 82 of *Electronic Notes in Theoretical Computer Science (ENTCS)*. Elsevier Science, 2003.

[7] E. Martins, C. Toyota, and R. Yanagawa. Constructing self-testable software components. In *Proceedings of the 2001 International Conference on Dependable Systems and Networks (DSN '01)*, pages 151–160. IEEE, 2001.

[8] C. Pavlopoulou and M. Young. Residual test coverage monitoring. In *Proceedings of the 21th International Conference on Software Engineering (ICSE'99)*, pages 277–284, New York, 1999. IEEE Computer Society Press / ACM Press.

[9] A. Peyracchia. From unified modeling language to end-of-production functional test: an industrial approach. Master's thesis, Graduate College of the University of Illinois at Chicago, Chicago, Illinois, 2002.

[10] M. Pezzè, editor. *Proceedings of International Workshop on Test and Analysis of Component-based Systems (TACoS), in conjunction with European Joint Conferences on Theory and Practice of Software (ETAPS)*, volume 82 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, Warsaw, Poland, April 2003. Elsevier Science.