Behavior Capture and Test: Automated Analysis of Component Integration

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

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

Abstract

Component-based technology is increasingly adopted to speed up the development of complex software through component reuse. Unfortunately, the lack of complete information about reused components, and the complex interaction patterns among components can lead to subtle problems that throw new verification challenges. Good components are often re-used many times, sometimes within product lines, in other cases across different products. The reuse of components provides a lot of information that could be useful for verification. In this paper, we show how to automatically analyze component interactions by collecting information about components' behavior during testing and field execution, and then using the collected information for checking the compatibility of components when updated or reused in new products. The paper illustrates the main problems in developing the idea, proposes original solutions, and presents a preliminary experience that illustrates the effectiveness of the approach.

1 Introduction

Software components are commonly developed independently, often by third party vendors, without full knowledge of all their possible uses, and they may be available for reuse without the source code, and with only (partial) information about their interfaces, the offered services and their environmental requirements [6].

Reuse promotes the development of high-quality components. Unfortunately, many subtle problems do not depend on faults in single components, but derive from component integration, which may result in unexpected interaction patterns, undesired behaviors and unwanted side effects. The limited availability of information about components complicates verification of Component-Based Systems (*CBS*): Lack of source code limits the applicability of many testing and verification techniques that require instrumentation or analysis of the code; Lack of complete documentation limits the understandability of both the code (if available) and implemented functionality; Re-testing the whole product from scratch may be too expensive and reduce the cost effectiveness of component-based technology.

This paper exploits an idea to automatically analyze component integration. The proposed approach consists in first collecting information about component interactions during testing and field execution, and then monitoring interactions for new component versions or for existing components in new software systems, to detect differences with respect to previously observed behaviors. The observed differences provide information about both new behaviors that may correspond to new requirements, and misbehaviors that may correspond to unexpected erroneous interactions. New expected behaviors indicate that the component correctly meets the new requirements; revealed misbehaviors can highlight integration faults.

The development of this approach presents two main difficulties: component interactions can involve complex objects that may be difficult to monitor, and large systems usually involve an enormous amount of interactions that are difficult to synthesize in a manageable set of concepts. This paper proposes a technique, called *Behavior Capture and Test (BCT)* that includes original solutions to both problems, and reports preliminary results that confirm the validity of the approach in gathering useful information about integration faults.

The problem of monitoring complex objects is solved by heuristically identifying a set of methods to extract state information without altering the monitored object. The proposed monitoring technique is referred to as *object flattening*. The problem of synthesizing component interactions is solved by generating two classes of invariants: interaction and I/O invariants. *Interaction invariants* describe the interactions among components, and are computed by synthesizing finite state automata (FSA) that summarize the interaction patterns among components. The many approaches to synthesize FSA from a given set of behaviors do not apply

straightforwardly to the specific context considered in this paper. The paper presents an original algorithm that overcomes the problems of existing approaches. *I/O invariants* describe the relation between data exchanged among components and are computed with Daikon, which is applied to the information produced with object flattening.

The paper is organized as follows. Section 2 overviews the *BCT* approach presenting the main phases and discussing related problems. Section 3 describes the *object flattening* technique that allows the automatic extraction of state information from complex objects, thus enabling the application of *BCT* to real systems. The section first presents the original technique and then the results that confirm its suitability in the considered context. Section 4 illustrates the techniques for distilling I/O and interaction invariants. The section identifies the conditions under which I/O invariants can be generated, indicates the main obstacle in applying existing techniques for generating FSA, and outlines an algorithm that satisfies the identified constraints. Section 5 presents a preliminary evaluation of *BCT* on the well known Pet Store system developed by Sun. The section illustrates the application of *BCT* on the original program and on a different program obtained by modifying few components. Section 6 illustrates further evaluations of *BCT* on four additional cases that sample possible application domains: analysis of a new system developed by reusing existing components, analysis of a new product of the same product line, analysis of a component plugged in different versions of the same system, and analysis of a component used in different systems. Section 7 places our approach in the context of related research. Section 8 summarizes the main contributions of the paper and describes ongoing research.

2 Behavior Capture and Test

BCT analyzes the interactions of reused software components. The technique collects information about the interactions of components when used in existing products, and uses the collected information to identify anomalous interactions of the components when they are updated or reused in new products. Anomalous interactions are behaviors not previously experienced and may be due to either new (legal) uses of the components, or erroneous interactions. In the first case, BCT indicates functionalities that have not been fully tested yet; in the second case, it signals faults and provides useful information for their localization.

The technique applies in two phases: *data collection* and *invariant checking* that are illustrated in Figure 1. The data collection phase involves the components when they are used as part of existing applications, and computes information about components' interactions in the form of invariants. The invariant checking phase involves the update

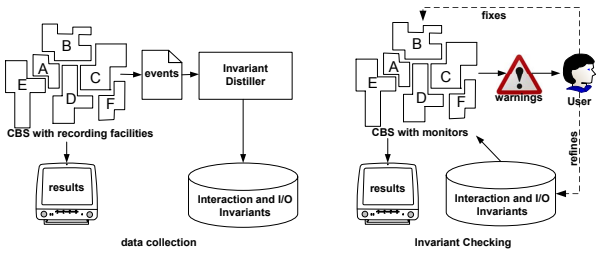

Figure 1. BCT's main phases.

or the reuse of components in new applications, and reveals invariant violations that correspond to interactions not previously experienced.

In the data collection phase, we extract information about components' interactions in the form of streams of events. The extracted events are sent to the *invariant distiller* that synthesizes *interactions* and *I/O* invariants. Interaction invariants are FSA. Each automaton describes the interactions of a service with the system. For example, the service `getItems()` of a `Catalog` component can be associated with the simple interaction invariant `catalog.getItem*`[1], which indicates zero or more invocations of the service. I/O invariants are boolean expressions that describe the relations between requests and results. A service can be associated with many I/O invariants. For example, the service `addItem(CartItem c)` of a `Cart` component can be associated with the I/O invariants `c.getQuantity`> 0 and `c.getUnitCost`$<=$`c.getTotalCost`. The automatic extraction of useful interaction information and the computation of invariants require new techniques for dealing with the complex objects often exchanged between components and for incrementally synthesizing FSA. The problems and the proposed solutions are discussed in Sections 3 and 4, respectively.

In the invariant checking phase, we insert *monitors* that check invariants at run-time in the new system, and signal invariant violations. Violations are interpreted by the designers, who can choose to disable invariants whose violations are benign (that is, behaviors that the designers consider correct despite not having been encountered during the invariant-building phase). Class and component monitors can be generated without using the source code by several technologies: instrumenting the compiled code [12], generating wrappers [3], using middleware functionality [9], and adopting an aspect-oriented approach [11]. We chose a suitable monitoring technology case by case.

The two phases can overlap: we can distill new invariants for analyzing future uses of components (data collection),

[1] In this paper, we indicate FSA with the corresponding regular expressions.

while checking previously generated invariants to identify interaction faults in the current system (invariant checking).

3 Interaction Recorders

Behavior recorders extract information about component interactions. They store the beginning and the end of the executions of the monitored methods in a trace file. Consecutive invocations at the same nesting level represent consecutive executions of methods; nested invocations represent nested executions. Concurrent executions are managed by using different files for the monitored threads, following the approach proposed by Reiss and Renieris [20].

Methods are often invoked with references to complex objects as parameters. Simply recording the references would provide little information for generating invariants. To record useful information, we need to gain information about the state of the objects referred to by the parameters. BCT automatically extracts state information using the objects' interfaces, therefore it can be applied when source code is not available and also to objects of languages different from Java, e.g., C++ objects. Classic approaches, like directly encoding the internal contents of objects with special inspectors, require access to the source code that may not be available when reusing components, thus we developed a new technique. The technique, hereafter referred to as *object flattening*, consists of (semi-automatically) identifying *inspectors*, i.e., public fields and methods that return state information without altering the object. The objects' interfaces are usually part of the specification of components; however they can be also automatically extracted at run-time if reflection APIs are available.

If the object is an instance of a class belonging to the standard Java distribution, inspectors are known. For example, the inspectors of the `java.io.File` class are `getAbsoluteFile()`, `isFile()`, `toURL()`, etc.... The inspectors of a class of the standard Java distribution are *consistent* and *complete*, i.e., they do not alter the state, and provide information about the whole accessible state.

If the object is an instance of a user-defined class, we identify possible inspectors heuristically. Our heuristic derives from conventions widely adopted by Java software developers: inspectors are identified syntactically according to the rules shown in Figure 2. Other heuristics can be added to benefit from conventions that may derive from the application domain, the programming language, the company standards or any other source.

The technique applies recursively on complex objects: if the state of the object includes other objects, object flattening recursively extracts state information until it obtains a primitive value, reaches a reference already inspected, or reaches a maximum depth.

Object Flattening has been implemented in a prototype tool that is available at `www.lta.disco.unimib.it/objectflattener`. The prototype object flattener supports several models and formats for the extracted data, and provides extension mechanisms to cope with additional models or formats. The flattener can be extended also with plug-ins that implement specific strategies to select and invoke inspectors of particular sets of classes. Plug-ins are used to analyze objects of particular importance for the target application.

`<any>get<*>()`	methods that return any data type, without parameters and with a signature's name starting with `get`
`<any>elements()`	methods that return any data type, without parameters and with `elements` as signature's name
`<any>length<*>()`	methods that return any data type, without parameters and with signature's name starting with `length`
`<any>[]toArray()`	methods named `toArray` that return an array
`<Boolean>is<U*>()`	methods that start with `is`, followed by an upper case letter and that return a boolean value
public fields	all public fields

Legend

`<any>`	any return type,
`<any>[]`	an array of any type,
`()`	a method with no parameters,
`<*>`	any sequence of characters,
`<U* >`	an upper case letter followed by any sequence of chars.

Figure 2. Heuristic for recognizing Java inspectors from idiomatic names and signatures of methods

Evaluation of Object Flattening In general, heuristically identified inspectors are neither *consistent* (guaranteed not to alter the state) nor *complete* (guaranteed to completely characterize the object state). Fortunately, a few missing inspectors do not significantly alter the effectiveness of BCT, but large omissions would not be tolerable. Thus, before using the heuristic, we validated object flattening on some case studies: the *Pet Store*, a well known example developed by Sun Microsystems with EJB technology, *Hermes*, Java middleware for developing mobile agent-based applications, and a University web presence developed in Java as part of a student project. The considered systems range in size from 120 (University web presence) to 500 (Pet Store) files and several tens of components.

We guarantee the consistency of object flattening by using an aspect [11] that intercepts any attempt to write an object field and prevents its modification if object flattening invoked the method that modifies the field value.

We measured the completeness of the set of inspectors automatically identified by our prototype object flattener as

the percentage of classes that can be fully examined with the automatically identified inspectors. A class is fully examined if the inspectors give full visibility of its internal state. The results are presented in Table 1. The first column presents the percentage of fully examined classes over the whole application; the second column considers only data classes, i.e., classes that implement the application entities. Such classes are the ones that contain relevant information about data, and thus are the primary target of this kind of analysis. We will see in Section 6 that we found it useful to disable such analysis when dealing with components without relevant data content. The technique provides reasonable results for the whole systems (first column), and becomes very effective for data classes (second column). Object flattening mostly fails when dealing with classes implementing control logic, e.g., the mailer component or the component supplying orders. However, these components can be monitored by observing requests and results that are used by BCT to compute I/O invariants.

	full system	data classes
Pet Store	68%	98%
Hermes	43%	88%
Univ. Web Presence	80%	100%

Table 1. Percentage of classes fully inspected.

Scalability of Object Recorders The object recorders introduce time and memory overhead. Time overhead depends on the complexity of the state of the observed objects. Memory overhead depends on the size of the recorded traces. We measured the overhead of the object recorders using a prototype for monitoring Jedit, a popular editor.

To measure the time overhead, we distinguished four different classes of objects: small, medium, large objects and collections. Small objects contain only simple state attributes; Medium objects contain references to other objects up to a small depth (we fixed a depth of 7 as threshold); Collections store sets of items; Large objects are the objects that do not match any of the former characteristics. Figure 3 shows the overhead that we measured when extracting state data from the different classes of objects. The horizontal line represents the threshold over which we perceived the application slowing down.

The cost of extracting state information from small and medium objects is almost irrelevant. The cost of extracting state information for large objects becomes critical when the depth of analysis increases (we found relevant slow down for depth over 15). Collections are difficult to manage also for small values of depth, since their size tends to affect performance more than the depth of the analysis. Some of these cases can overlap, for example analysis of large objects can include a reference to a collection. In this case, the cost of analyzing a large object can increase heavily also for small values of the depth.

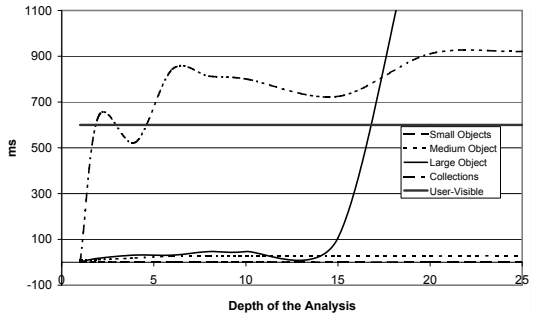

Figure 3. State extraction costs

The costs of extracting state data may quickly increase in the presence of tightly coupled components. Such cases are usually localized and can be easily identified and bypassed by disabling part of the analysis and useful results can still be collected. Excessive overhead can be overcome by disabling the object flattener and collecting only the information provided by the monitors. In this way, we can contain the overhead within an acceptable threshold and yet be able to effectively analyze the component interactions.

The measured time overhead indicates that object flattening does not prevent the execution of the monitored system, but can reduce the performances in some cases. Thus it can be used without particular limitation during testing, but requires some attention when used in field execution, where it finds two possible applications: *shallow* and *target monitoring*. In shallow monitoring, we fix a small upper bound for the analysis depth ($<= 5$), we disable analysis of collections, and we allow high depth values only for specific components that need careful analysis. In target monitoring mode, we apply monitors only to a few interesting or critical components. Executions did not reveal significant memory overhead when data are stored in a compact form and traces are incrementally overwritten after updating the invariants with the new traces. Some optimizations such as the one proposed by Reiss and Renieris further reduce overhead [20]. Only large objects and collections typically result in important memory consumption, which needs to be estimated before long runs.

4 Invariant Distillers

The invariant distiller uses the information extracted by the interaction recorders to synthesize invariants that describe the interplay between components in a running system. The distiller produces two kinds of invariants: I/O and interaction invariants. I/O invariants describe the relation

on the values exchanged between components during the execution in the form of boolean constraints. Interaction invariants capture the sequences of methods invocations in the form of FSA.

I/O invariants can be easily computed using Daikon [7] thanks to the information provided by the object flatteners used in the interaction recorders. Daikon automatically infers invariants over scalar and structured variables from a set of positive samples. The inference starts with a large set of assertions over the monitored variables, and incrementally eliminates assertions violated by single values. The set of live assertions describes the relations observed on the set of values as invariants over the observed behaviors.

Interaction invariants are FSA, whose edges are labeled with strings of the form `C.s`, where `s` is a service implemented by the component `C`. The regular language generated by the FSA corresponds to the set of interactions that can take place when the service associated to the invariant is executed.

The characteristics of the application impose several requirements on invariant inference. Some requirements simplify the problem, but others limit the applicability of known algorithms. The algorithm must identify *loops*, *conditional branches* and *sequential flows*, and must accept *already generated sub-behaviors*. This excludes algorithms that do not work well in some of these cases [2]. The application domain provides only *positive samples*. This excludes algorithms optimized for negative samples, i.e., samples that must not be recognized by the generated FSA [4]. All provided samples *must be included* in the FSA *independently* from the *frequency* they occur. This excludes approximate and probabilistic inference algorithms [1, 18]. The FSA must be built *incrementally* by removing old samples as soon as they are used, to limit memory consumption. This excludes algorithms that require the whole set of samples available at the beginning of the computation [5, 20]. The algorithm cannot rely on any *teacher*, i.e., externally provided information [4].

The well known `kTail` algorithm proposed by Biermann and Feldman provides a good starting base, but it is not incremental, and generates several redundant nodes when identifying ends of loops [2]. Cook and Wolf solved some limitations of the `kTail` algorithm when dealing with loops, but their algorithm is non-incremental and still includes duplicated nodes in the inferred loops [5].

To meet all application requirements, we developed a new algorithm that incrementally extends the FSA for each new behavior. Our algorithm works as follows. It starts with an empty automaton. It examines each behavior only once. If the behavior is already recognized by the automaton, the behavior is discarded and the FSA is not modified. Otherwise, the algorithm identifies sub-behaviors already generated by some subautomata and augments the FSA to connect the identified subautomata to include the considered behavior. The behavior is then discarded. In particular, given a new behavior `newb`, the algorithm extends the automaton `FSA` as follows[2]:

(1) it searches for the longest prefix `pre` of `newb` generated by `FSA` and removes `pre` from `newb`

(2) if after step 1 `newb` is not empty, it searches for a sub-behavior `sub` of `newb` of minimal length k that is generated by a sub-automaton `subFSA` of `FSA`. The algorithm works in two possible modes: either it stops when it finds the first sub-behavior of length greater than a given threshold `cutOffSearch`, or it searches the whole FSA for the longest sub-behavior of `newb` generated by a sub-automaton of `FSA`. `cutOffSearch` represents the optimization that we are ready to tolerate for gaining performance. In our case studies we used $k = 2$ and $cutOffSearch = 4$.

(3) if `sub` is not empty, the algorithm extends `FSA` by suitably "connecting" the state reached with `pre` at step 1 with the initial state of the sub-automaton `subFSA` that recognizes `sub`. It removes the prefix that includes `sub` from `newb`, considers the state reached with `sub` as the new initial state of `FSA`, and goes back to step 1.

(4) Otherwise (`sub` = empty), the algorithm terminates extending `FSA` by suitably "connecting" the state reached with `pre` at step 1 with a new subautomaton that generates the current content of `newb`.

The FSA generated with our algorithm generates behaviors that suitably represent possible interactions among components reducing over-generalization and over-restrictiveness of other inference algorithms [14]. Moreover, when our algorithm considers a new behavior, it identifies both new cycles that derive from the new behavior and sequences that are already generated by the current FSA in a single step, while "classic" algorithms first augment the FSA with the new sequence and then optimize the augmented FSA, with an extra overhead [2, 5, 20].

It is possible to show that at each step all behaviors considered up to that point are generated by the FSA produced with our algorithm. Thus the strings (behaviors) do not need to be saved. The prototype implementation used in the case studies is available from `www.lta.disco.unimib.it/kbehavior`.

5 Preliminary Evaluation

We evaluated BCT in two steps: first, we considered a simple case study to verify the effectiveness of the approach; then, we identified a set of further experiences

[2] a precise description of the algorithm can be found in [14].

aimed at verifying the effectiveness of BCT in different contexts. This section summarizes the preliminary results that we obtained with a well know case study: the *Sun Pet Store*. The next section describes the main results obtained with additional case studies.

The *Sun Pet Store* is a Java J2EE web-shop developed by Sun Microsystems. It implements a set of common functionalities of a web-shop, such as user authentication, cart management, catalog browsing, and administration, and exploits common features of component-based systems, such as persistence, transactions and messaging. It consists of about 500 files that implement 20 components of varying complexity.

The experience with the *Sun Pet Store* is aimed at evaluating the effectiveness of BCT in detecting behavioral incompatibilities among different products belonging to the same product family. To this end, we needed two versions of the same system differing for some components. We used the original Sun version, hereafter version *A*, and a different version, hereafter version *B*. Version *B* includes new versions of four key components, *catalog*, *cart*, *customer* and *contactInfo*, and uses a new configuration setting. All new components implement simple, but semantically relevant changes in the service requirements. We used the invariants computed by monitoring the behavior of the new components in version *B* to analyze the run time behavior of version *A*.

We computed the invariants on version *B* for components *catalog*, *cart*, *customer* and *contactInfo* by first executing a set of 95 test cases derived from the new requirements, and then simulating normal use through a group of students who were asked to use the system to shop for pets.

We then executed version *A* with the same test cases, and we evaluated the invariants computed on version *B* for the selected components. BCT reported several violations of both interactions and I/O invariants due to the different behavior of the monitored components and of the configuration setting. Here we discuss in detail some relevant violations. Interested readers can find additional information in [13].

The I/O invariants that were automatically extracted from objects of type `HTTPServletRequest` mostly provided information about changes in the configuration settings: host name, location of configuration files, port numbers, etc. Such information is helpful in identifying configuration problems that are often hidden and difficult to reveal. We revealed the violation of the I/O invariant `request.getSession...getHostName[]=="sirio"` that indicates that the application was executed on a different computer. This violation is not particularly meaningful, but in general, such information can be useful in identifying faulty interactions related to configuration problems.

The violation of the following interaction invariants of component `ShoppingCart` provided information about unforeseen performance problems (each row indicates the name of the method implemented by `ShoppingCart` and the corresponding interaction invariant):

```
(): CatalogHelper.(),
getItems: CatalogHelper.getItem*,
getSubTotal: CatalogHelper.getItem*.
```

The invariants indicate that in version *B* the creation and following uses of a `ShoppingCart` create and uses a single instance of `CatalogHelper`. Version *A* executed with the same test cases violate the invariants because in version *A* the component `ShoppingCart` creates a new component `CatalogHelper` for each interaction. The different behavior of the two versions does not result in functional differences, but in performance differences that may become problematic only in the presence of heavy load conditions, when unexpected problems may be difficult to diagnose and repair.

Version *A* violates also the following interaction invariant of the method `perform` of `CartEJBAction`:

```
(ShoppingCart.getItems ShoppingCart.addItem)|
(ShoppingCart.updateItemQuantity)* |
(ShoppingCart.deleteItem).
```

The violation indicates that version *A* invokes `addItem` without previously invoking `getItems`. In this way, the component updates the quantity in the cart ignoring the current quantity. Therefore, two purchases of the same product result in a cart containing only one item of the product, and not two as expected.

The early experience indicated that the use of BCT is simple and useful. The possibility of incrementally disabling checks of invariants reduces useless warnings. The first runs immediately revealed most configuration violations and expected invariant violations, i.e., violations that result from new requirements. The corresponding invariants have been easily checked and disabled. In this way, most additional violations provided useful information about the behavior of the new version.

This early experience revealed problems in the recording phase in presence of collections and polymorphic objects, since they both lead to records with different structures that need pre-processing before invoking Daikon. Daikon supports inference of invariants over collections, but requires the modification of configuration files to match the specific structure of the trace. Different polymorphic instances of the same object type can also be handled by suitably adapting both trace files and the Daikon configuration files.

6 Additional Case Studies

After the encouraging results of the preliminary evaluation reported in the previous section, we defined a validation plan that includes several case studies. Here we sum-

marize the most relevant results. The reported results represent a sample of possible usages of BCT: development of a new component-based application by reusing existing components (*computer store*), development of a new application within a product line (*Hermes*), "hybrid" applications, i.e., applications that merge components and object-oriented classes (*Jedit*), and components re-used in different application domains (*XML parser*).

Computer Store This first case study simulates the development of a new application by reusing available components already used and monitored in previous applications.

We considered a set of components of the Sun Pet Store, for which we derived invariants, as described in the former section. We then recruited a team of 20 senior students for a period of 5 months, and we asked them to develop a *Computer Store*. The initial Computer Store requirements were analyzed and specified independently of the Sun Pet Store. The team was then asked to design and develop the Computer Store according to the requirements, by reusing components selected from the Sun Pet Store when possible. As a result, the Computer Store reuses some of the selected components, and includes both some modified components of the Pet Store and some new components.

We then executed the Computer Store with a set of test cases developed by an independent group of testers. During the execution, we monitored the components reused from the Sun Pet Store, and we evaluated the invariants computed on the Sun Pet Store for these components.

We detected several invariant violations. Many violations derive from different uses of the components in the new context. In fact, the Computer Store uses different sets of values; for instance, new labels displayed in the application and new naming conventions for products. BCT signals all these differences. The feedback is useful for identifying possible unexpected uses and verifying that the current implementation satisfies the new requirements. The possibility of incrementally masking invariants was appreciated by the testers who could eliminate useless noise after checking the correctness of the new values. Testers found this feedback very useful for increasing confidence in the components.

One of the detected violations allowed a fault to be quickly identifying and diagnosed, which otherwise would be difficult to reveal and locate. The Computer Store implements catalogs with arbitrary nesting levels, while the Pet Store supports only two fixed catalog levels. The component *catalog* has been suitably modified to support recursive exploration of the catalog. Execution of the Computer Store violated the following interaction invariant of the *catalog*: $S|(SCC^*R)$. In the invariant, S is a call to the service locator, C is the creation of a category and R is the production of the result. The invariant states that the Sun Pet Store calls the service locator exactly once, before possibly creating one or more categories, and then producing the result. The violating trace is prefixed by SS and thus indicates that the Computer Store calls the service locator more than once. A simple inspection of the code guided by the violating trace revealed the subtle fault: when the recursive exploration of the catalog backtracks and closes the current database connection, it erroneously closes all connections opened in the previous steps of the recursion, because connections are erroneously shared among the different invocations. This is a typical integration fault that can be hard to diagnose in a large system: The new component was clearly designed under the hypothesis of gaining multiple distinct connections to the database, while the database access was designed to provide only a unique connection.

This case study brought the usefulness of BCT to our attention for supporting also debugging, by providing information in terms of violating traces.

Hermes The second case study studied product lines, which represent a straightforward application of BCT, since different products of the same line share many components that are usually available in many versions to cope with different requirements.

In this case study, we considered *Hermes*: a component-based middleware for distributed mobile agent-based systems, developed in Java (`http://www.bioagent.net/hermes`). *Hermes* is composed of about 320 files that implement 10 components. We considered versions 2.02 and 2.05. We executed versions 2.02 with an available set of test cases and we distilled invariants with BCT. In this case, we focused on the analysis of the logic of control, because the features of a middleware mostly concern control aspects, and for this class of interactions, the object flattening technique likely produces irrelevant results with a significative overhead. We then executed version 2.05 with the same set of test cases, and we evaluated the invariants produced for version 2.02.

Initially, BCT reported an enormous amount of invariant violations, due to the presence of a new logging functionality in version 2.05. After the first executions, we identified the nature of the violations. We solved the problem of these "anomalous violations" by temporarily disabling the checking phase, and enabling extensions of the current set of invariants with interactions related to the new functionality only. In this way, we were able to eliminate "spurious violations", i.e., violations produced by the new functionality, without masking other possibly interesting violations.

The reported violations highlighted a subtle problem that could go unrevealed for a long time before causing relevant problems: all calls to the logger are synchronous, thus if the logger fails, e.g., because the log file consumes the available memory, the overall computation can be compromised.

BCT revealed also a second relevant problem related to the construction of invariants. When building interaction

invariants, BCT pairs starts and ends of method calls to reconstruct the nesting of method calls. In this case, BCT has not been able to pair all starts and ends, and this has been easily tracked back to incorrect thread management: some threads were not properly terminated when the platform was shut down. Spurious live threads may cause problems in particular situation even long after the termination of the application and are thus often very difficult to reveal and diagnose.

This experience confirmed that BCT is able to direct the attention of testers to the few relevant problems present in the new platform. The case study also highlighted the "pervasive" changes problem. The problem of too many spurious violations occurs in general when the monitored system introduces new "pervasive" functionality: if the new functionality is localized in a single cluster of components, we can isolate spurious violations by either not monitoring these components or by considering the violations related to the cluster at a different time; but when the new functionality pervades many components, we cannot isolate all involved components. The approach that we first tried with *Hermes* of temporary disabling the checking phase to enable a partial update of invariants, before re-enabling the checking phase, has sufficed each time we encountered the pervasive changes problem across several example systems.

Jedit In the third case study, we investigated the application of BCT with a "hybrid" system, i.e., a system composed of proper components plugged into a classic object-oriented system. *Jedit* (`http://www.jedit.com`), an extensible text editor composed of about 1000 files is a classic Java application that can be extended with plug-in components. We integrated *Jedit* with the *JdiffPlugin* component, a plugin that compares different files. We studied a previously known problem with the plugin when the system is integrated.

We first executed version 4.1 of *Jedit* with `JdiffPlugin v.1.3.2`. We simulated normal execution, monitored the execution, and distilled invariants. We then updated *Jedit* to version $4.2pre9$; we executed it with the same version of `JdiffPlugin`, and we evaluated the invariants automatically computed during the executions of the previous version of *Jedit*. We were aware of a new problem with `JdiffPlugin` when moving from version 4.1 to version $4.2pre9$. We also knew that the fault was not easy to identify and diagnose, and we wanted to evaluate the ability of BCT to help reveal the fault and provide enough information to diagnose it.

When the execution of version $4.2pre9$ failed as expected, BCT reported a violation of the I/O invariant `ret`$\neq$`null` of method `findScrollBar`. The information provided by the invariant violation is precious, since it points to method `findScrollBar` for the possible fault. A quick examination of version $4.2pre9$ indicates that the scrollbar has been removed from the new version thus causing the unexpected failure when the new version is integrated with the plugin.

The cases study confirmed the usefulness of BCT for identifying and locating unexpected faults that can be difficult to diagnose otherwise.

XML Parser The last case study reported in this paper dealt with components used in different application domains as COTS components often are. We considered the Ælfred XML Parser, a COTS component for parsing XML documents, and we examined its use in *Jedit v4.2pre9*, the application discussed above, and in *PtPLot 5.3*, a 2D Java plotter developed at Berkeley (`ptolemy.eecs.berkeley.edu/java/ptplot/`). Both applications use the Ælfred XML Parser to input XML files.

In the experience, we considered only the interactions with Ælfred XML Parser, and we generated only the interaction invariant, since the interface used by the component to communicate with the system produces many method calls that would lead to very high overhead if monitored for I/O behaviors, as already discussed in Section 3. The interaction invariant for the Ælfred XML Parser reflects the structure of the parsed configuration files, and in this case can be generated with just a few runs, since it does not depend on the particular data exchanged between the component and the system.

As expected, the structure of the parsed document in *PtPlot* is different from that in *Jedit*. In fact, *PtPlot* uses XML documents to store pictures to be plotted on the screen, while *Jedit* stores textual information. To skim irrelevant violations, we use the same disabling/enabling procedure introduced in the *Hermes* case study. Once the invariant was adapted to *PtPlot*, we monitored the execution of the system by simulating normal usage. BCT reported an invariant violation for an unexpected behavior of the system: the loading of an empty file. Although the behavior of the systems was legal, knowing that the input file was not empty could have signaled the problem. The reported invariant violation not only confirmed the presence of a problem, but provided important information for the localization and diagnosis of the corresponding fault, which was due to an incorrect assumption on the component behavior: the component does not check for the syntactic correctness of the input XML file and signals a parsing problem to the system when it meets an illegal XML item; the system assumes that the component performs syntactic checks on the input XML file and does not react properly to an error signal, thus producing an empty canvas when the input file contains an error. The invariant violation allowed us to discard any other possible cause of the problem, and immediately diagnose the fault.

Once more, the case study confirmed the applicability of BCT not only to identify problems, but also to provide useful diagnosis information.

7 Related Work

This paper proposes an original method for automatically analyzing components' integration at run time. The method includes techniques for invariant detection and grammar inference. We already discussed in Section 4 the inadequacy of classic grammar inference algorithms in the specific context to motivate the definition of a new algorithm. In this section, we briefly compare the object flattening technique discussed in Section 3 with the facilities offered by Daikon for analyzing object-oriented software, and we then compare BCT with other approaches to component run time analysis.

Daikon can detect invariants on scalar and structured variables, but it is not directly applicable to object-oriented software [7]. The extension to object-oriented software proposed in [8] "linearizes" object attributes, i.e., writes object attributes into arrays compatible with Daikon. This extension requires the instrumentation of the source code that is not realistic in the case addressed in this paper that deals with components reused often without source code and with incomplete specifications. The object flattening technique proposed in this paper extracts state information by identifying inspectors, without accessing either the source code or the specifications, thus overcoming the limitations of the linearization approach. Moreover, the object flattening technique also automatically extracts derived data values, i.e., values obtained by elaborating object's attributes.

Methods for analyzing components' integration at run-time have been proposed by Ernst, who first exploited static verification with Nimmer, and then service verification with McCamant; by Raz, Koopman and Shaw who addressed *data feed systems*; and by Hangal and Lam who proposed the DIDUCE tool.

Nimmer and Ernst exploited static verification performed by ESC/Java to confirm properties dynamically inferred by Daikon [17]. The integration of two complementary technologies helps overcome limitations of either approaches in isolation, but static analysis requires the availability of the source code, and thus the approach is applicable in a smaller context than BCT, which works for the many cases of reuse of components available without source code.

McCamant and Ernst focused on component updates. They use Daikon invariants to infer pre- and post-conditions of the services of the components and of the corresponding updates, and then use the inferred conditions to prove the safety of the updates [16]. The formal proof of correctness proposed by McCamant and Ernst requires a complete set of pre- and post-conditions for both the component and its updates. BCT uses the generated invariants for checking the compatibility of components at run-time. In this way, BCT does not require a complete set of invariants for components and updates, and so is able, unlike the technique proposed by McCarmant and Ernst, to work with the incomplete information that is often available.

Raz, Koopman and Shaw studied *data feed systems*, i.e., systems that provide services to (remote) clients based on data sources available on-line [19]. They use Daikon to compute invariants over multiple numeric fields, and statistical estimation techniques to compute invariants over single numeric fields. The computed invariants are used at the client-side to monitor evolution of *data feed systems* with respect to both updates of implemented functionalities and relevant modifications of data sources. This approach shares a common goal with BCT, but differs in many ways: (1) I/O invariants used in BCT work with complex objects, while client-side invariants used by Raz, Koopman and Shaw deal with numeric values only; (2) BCT uses invariants for tracking the evolution of both the system and its components, to verify the correctness of the current implementation, while Raz, Koopman and Shaw's technique signals anomalous values only; (3) BCT associates invariants to components, while Raz, Koopman and Shaw use invariants only at the client side. Raz, Koopman and Shaw's statistical estimation is particularly useful on large data sets that evolve regularly, but less useful on irregular peer interactions such as the ones taking place between components.

The DIDUCE tool instruments the source code to derive invariants, which are continuously verified and updated at run-time [10]. The technique focuses on the resource consumption problem: lightweight computation of invariants is obtained at the cost of limited expressiveness of the inferred invariants. The DIDUCE invariants can only indicate that the target value is always constant, positive, negative, odd, even or approximatively bounded. This approach is particularly effective for debugging, since anomalous behaviors often leave long traces of violated invariants that can help in tracking the failure to the fault which caused the failure, but is less effective in our context. BCT prioritizes expressiveness over computation time to better address the target domain.

8 Conclusions

This paper proposes BCT, a technique for run-time analysis of systems that reuse components. BCT first derives information about component behavior when they are used as part of existing systems, and then uses the inferred information for analyzing the compatibility of the components when reused in new systems. Comparing invariants satisfied before and after a change, or in different contexts, has been described in the literature. Such an approach should be useful for component integration testing, but for that it must be adapted to practical constraints of component integration, including unavailability of source code.

We monitored components completely at the interface level using wrappers and aspects, and exploiting reflection when it is available. Invariants are collected and checked automatically, with the option of programmer control to distinguish between intentional and unintentional differences in behavior. Invariants on input/output data are inferred with object flattening, which produces information elaborated with Daikon. Invariants on interaction patterns are distilled as finite-state acceptors by an incremental algorithm for inferring FSA. The incremental algorithm proposed in this paper solves space problems that would otherwise unacceptably limit the applicability of the approach.

The paper reports preliminary results obtained with a prototype on a set of applications that represent different scenarios. The case studies demonstrate that BCT can successfully detect incompatibilities that depend on the interactions between components and that data exchanged between them. The case studies show also that BCT provides information useful for understanding the application, diagnosing the faults and debugging the components.

We recognized that invariants computed for a component can be influenced by the way the system uses the component. In particular, invariants depend on both the data used as input and the way the provided services are used by the system. We found functional testing very useful for deriving an initial set of invariants independent from the specific use of the component. In this way, we can eliminate many initial warnings that depend on the different use of the components within different systems.

Our research agenda includes a wider set of experiences joined with tuning the prototype, and further studies on heuristics in order to produce regression test suites to be used when integrating components into new systems. Preliminary results in this direction are described in [15].

References

[1] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *proceedings of the 29th Symposium on Principles of Programming Languages*, pages 4–16. ACM Press, 2002.

[2] A. Biermann and J. Feldman. On the synthesis of finite state machines from samples of their behavior. *IEEE Transactions on Computer*, 21:592–597, June 1972.

[3] K. Brown. Building a lightweight COM interception framework, part 1: The universal delegator. *Microsoft Systems Journal*, January 1999.

[4] O. Cicchello and S. C. Kremer. Inducing grammars from sparse data sets: a survey of algorithms and results. *Journal of Machine Learning Research*, 4:603–632, 2003.

[5] J. Cook and A. Wolf. Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology*, 7(3):215–249, 1998.

[6] I. Crnkovic and M. Larsson. *Building Relyable Component-Based Software Systems*. Artech House, 2002.

[7] 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.

[8] M. D. Ernst, W. G. Griswold, Y. Kataoka, and D. Notkin. Dynamically discovering pointer-based program invariants. T.R. UW-CSE-99-11-02, Univ. of Washington, 1999.

[9] M. R. G. Rackl, M. Lindermeier and B. Suss. MIMO - an infrastructure for monitoring and managing distributed middleware environments. In *proceedings of the Multimedia Middleware Workshop at FIP/ACM International Conference on Distributed systems platform*, pages 71–87. Springer-Verlag, 2000.

[10] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *24th International Conference on Software Engineering*, pages 291–301. ACM Press, 2002.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *proceedings of the European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, 1997.

[12] H. Lee and B. Zorn. BIT: A tool for instrumenting java bytecodes. In *proceedings of the USENIX Symposium on Internet Technologies and Systems*, pages 73–82, 1997.

[13] L. Mariani. Capturing and synthesizing the behavior of component-based systems. Technical Report LTA:2004:01, Università degli Studi di Milano Bicocca, 2004.

[14] L. Mariani. *Behavior Capture and Test: Dynamic Analysis of Component-based Systems*. Phd thesis, Università degli Studi di Milano Bicocca, January 2005.

[15] L. Mariani, M. Pezzè, and D. Willmor. Generation of integration tests for self-testing components. In *proceedings of the 1st Internationl Workshop on Integration of Testing Methodologies*, volume 3236 of *LNCS*, pages 337–350. Springer, 2004.

[16] S. McCamant and M. D. Ernst. Predicting problems caused by component upgrades. In *proceedings of the 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 287–296. ACM Press, 2003.

[17] J. W. Nimmer and M. D. Ernst. Static verification of dynamically detected program invariants: Integrating daikon and ESC/Java. In *1st Workshop on Runtime Verification*, 2001.

[18] A. Raman, P. Andreae, and J. Patrick. A beam search algorithm for PFSA inference. *Pattern Analysis and Applications*, 1(1):121–129, 1998.

[19] 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. ACM Press, 2002.

[20] S. P. Reiss and M. Renieris. Encoding program executions. In *proceedings of the 23rd International Conference on Software Engineering*, pages 221–230. IEEE Computer Society, 2001.