Compatibility and regression testing of COTS-component-based software

Leonardo Mariani, Sofia Papagiannakis and Mauro Pezzè
Università degli studi di Milano Bicocca, DISCo
via Bicocca degli Arcimboldi, 8 - 20126 Milano, Italy
{mariani,papagian,pezze}@disco.unimib.it

Abstract

Software engineers frequently update COTS components integrated in component-based systems, and can often chose among many candidates produced by different vendors. This paper tackles both the problem of quickly identifying components that are syntactically compatible with the interface specifications, but badly integrate in target systems, and the problem of automatically generating regression test suites. The technique proposed in this paper to automatically generate compatibility and prioritized test suites is based on behavioral models that represent component interactions, and are automatically generated while executing the original test suites on previous versions of target systems.

1 Introduction

COTS components are third-party software modules that are integrated into software products to reduce development time and effort. COTS components are usually provided as binary code with natural language specifications of integration and deployment procedures [21]. Even if extensively tested in isolation and in several contexts, system integrators must re-test COTS components in the specific contexts to look for possible integration faults not revealed in previous contexts. Unfortunately, lack of both source code and complete specifications hinders applicability and effectiveness of many classic testing and analysis techniques, and challenges system integrators and test designers [1].

Vendors frequently release new versions of popular components, to keep their products competitive in the market. For instance, Crystal Report (http://www.businessobjects.com), a popular component for creating reports, is updated on a monthly basis. Consequently, software engineers often update COTS components integrated in their systems with either new versions of the same COTS products or equivalent COTS products, to keep the overall system up-to-date and competitive. After each update, test designers must design and execute regression test suites, to check that no new faults are introduced.

The characteristics of COTS components reduce the techniques that can be used for regression testing. Test designers have two main options: They can either re-execute all test cases or select regression test cases by identifying changes between binary files and selecting the test cases that exercise the changes [25, 20].

Re-executing all test cases for the application with the new COTS component can be time consuming, especially when test suites are large or test case execution cannot be fully automated and results must (at least partially) be inspected by testers. For example, in our experience with the *GiniPad* Java editor [7], re-execution of the full test suite required one working day (see Section 6 for details).

Selecting test cases according to changes in binary files is effective only if the new and old components are built within equivalent environments (same compiler, same compiling options, ...), and the components are similar versions of the same software unit. Both requirements are needed to usefully reduce the test suite: files built within different environments are likely to be very different even if changes between components are relatively small; if changes between old and new versions are pervasive, the changes will involve most of the binary code.

Unfortunately, COTS components seldom satisfy both requirements: System integrators have no control over the generation of COTS components, and do not know if components are built within the same environment. Moreover, the nature and size of changes can be limited between consecutive versions, but it is often pervasive between different releases of the same component or between components of different vendors. For instance, releases that incorporate novel technologies or novel architectural solutions are likely to introduce many structural differences that can lead to re-execution of most, if not all, of the test cases.

Many popular components often implement de-facto standard interfaces, e.g., many XML parsers and GUI components, and thus can be easily interchanged. In these cases, system integrators can chose a replacement of an obsolete component among many alternatives. However, verifying the compatibility of all candidates may be extremely time consuming. For example, in one of our experiences we evaluated 15 alternative components for managing the text area of the *Ginipad* editor. Re-executing all test cases for the 15 different versions of the editor obtained for the 15 alternative components to test for compatibility, requires about 15 working days. Unfortunately no effective techniques exist to quickly skim the alternatives and focus on a subset of likely compatible components to select the best one.

In this paper, we tackle both the problem of compatibility testing among alternative components and the problem of generating efficient regression testing for compatible COTS components. In particular we propose: (1) a technique for automatically deriving small *compatibility test suites* to efficiently evaluate several alternative candidate components that can replace a COTS component within a software system, and (2) a technique for *prioritizing test cases* to improve the efficiency of regression testing of COTS components, when integrated in new software systems to update obsolete components.

Differently from most popular regression testing techniques that are based on either source code or specifications, which are seldom available to the system integrators who use COTS components, our technique relies on automatically generated behavioral models to generate both compatibility test suites and prioritized regression suites. The behavioral models that we automatically derive from execution traces represent two aspects of component integration: properties of the data exchanged between components and sequences of invocations to components' interfaces.

In this paper we describe the technique and we report preliminary results, which indicate the effectiveness of the approach: Compatibility test suites quickly reveal common integration problems and efficiently support testers in identifying components compatible with the considered system; Prioritized regression test suites reveal most integration faults after the execution of a small fraction of the test cases, thus reducing the overall test and debug cycle, and providing efficient results even when resource constraints limit the amount of test cases that can be executed. In the experience reported in the paper we revealed all faults that were revealed by the original test suite by executing less than 10% of the test cases, for 14 out of the 15 considered COTS components.

The paper is organized as follow. Section 2 introduces the technique. Section 3 overviews BCT, the approach that we use for generating behavioral models. Section 4 discusses the coverage criteria that we use to select compatibility test suites. Section 5 presents the technique that we propose for prioritizing regression test cases. Section 6 illustrates the results obtained with preliminary experiences. Section 7 discusses related work, and Section 8 indicates future research directions.

2 Regression Testing of COTS Components

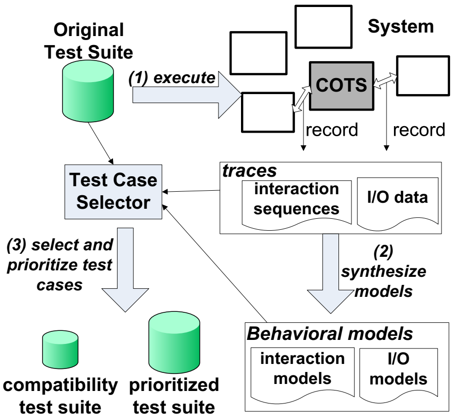


Figure 1. Generation of compatibility and prioritized regression test suites.

The technique presented in this paper supports compatibility and prioritized regression testing of components. *Compatibility testing* aims to quickly identify incompatibilities when updating components, and is particularly useful to quickly eliminate most of the incompatible components among many candidates. *Prioritized regression testing* aims to define a priority schema that increases the likelihood of revealing faults early during test execution, thus helping developers reduce the overall costs of the test-debug cycle, and select efficient subsets of regression test cases.

Compatibility test suites and test case prioritization schemas are generated when testing the original component-based software, and then used when updating components in the original software. Figure 1 shows the process of generating compatibility test suites and test case prioritization schemas, while Figure 2 illustrates a typical usage scenario of compatibility and prioritized test suites.

We automatically generate compatibility and prioritized test suites from behavioral models of components. The generation is composed of three main steps: component monitoring, behavioral model generation and test case selection.

We monitor components with a suitable infrastructure that traces interactions between components and system.

We automatically synthesize behavioral models from execution traces with the BCT technology, which is described in [13], and is briefly summarized in the next section. BCT produces behavioral models that describe two aspects of interactions between components: sequences of invocations

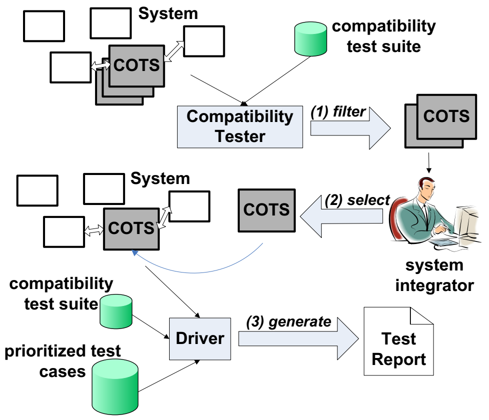


Figure 2. Typical usage scenario of compatibility and prioritized test suites.

(*interaction models* given as finite state machines annotated with inter-component invocations) and properties of data values exchanged between components (*IO models* given as boolean expression over data values).

We select compatibility test suites from behavioral models, and we prioritize regression test cases according to both behavioral models and execution traces.

Behavioral models allow us to group test cases according to the interactions of the software system with the tested component, and can be used to select a small subset of test cases that represent all classes of interactions of the system under test with the component of interest. Compatibility test suites include a set of test cases that cover the behavioral models associated with the tested component.

During integration testing, long interactions are more likely to expose faults than simple ones, which should be already covered by the unit testing of the component. We prioritize regression test cases according to the complexity of the interactions between the system and the target component that are triggered by the test. Thus, after executing the compatibility test suite to guarantee adequate coverage of the data values involved in the communication, we sort the remaining test cases according to the complexity of the interactions of the system with the target component.

System integrators use compatibility and prioritized test suites when updating or substituting components, whenever they have a large number of candidate components and executing regression suites is expensive. System integrators quickly skim the components to be considered by running the compatibility test suite associated with the components that are targeted by the changes, and discarding incompatible candidate components. Then, system integrators select the new component from the (small) subset of (likely) compatible components, and automatically execute the prioritized regression test suite to reveal possible integration problems. Executing the test cases in the order of priority increases the chances to reveal integration faults, and thus removes them early during regression testing. Moreover, it gives higher confidence of the correct integration of the new component, when not all test cases can be executed due to time or resource constraints.

3 Behavior Capture and Test

Behavior Capture and Test (BCT) is a dynamic analysis technique that automatically synthesizes behavioral models from execution traces [13]. Models are produced by first recording the data values exchanged between components and the sequences of invoked methods, and then automatically synthesizing boolean properties that hold on all recorded values and finite state automata that generalize the recorded interaction sequences. When components exchange complex objects, BCT recursively extracts their state attributes to obtain the data values stored inside.

BCT automatically infers IO and interaction models. IO models are boolean expressions over the values exchanged during the computation, and are inferred from traces with the Daikon engine, which builds boolean expressions by combining a finite set of operators with recorded variables [5]. All inferred boolean expressions hold for any trace. For example, Daikon can generate properties like `person.getFirstName.length`≥ 6. We slightly modified the set of operators offered in the Daikon standard release, by removing some of the operators enabled by default, since they usually generate properties that are of little use in general-purpose software, e.g., the "<<" operator can be useful in embedded applications, but seldom generates interesting properties in general-purpose software according to our experience.

Interaction models are finite state machines labeled with method invocations, and are derived with the kBehavior engine, which incrementally infers finite state models from sets of positive traces [13]. Interaction models summarize interaction sequences that can occur when a given method is executed. Interaction models always generate all traces provided as input for the inference.

BCT has been developed for Java software and uses AOP for monitoring applications, but the technology can be applied to any type of system that can be suitably monitored. For example, it can be applied to C++ programs by instrumenting executable code.

4 Compatibility Test Suite

Compatibility test suites should be small to reduce execution overhead, but should exercise a useful subset of interactions between the component of interest and the rest of the system, to maximize the chances of revealing integration problems. We meet both goals by generating a minimal subset of test cases that covers both the IO and the interaction models associated with the target component. The IO and the interaction models associated with a component are usually quite small, and can be covered with a small subset of test cases (the experiences reported in Section 6 indicate that for large test suites, models can usually be covered with less than 10% of the original test cases). Since IO and interaction models represent all interactions of the specific component with the system during test, the test cases that cover these models are good representative of component-system interactions, despite the relatively small size.

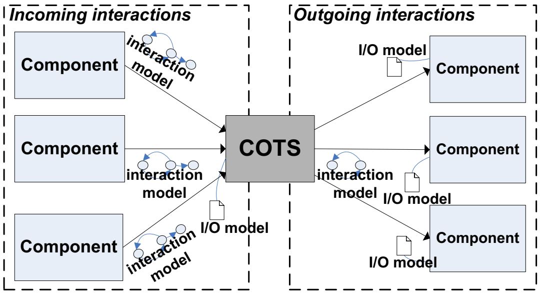


Figure 3. Behavioral models considered in the generation of a compatibility test suite.

Figure 3 shows the models that must be considered to cover all interactions between a component and the system. They include incoming and outgoing interactions. Incoming interactions correspond to requests from the system to the target component, while outgoing interactions correspond to requests from the component to the system.

Incoming interactions are described by the interaction models associated with components that access the target component, and the IO models associated with the interface of the target component. A test suite that exercises incoming interactions covers both the invocation sequences and the data values that can be issued from the system to the component.

Outgoing interactions are described by the interaction models associated with the target component and the IO models associated with the components accessed by the target component. A test suite that exercises outgoing interactions covers both invocation sequences and data values that can be issued from the component to the system.

Interaction models are finite state machines that can be covered with many criteria that range from simple state coverage, which requires each state to be traversed at least once, to complex path coverage that requires each path to be traversed at least once (with a different way of finitely reducing the possibly infinite paths) [16]. In our empirical investigations we used transition coverage, which requires each transition to be traversed at least once. Transition coverage often represents a good compromise between accuracy, i.e., number and diversity of selected behaviors, and size, i.e., number of selected test cases. In our case, this criterion guarantees the execution of each interface method of components involved in interactions.

IO models are boolean expressions that are obtained by combining a (finite) set of operators and variables. We define coverage criteria for boolean expressions inductively on the operators, and we require covering all operators that occur in the expressions [14]. We define single operator coverage by referring to boundary-value analysis [22]: each operator must be executed with values on the boundary, next to the boundary and inside the boundary of its domain. For example, the expression `person.getAge≥40` is covered by three test cases that satisfy the following constraints: `person.getAge=40` (boundary value), `person.getAge=41` (next to the boundary value) and `person.getAge>41` (inside boundary value). Some constraints may not be satisfied by the considered set of test cases. For example, the constraint `person.getAge≥40` may be derived by a set of executions that do not include value 41, and thus we cannot select a test case corresponding to `person.getAge=41`. Test designers may decide whether to add the test or ignore it. While classic criteria include also error values, i.e., values outside the domain, we do not have this option since we select a subset of test cases which satisfy the constraints by construction. Table 4 summarizes the criteria used for the main classes of operators, grouped by arity and type of operands.

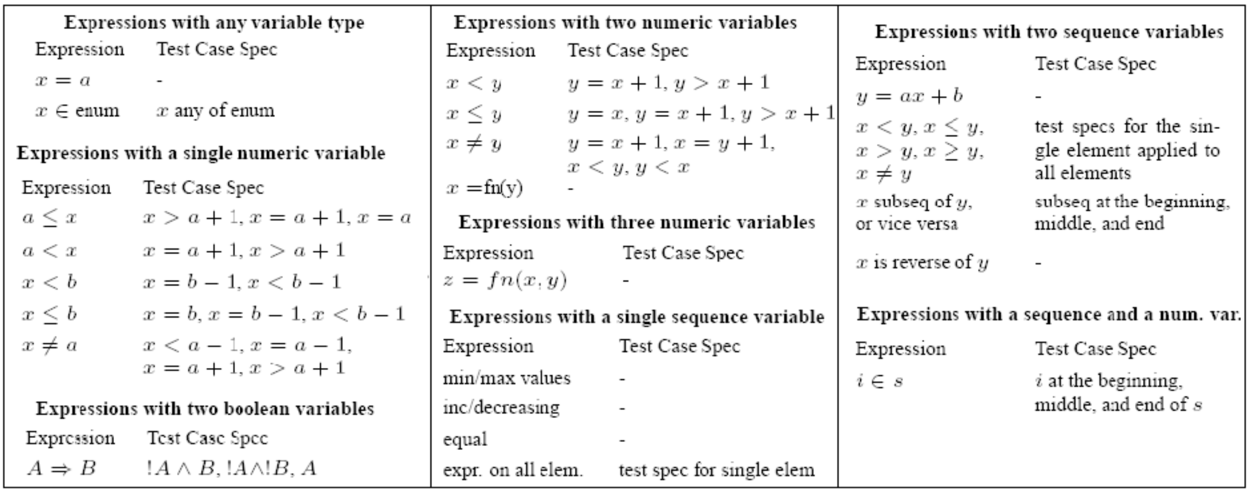
5 Test Case Prioritization

Ideally, regression test cases should be sorted according to their capability of revealing regression faults. Unfortunately, this order can be completely identified only by executing all test cases (too late to be of practical use).

We try to approximate fault revealing capability by first exercising all distinct classes of interactions between the system and the target component, i.e., we assign highest priority = ∞ to test cases of the compatibility test suite, and then privileging complex over simple interactions, i.e., priority of the remaining test cases is computed as the number of distinct interactions between the system and the target component that are covered. In particular, given a test case t, its priority index with respect to a component C is indicated with $PI_C(t)$ and is computed as

$$PI_C(t) = \sum_{fsa \in IM(C)} \sharp CT(fsa, t)$$

where $IM(C)$ is the set of interaction models associated



Expressions with any variable type

Expression	Test Case Spec
$x = a$	-
$x \in$ enum	x any of enum

Expressions with a single numeric variable

Expression	Test Case Spec
$a \leq x$	$x > a + 1, x = a + 1, x = a$
$a < x$	$x = a + 1, x > a + 1$
$x < b$	$x = b - 1, x < b - 1$
$x \leq b$	$x = b, x = b - 1, x < b - 1$
$x \neq a$	$x < a - 1, x = a - 1,$ $x = a + 1, x > a + 1$

Expressions with two boolean variables

Expression	Test Case Spec
$A \Rightarrow B$	$!A \wedge B, !A \wedge !B, A$

Expressions with two numeric variables

Expression	Test Case Spec
$x < y$	$y = x + 1, y > x + 1$
$x \leq y$	$y = x, y = x + 1, y > x + 1$
$x \neq y$	$y = x + 1, x = y + 1,$ $x < y, y < x$
$x =$fn(y)	-

Expressions with three numeric variables

Expression	Test Case Spec
$z = fn(x, y)$	-

Expressions with a single sequence variable

Expression	Test Case Spec
min/max values	-
inc/decreasing	-
equal	-
expr. on all elem.	test spec for single elem

Expressions with two sequence variables

Expression	Test Case Spec
$y = ax + b$	-
$x < y, x \leq y,$ $x > y, x \geq y,$ $x \neq y$	test specs for the single element applied to all elements
x subseq of y, or vice versa	subseq at the beginning, middle, and end
x is reverse of y	-

Expressions with a sequence and a num. var.

Expression	Test Case Spec
$i \in s$	i at the beginning, middle, and end of s

legend: x, y and z indicate names of variables or sequences; a, b and c indicate constants; fn indicates any function; A and B indicate boolean expressions, and $-$ indicates that any test case that covers the variables in the expression covers also the expression.

Figure 4. Criteria for covering the main operands that occur in IO models.

with component C (as shown in Figure 3), and $\sharp CT(fsa, t)$ is the number of transitions in fsa covered by the execution of t.

Exercising all classes of interactions aims to reveal common integration faults, while privileging complex interactions addresses the subtle faults that occur in rare and long interaction patterns.

Note that priority considers only the interactions that involve the target component: a test case that triggers many interactions within the system, but triggers only one interaction with the target component is given priority 1, while a test case that triggers few interactions within the system, but calls many methods of the target component (e.g., 5) is assigned high priority (e.g., 5).

6 Early Validation Results

We evaluated the effectiveness of compatibility test suites by measuring the size of the suites and the number of revealed integration faults and incompatible components. The size of the compatibility test suites represents the savings in terms of time with respect to the execution of all test cases for all candidate components. The number of revealed integration faults and incompatible components measures the effectiveness of compatibility test suites by indicating the amount of candidate components that can be quickly excluded before a thorough evaluation. We evaluated compatibility test suites considering both common and actual integration faults.

We evaluated the effectiveness of prioritization schemas by measuring the fault revealing rate when executing regression test suites according to the prioritization schema, and by computing the amount of prioritized test suites that must be executed to reveal all known integration faults.

We do not consider the cost of generating the compatibility suite and the prioritization schema, since the generation process is fully automated and can be completed before starting the regression activities, thus this does not contribute to the regression schedule.

The results of the experiences conducted so far are positive: in most of the considered cases, compatibility test suites were very small (less than 7% of the original test suites), and could reveal most incompatibilities across several components (74% of the faults revealed by the whole test suite); moreover prioritization schemas could reveal most faults very early (96% of faults were revealed while executing less than 10% of test cases). The amount of empirical investigations conducted so far does not allow us to generalize the results yet, but in the following we report the results of two representative experiences to provide evidence of the effectiveness of the technique proposed in this paper.

Evaluating Compatibility Suites wrt Common Faults To measure the ability of revealing common integration faults, we proceeded as follows: We selected applications provided with test cases; We generated incompatibilities by seeding common integration faults with mutant operators; and we measured the effectiveness of the compatibility suite as the ration between the number of faults revealed by compatibility testing and the number of faults revealed by the original test suite. We discuss the experiences with real

Mutant operator	Configs	Traversing Suite (29 TC)	Compatibility Suite (6 TC)
DirVarRepPar	5	4 (80%)	4 (80%)
DirVarRepLocal	3	2 (67%)	2 (67%)
DirVarRepExt	5	4 (80%)	4 (80%)
DirVarRepCon	5	4 (80%)	4 (80%)
DirVarRepReq	5	4 (80%)	4 (80%)
IndVarRepPar	5	4 (80%)	4 (80%)
IndVarRepLocal	5	2 (40%)	2 (40%)
IndVarRepExt	5	3 (60%)	3 (60%)
IndVarRepReq	5	5 (100%)	5 (100%)
DirVarIncDec	5	5 (100%)	5 (100%)
IndVarIncDec	5	5 (100%)	5 (100%)
DirVarAriNeg	5	4 (80%)	4 (80%)
IndVarAriNeg	5	5 (100%)	5 (100%)
RetStatDel	2	1 (50%)	1 (50%)
RetStatRep	2	1 (50%)	1 (50%)
ArgRepReq	5	3 (60%)	3 (60%)
ArgStcAli	2	2 (100%)	2 (100%)
ArgAriNeg	2	1 (50%)	1 (50%)
ArgIncDec	5	2 (50%)	2 (50%)
FunCallDel	5	4 (80%)	4 (80%)
NullAtt	5	2 (40%)	2 (40%)
NullSet	5	5 (100%)	5 (100%)

Legend
Mutant Operator indicates the mutant operator used to generate the incompatible configurations (the names are taken from [3, 6]), *Configs* is the number of configurations generated by the operator, *Traversing Suite* indicates the number and percentage of configurations identified as incompatible by executing the traversing test suite of 29 test cases, *Compatibility Suite* indicates the number and percentage of configurations identified as incompatible by executing the compatibility test suite of 6 test cases.

Table 1. Effectiveness of compatibility testing for component `SignatureAlgorithm`.

faults in the next subsection.

In this subsection, we present the results obtained with the Apache XML Security application (version 1.0.5$D2$ downloaded from the SIR repository[1]) that is provided with 84 test cases. We focused on component `SignatureAlgorithm`, and we generated a compatibility test suite by covering the behavioral models obtained by executing the 84 test cases. Only 29 out of the 84 test cases traverse the behavioral models for component `SignatureAlgorithm` (we call this set of test cases the *traversing suite*), and only 6 are enough to cover the models.

We seeded common integration faults by mutating the original program. We referred to mutant operators for integration problems defined by Delamaro, Maldonado, Mathur and Ghosh [3, 6]. We extended the mutant operators that replace a target variable with a constant, by considering also values `null` and `""` for objects and strings. The operators *NullAtt* and *SetAtt* that were original introduced by Ghosh and Mathur in [6] to address modification of parameters' attributes with `null` and constant values are particularly useful to deal with complex objects passed between components. The mutants used in our empirical study are reported in the first column of Table 1.

We generated 96 incompatible components for `SignatureAlgorithm` by randomly mutating: methods that interact with `SignatureAlgorithm`, methods implemented by `SignatureAlgorithm` and methods invoked by `SignatureAlgorithm`. We applied each operator up to five times, and no more than two times to the same program location. We identified and removed equivalent mutants, i.e., programs that cannot be distinguished from the original one.

Table 1 reports the results of executing all 96 configurations obtained by substituting the original component with each of its mutants both with the traversing test suite (29 test cases) and with the compatibility suite (6 test cases). The table shows that the compatibility test suite, which includes only 21% of the traversing test suite, is as effective as the original test suite.

Both the traversing and the compatibility suites do not identify all incompatible configurations. This is due to the original test suite that does not include test cases for 2 constructors, 6 methods, 4 special cases and the exceptions, resulting in 44.30% statement coverage[2] and in 50% of conditions coverage. All unrevealed incompatible components are obtained by mutating parts of the code not covered by the original suite.

Evaluating Compatibility Suites wrt Actual Faults To measure the ability of revealing actual faults, we proceeded as follows: We identified component-based applications provided with test suites and (many) syntactically-compatible alternatives for at least a component. We executed the whole test suite for all configurations obtained by substituting the alternative components to the target component, to identify integration faults. We derived compatibility test suites for the target component. We executed the compatibility test suite and we measured size, execution time and number of identified regression faults, to substantiate the results illustrated in the former subsection.

In this subsection, we illustrate the results obtained with the Ginipad Java editor [7]. We applied our technique to version 2.0.3, which include 316 Java classes. We considered 182 test cases generated with the category partition method [15]. The time required to execute the whole suite is about 10 hours. We focused on the new component `textArea` developed for Ginipad version 2.0.3, and we identified 14 additional components syntactically compatible with `textArea`. Thus, we considered a total of 15

[1] The SIR (Software-artifact Infrastructure Repository, http://esquared.unl.edu/sir) is a publicly available repository where several applications in different versions and languages are maintained to provide a common platform to run experiments and case studies.

[2] The original suite covers 42.50% of the statements. We slightly modified the `update` method of `SignatureAlgorithm` to execute a part of the code to which we can apply a few mutant operators that would not be applicable elsewhere.

Id	Component Name	Web Site	No. faults	Fault Type
c1	SyntaxHighlighter	www.cs.bris.ac.uk/Teaching/Resources/	1	faulty save (C)
c2	JavaEditorKit	java.sun.com	4	faulty undo (M), faulty color (L), faulty typing (C), faulty save (C)
c3	RTextArea	rtext.sourceforge.net/	1	faulty save (C)
c4	NonWrapping JTextPane	theory.lcs.mit.edu/m̃jt/IOA_Toolkit/	1	faulty save (C)
c5	RTFTextPane	java.sun.com/	3	faulty compile (C), delete document(C), no save doc (C)
c6	LineNumberedPaper	www.esus.com	1	faulty save (C)
c7	NumberedEditorKit	www.developer.com	1	faulty save (C)
c8	GinipadTextArea	www.mokabyte.it/ginipad/english.htm	2	faulty undo (M) faulty highlighting (S)
c9	RSyntaxTextArea	rtext.sourceforge.net/	1	faulty save (C)
c10	MultiLineLabel	www.koders.com	1	faulty highlighting of compilation problems (M)
c11	ColorTextInTextPane	forum.java.sun.com	1	no highlighting (S)
c12	ConsoleOutput	www.rosuda.org	1	faulty save (C)
c13	ColorNumberedPane	c12 + c11	1	no highlighting (S)
c14	CustomJTextPane	forum.java.sun.com	1	faulty save (C)
c15	ScaledTextPane	www.developer.com	2	faulty save (C), faulty zoom (L)

Legend
Id is the identifier of the configuration, *Component Name* is the name of the component, *Web Site* is the URL from which we downloaded the component, *No. faults* is the number of integration faults reveled by the original test suite and *Fault Type* is a short description of the type of the integration faults and their severity (severity can be C=Critical, S=Severe, M=Moderate and L=Cosmetic).

Table 2. Configurations used in the experiments.

alternative configurations listed in Table 2.

We executed the test suite and we identified 22 integration faults. We considered only integration faults, i.e., faults that depend on the integration of two or more components, and cannot be identified while testing components in isolation. Not all faults prevent successful integration, but they provide useful information to quickly remove bad candidates from the list of candidate components.

We automatically generated a compatibility test suite for component `textArea`. The suite includes 13 test cases. Its size is thus 7% of the size of the whole test suite (182 test cases). We executed the compatibility test suite for the 15 configurations. The execution of the compatibility suite for all configurations required about one working day, while the execution of the whole regression suite for all configurations requires about three working weeks on a single machine. The execution time of the compatibility suite is thus 7% of the execution time of the whole suite. The compatibility test revealed 17 out of the 22 actual integration faults. Its effectiveness is thus 77%.

Table 3 shows the detailed results. The first three columns of the table (Experimental Setting) indicate the details of the considered configurations, the following columns (Comp. Suite) report the number and amount of faults revealed by the compatibility suite, and the last columns (Avg. Rev. Rate) indicate the average number of faults that can be revealed by a random set of 13 test cases and their rate. It is computed as the mathematical average over the revealed faults of all combinations of 13 test cases of the original suite. We can notice that the compatibility suite identifies completely 12 faulty configurations and partially two additional configurations. Thus it completely misses only one faulty configuration out of 15. The difference between faults revealed by the compatibility suite and the average number of faults revealed by sets of test cases of the same size of the compatibility suite is a good index of quality of the technique, and is extremely high.

Information about faults can be used in many ways. A simple approach consists of considering only configurations that do not fail during testing. In our example, executing all test cases would discard all choices, while the compatibility test cases would not discard component $c10$, thus restricting the needs of executing the whole test suite for one configuration only. A careful approach would discard configurations based on fault severity, and would drop only components that present critical integration faults or an overall high number of faults. In our example, executing all test cases would lead to consider components $c8$, $c10$, $c11$ and $c13$. The compatibility test suite leads to the same result, even if it misses a few faults. In general compatibility test suites may select a larger set of candidates, since compatibility suites can miss some critical faults, but they never discard interesting candidates.

Evaluating Priority Schemas We evaluated the effectiveness of priority schemas by measuring the amount of test cases that must be executed according to the priority schema to reveal the known faults. We compared the result with the mathematical average of test cases that must be executed to reveal all known faults by considering all possible permutations of executions.

Table 4 shows the results obtained with the 15 configurations of the Ginipad version 2.0.3 listed in Table 2. In 14 out of the 15 considered configurations, we revealed all known faults after executing less than 10% of the whole suite. Only one configuration required executing more than 47% of the test cases. The mathematical averages presented in the table show that without the priority schema the effectiveness of the execution drops dramatically.

Experimental Setting			Comp. Suite		Avg Rev. Rate	
Conf	No Faults	No Rev TC	No rev faults	Rev rate	No rev faults	Rev rate
c1	1	2	1	100%	0.14	14%
c2	4	13	2	50%	0.87	21.76%
c3	1	2	1	100%	0.14	14%
c4	1	2	1	100%	0.14	14%
c5	3	28	1	33.33%	1.98	66%
c6	1	2	1	100%	0.14	14%
c7	1	2	1	100%	0.14	14%
c8	2	29	2	100%	1.51	50%
c9	1	2	1	100%	0.14	14%
c10	1	4	0	0%	0.26	26%
c11	1	95	1	100%	0.99	99%
c12	1	2	1	100%	0.14	14%
c13	1	95	1	100%	0.99	99%
c14	1	2	1	100%	0.14	14%
c15	2	54	2	100%	1.06	53%

Legend
Conf is the identifier of the configuration, *No Faults* is the number of faults revealed by executing the original test suite (182 test cases), *No Rev TC* is the number of fault revealing test cases that are part of the original test suite, *Comp Suite* indicates the number of integration faults revealed by compatibility suite (13 test cases) and the corresponding rate, *Avg. Rev Rate* indicates the average number of faults revealed by randomly selecting 13 test cases and the corresponding rate.

Table 3. Compatibility vs randomly selected test suites.

Config	No Faults	Prio		Avg	
		No rev faults	Rev rate	No rev faults	Rev rate
C1	1	13	7.14%	61	33.52%
C2	4	85	47%	123.75	67.99%
C3	1	13	7.14%	61	33.52%
C4	1	13	7.14%	61	33.52%
C5	3	14	7.69%	14.14	7.77%
C6	1	13	7.14%	61	33.52%
C7	1	13	7.14%	61	33.52%
C8	2	9	4.95%	32.95	18.1%
C9	1	13	7.14%	61	33.52%
C10	1	17	9.34%	36.6	20.1%
C11	1	1	0.55%	1.91	1.05%
C12	1	13	7.14%	61	33.52%
C13	1	1	0.55%	1.91	1.05%
C14	1	13	7.14%	61	33.52%
C15	2	13	7.14%	61	33.52%

Legend
Config is the identifier of the configuration, *No Faults* is the number of faults revealed by executing the original test suite (182 test cases), *Prio* indicates the number of test cases that must be executed according to the priority schema to reveal all faults, and *Avg* indicates the average number of test cases that must be executed to reveal all faults.

Table 4. Priority vs randomly selected test suites.

7 Related Work

The technique proposed in this paper relates to the research in regression testing. In this section, we summarize the state of research in the field, and we relate our work to previous work, highlighting original contributions, complementarities and synergies. We distinguish four main classes of research contributions: classical regression testing techniques, which address the problem of selecting test cases while software evolves; regression testing techniques for COTS components, which address the problems of selecting test cases when neither source code nor complete specifications are available; regression testing techniques based on dynamic analysis, which derive test cases from information about software behavior; and test case prioritization techniques, which sort test cases to improve fault revealing rate.

Classical Regression Testing. Classical regression testing techniques select test cases that need to be re-executed by comparing different versions of the application under test, and extracting test cases that traverse the modified parts. Changes in successive versions are identified by comparing either source code or specifications.

Techniques based on source code identify differences by comparing abstract models of the code, usually control or data flow models [2, 8, 18, 11]. The effectiveness of these approaches depends on the pervasiveness of the changes: They work well when changes affect small parts of the control or data flow graph, but do not produce useful results when changes affect directly or indirectly large portions of the models.

Techniques based on specifications identify differences by comparing specifications [16]. The effectiveness of these techniques depends on the consistency between specifications and code: They work well when specifications are complete, up to date and very detailed, but they do not produce good results when specifications do not perfectly match the code.

Classic regression testing techniques do not apply well to COTS components, since COTS components are often provided without source code and with incomplete specifications. Techniques for regression testing of COTS components inherit basic ideas, and take advantage from the experimental results on the relation between test cases, type of software, and nature of changes [17, 8].

The technique proposed in this paper is grounded on the idea of deriving information from software characteristics, but overcomes the limitations that derive from lack of code and complete specifications for COTS components, by focusing on behavioral models that are automatically extracted from the execution of test cases on previous versions of the software, and by computing priorities according to the exercised behaviors and not the specific changes.

Regression Testing of COTS Components. Most research on regression testing for COTS components extends classical regression techniques to the binary code. These techniques identify changes at either bytecode or binary code level, and select test cases that cover the modified parts [25, 20].

The effectiveness of these techniques depends on the correspondence between changes in the bytecode or the binary code and the actual changes in the component. When components are produced with different development environments or even simply with different compiling options, as often happens, differences in the bytecode become pervasive and may not reflect the changes in the source programs.

Other techniques require additional information about the component [10]. This may become common practice in the future, but does not apply well to most current situations, where COTS components are rarely provided with more than executable code and partial specifications.

Our technique avoids the problems that derive from comparing binary code by basing test case selection and prioritization on behavioral models automatically derived from executions. It thus complements techniques based on comparison of binary code by managing those changes that cannot be effectively addressed by comparing compiled artifacts, i.e., non-trivial modifications of the same components or replacements of components with others from different vendors.

Regression Testing from Dynamic Analysis. Dynamic analysis techniques automatically derive behavioral properties from program execution. They require only the executable code, and not source code or complete specifications. The behavioral properties derived with dynamic analysis are useful sources of information for generating, selecting and prioritizing test cases.

Dynamic analysis is raising a large interest in the research community [5, 24, 13]. However, its use for regression testing has not been widely investigated yet. Dynamic analysis for regression testing has been investigated mainly by Harden, Meller and Ernst [9] and by Xie and Notkin [23]. Harder, Meller and Ernst use incremental refinements of dynamically inferred information (very similar to our IO models), to improve test suites, while Xie and Notkin extract dynamic information about invocation sequences to automatically identify the causes of deviations revealed during regression testing.

In our study, we address the new problem of generating compatibility test suites and test case prioritization schemas, addressing a problem similar to the one considered by Harden, Meller and Ernst, who derive regression tests, but different from the one addressed by Xie and Notkin, who look for fault causes. Harder, Meller and Ernst consider only information on the relation of the values exchanged with the component; Xie and Notkin consider only invocation sequences. We use information about both the value exchanged between system and components and the sequences of invocations of component services.

Test Case Prioritization. Recently, a great deal of research has investigated test case prioritization schemas. Some approaches prioritize test cases according to execution age to expose possible latent faults and assure uniform testing of code: Kim and Porter suggest postponing the execution of recently executed test cases [12], while Elbaum, Malishevsky and Rothermel postpone the execution of test cases that refer to recently executed code [4]. Other approaches favor test cases that revealed faults in previous versions, based on the observation that faults are not uniformly distributed, but tend to accumulate in small portions of the code [4]. Yet other approaches prioritize test cases according to the amount of covered code, to cover a large amount of entities with a small set of initial test cases [19].

Prioritization schemas based on execution age and fault distribution work well across different versions of the same component, but are not justified when replacing components that implement the same interface but are provided by different vendors, as frequently happen for COTS components. Prioritization schemas based on code coverage suffer from lack of source code of COTS components.

Our prioritizing schema is based on a mixed strategy that combines coverage of interactions between system and components with the coverage of complex execution sequences, in the novel setting of dynamically inferred models. The results presented in the paper substantiate the expectation that the two criteria work well together, since early covering interactions captures common integration faults. Whereas privileging complex interaction sequences reveals rare subtle faults.

8 Conclusions

Our research focuses on the problem of regression testing of systems that integrate components available without easy access to both source code and complete specifications, as in the case of COTS components. This paper addresses two issues: the problem of quickly identifying and discarding components that are syntactically compatible with the interface specifications, but badly integrate in the system, and the problem of automatically generating efficient regression test suites for component integration. The problems become important when regression test suites are time and resource consuming, and are hot when replacing COTS components with products from different vendors.

Both classic regression testing techniques and techniques developed specifically for COTS components address these problems only partially, and are particular inefficient when COTS components are substituted with products provided by other vendors.

This paper proposes a technique for generating compatibility test suites and prioritized regression suites. The technique is based on models that represent inter-component behavior. Such models are automatically derived while testing the previous versions of a system, and do not depend on either source code or component specifications.

Compatibility test suites are small (about 7% of the size of the test suite, in the empirical investigations conducted so far), and can quickly identify many incompatible components within a set of syntactically-compatible candidates (they revealed about 77% of the integration faults, in the empirical investigations conducted so far). Thus they pair high revealing rate with important time and effort saving.

With prioritized regression suites, many faults are discovered after executing a small amount of high-priority test cases (less than 10% of the highest priority test cases revealed all faults for all considered configurations except one, in the empirical investigations conducted so far). Thus prioritized test suites can improve the test-debug cycle, and increase the efficacy of testing when time or resource constraint limit the number of test cases that can be executed.

The promising results presented in this paper indicate important research directions: (1) investigating the use of additional dynamic models (e.g., temporal properties [24]) to generate regression suites, (2) adding empirical experiences to further validate the results, and (3) studying the relation between the effectiveness of prioritized test suites and the nature of the test suite, such as test suite granularity and grouping [17].

References

[1] S. Beydeda and V. Gruhn. *Testing Commercial-off-the-Shelf Components and Systems*. Springer, 2005.

[2] J. Bible, G. Rothermel, and D. Rosenblum. A comparative study of course- and fine-grained safe regression test-selection techniques. *ACM Transactions on Software Engineering and Methodology*, 10(2):149–183, 2001.

[3] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur. Interface mutation: An approach for integration testing. *IEEE Transactions on Software Engineering*, 27(3):228–247, March 2001.

[4] S. Elbaum, A. Malishevsky, and G. Rothermel. Test case prioritization: a family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2):159–182, 2002.

[5] 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, 2001.

[6] S. Ghosh and A. Mathur. Interface mutation to assess the adequacy of tests for components and systems. In *proceedings of Technology of Object-Oriented Languages and Systems*, pages 37–46. IEEE Computer Society, 2000.

[7] Ginipad. http://www.mokabyte.it/ginipad/english.htm.

[8] T. Graves, M. Harrold, M. Kim, Y. Porter, and G. Rothermel. An empirical study of regression test selection techniques. *ACM Transactions on Software Engineering and Methodology*, 10(2):184–208, 2001.

[9] M. Harder, J. Mellen, and M. D. Ernst. Improving test suites via operational abstraction. In *proceedings of the International Conference on Software Engineering*. IEEE, 2003.

[10] M. Harrold, A. Orso, D. Rosenblum, G. Rothermel, M. Soffa, and H. Do. Using component metacontents to support the regression testing of component-based software. In *proceedings of the IEEE International Conference on Software Maintenance*, 2001.

[11] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for java software. In *proceedigns of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 2001.

[12] J.-M. Kim and A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *proceedings of the International Conference on Software Engineering*, pages 119–129. ACM, 2002.

[13] L. Mariani and M. Pezzè. Behavior capture and test: Automated analysis of component integration. In *proceedings of the IEEE International Conference on Engineering of Complex Computer Systems*, 2005.

[14] L. Mariani, M. Pezzè, and D. Willmor. Generation of selft-test components. In *proceedings of the International Workshop on Integration of Testing Methodologies*, volume 3236 of *LNCS*, pages 337–350. Springer, 2004.

[15] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, June 1988.

[16] M. Pezzè and M. Young. *Software Test and Analysis: Process, Principles and Techniques*. John Wiley & Sons, 2007.

[17] G. Rothermel, S. Elbaum, A. Malishevsky, P. Kallakuri, and X. Qiu. On test suite composition and cost-effectiveness regression testing. *ACM Transactions on Software Engineering and Methodology*, 13(3):277–331, July 2004.

[18] G. Rothermel and M. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, 1996.

[19] G. Rothermel, R. Untch, C. Chu, and M. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on software engineering*, 27(10):929–948, 2001.

[20] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *International Symposium on Software Testing and Analysis*. ACM Press, 2002.

[21] M. Vigder and J. Dean. Building maintainable COTS based systems. In *International Conference on Software Maintenance*. IEEE, 1998.

[22] L. White and E. Cohen. A domain strategy for computer program testing. *IEEE Transactions on Software Engineering*, 17:703–711, 1991.

[23] T. Xie and D. Notkin. Checking inside the black box: Regression testing by comparing value spectra. *IEEE Transactions on Software Engineering*, 31(10):869–883, 2005.

[24] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: Mining temporal API rules from imperfect traces. In *proceedings of the International Conference on Software Engineering*. ACM, 2006.

[25] J. Zheng, B. Robinson, L. Williams, and K. Smiley. Applying regression test selection for cots-based applications. In *proceedings of the International Conference on Software Engineering*. ACM, 2006.