Automated, Contract-based User Testing of Commercial-Off-The-Shelf Components

Lionel C. Briand\textsuperscript{\textcopyright} Yvan Labiche\textsuperscript{\textcopyright} Michał M. Sówka\textsuperscript{\textcopyright}
\textsuperscript{\textcopyright} Carleton University, Department SCE
Software Quality Engineering Laboratory
1125 Colonel By Drive, Ottawa, ON K1S5B6, Canada
\{briand, labiche, msowka\}@sce.carleton.ca
\textsuperscript{\textcopyright} Simula Research Laboratory
Department of Software Engineering
P.O.Box 134, 1325 Lysaker, Norway
briand@simula.no

ABSTRACT
Commercial-off-the-Shelf (COTS) components provide a means to construct software (component-based) systems in reduced time and cost. In a COTS component software market there exist component vendors (original developers of the component) and component users (developers of the component-based systems). The former provide the component to the user without source code or design documentation, and as a result it is difficult for the latter to adequately test the component when deployed in their system. In this article we propose a framework that clarifies the roles and responsibilities of both parties so that the user can adequately test the component in a deployment environment and the vendor does not need to release proprietary details. Then, based on this framework we combine and adapt two specification-based testing techniques and describe (and implement) a method for the automated generation of adequate test sets. An evaluation of our approach on a case study demonstrates that it is possible to automatically generate cost effective test sequences and that these test sequences are effective at detecting complex errors.

Categories and Subject Descriptors
D.2.4 [Software Engineering]: Software/Program Verification--Validation, Reliability

General Terms

Keywords
COTS, Component, UML, Adequacy Criteria.

1. INTRODUCTION
The concept of software components, software modules, or more generally software reuse, is not new and dates back to the late 1970s. The structure-oriented software development methodology of the time often limited the reuse of the software modules to within the scope of a project. More recently, object-oriented development methodologies have facilitated packaging of sub-systems into software modules for reuse across development teams and projects. Modern software models such as ActiveX, CORBA, and Enterprise JavaBeans (EJB) take a further step in promoting code reuse.

A software component, adopting the definition provided in [28], is a “unit of composition with contractually specified interfaces and context dependencies only. A software component can be deployed independently and is subject to composition by third parties”. Composition is the assembly of one or more software components into composite software systems. Contractually specified interfaces are either of the two following types: provides representing the functionality that the component provides; or requires representing the functionality that the component requires (e.g., from other components). By specifying strict interfaces through which they are deployed and exercised, modern component models\textsuperscript{1} allow component reuse across organizations and create a Commercial-off-the-Shelf (COTS) component market which consists of component vendors and component users. Component users purchase the vendors’ components for use in their component-based software systems deployed on a specific deployment platform.

The deployment platform typically consists of computer hardware and network, operating system, system services (e.g., email address lookup), and a component container providing (among other things) the interfaces required for component deployment. Component containers are implemented strictly according to the specification of a particular component model (e.g., EJB).

Component-based software engineering has many advantages [28]: the most obvious ones are reusability and reduced costs. But it also poses new challenges to component users that are non-issues in traditional software development practices [14]. One of these challenges is the evaluation (verification and validation) of a component by the user on a specific deployment platform, for instance by means of testing techniques. Four main reasons are:

- Heterogeneity of deployment platforms: The user may not necessarily use a platform on which the component has been evaluated by the vendor.

- Limited Test model: Because no source code or design information is usually provided, the information to develop a test model is often limited to textual documentation and an API. Controllability and observability are then limited [5].

- Difficult definition of testing adequacy criteria: Although component interfaces are described according to specific

\textsuperscript{1} In this article, unless accompanied by contractually specified (possibly standard-based) provides and requires interfaces, class packages and libraries are not considered software components.
component models, they often do not provide much information that can be used to define functional testing adequacy criteria. The component user has to mostly rely on the textual documentation of the component. For instance, the often used adequacy criteria all-method and all-exception have been shown to be insufficient [15].

- **Generality of components**: Components are often designed for broad applicability, whereas users often require only a subset of the provided functionalities. Users cannot afford to test the entire component and need techniques to select component partitions to be tested. This need has been illustrated for instance by the formal component model provided in [25].

The main contributions of our work include: (i) The definition of test adequacy criteria in a context where no source code and design information is available (Section 4); (ii) An automation framework, including an algorithm for the construction of an adequate but minimal set of test sequences for those criteria (Section 5). The article starts with a review of related work (Section 2) and discusses how the component vendor could facilitate the deployment testing of the user without revealing proprietary details (Section 3). Our test technique is then detailed in Sections 4 and 5. It is evaluated in Section 6 on an actual component and conclusions are drawn in Section 7.

2. RELATED WORKS

A variety of functional testing techniques can be considered for component testing. First of all, some that rely only on a textual documentation come to mind (e.g., Category Partition [23]). However, we describe in Sections 2.1 to 2.3 three other techniques that can be used by the component user and are amenable to the automatic generation of adequate test sets. Section 2.4 discusses a strategy that component vendors can use to conveniently provide (testing) artifacts to the component user.

2.1 CSPE Constraint-based Testing

Constraints on Succeeding and Preceding Events (CSPE) is a testing technique originally used for the functional testing of concurrent programs [7], and more recently used for class unit testing [10]. Following the CSPE technique, CSPE constraints on sequences of invocation of pairs of methods are first derived from the specification. We denote them as 3-tuples (preceding method, succeeding method, predicate) indicating that a call to the succeeding method can be performed after a call to the preceding method when the predicate is true. CSPE constraints, and more specifically predicates, can be derived from method preconditions and postconditions. The postcondition of the preceding method either implies, contradicts, or partially implies the precondition of the succeeding method, resulting in four types of constraints: Always Valid, Never Valid, Possibly Valid, and Possibly Invalid. In the first two cases, the 3-tuple predicate is always True or always False. The latter two are complementary: Invoking the succeeding method immediately after the preceding method is valid (resp. invalid) only when a specific predicate is true (resp. false). A possibly valid constraint occurs when the postcondition of the preceding method does not imply the precondition of the succeeding method, but the conjunction of the preceding method’s postcondition and the predicate implies the precondition of the succeeding method.

CSPE constraints can be used to generate method test sequences according to four CSPE constraint coverage criteria [10]. Always Valid Coverage (A) requires that all Always Valid constraints be covered at least once. Always/Possibly Valid Coverage (AP) includes both Always Valid and Possibly Valid constraints. Never Valid/Possibly Invalid Coverage (NP) requires that all Never Valid and Possibly Invalid constraints be covered at least once. Finally, Always/Possibly Valid/Never Valid/Possibly Invalid Coverage (ANP) requires that all constraints be covered at least once. It is also possible to add a Never Valid Coverage criterion (N) that requires covering only the Never Valid constraints. It is important to note that Possibly Valid and Possibly Invalid constraints conceptually represent only one constraint since the predicates involved in the latter are the negations of predicates involved in the former.

Automatically constructing adequate test sets for these criteria is possible, as suggested in [10], by following the heuristic approach presented in [17]: It consists in building a tree where nodes are the CSPE constraints to be covered according to the selected criterion and arcs represent the concatenation of the CSPE constraints into test sequences: concatenating constraints \((m_1, m_2, \text{true})\) and \((m_2, m_3, \text{true})\) for instance amounts to executing methods \(m_1, m_2\) and \(m_3\) in sequence. Tree paths then form an adequate test for the selected criterion. The tree is grown breadth first, appending method sequence constraints from the set of constraints to be covered. Growth is terminated once all of the constraints have been covered in the tree, such that each constraint is covered at least once. The construction of the tree path also stops when a Never Valid or Possibly Invalid constraint is reached as this would result in an exception being thrown, reporting the incorrect use of the succeeding method. The tree is then pruned to reduce its size while still satisfying the criterion, resulting (because of the breadth first construction) in a test set composed of a large number of short test cases (tree paths).

2.2 Testing Logical Expressions

A number of coverage criteria have been defined for logical expressions. We use them when defining our test criteria in Section 4.2 in cases where complex predicates are involved in CSPE constraints: predicate coverage, combinatorial coverage, implicant coverage, prime implicant coverage [1, 5].

**Predicate Coverage** (PC) requires that the selected tuples of predicate clauses test the predicate twice, once such that it evaluates to true and once for false. **Combinatorial Coverage** (CoC) requires that all possible combinations of clauses’ truth values be exercised\(^2\). **Implicant Coverage** (IC) requires that the selected tuples test the predicate such that each implicant is true at least once, where implicants are disjuncts in the Disjunctive Normal Form of the predicate and its negation. **Prime Implicant Coverage** (PIC) requires that the selected tuples test the predicate such that, for the predicate and its negation, each implicant is set to true while all other implicants are false. The reader interested in more formal definitions is referred to [1, 5].

PC is a weak criterion (only two test cases are required), and CoC is an exhaustive criterion. The other two criteria, IC and PIC, are intermediate criteria in terms of cost (i.e., number of test cases),

\(^2\) A clause is a Boolean variable, the negation of a Boolean variable, or, in the context of OCL, an OCL constraint, i.e., an OCL expression that evaluates to true or false.
2.3 Statechart-based Component Testing

Following the strategy reported in [3], components used in a component-based system are assumed to be described with state machines. Those state machines are used to build a control flow graph of the overall behavior of all the components. The component user can then apply traditional control and data flow testing techniques for devising test cases.

It is worth mentioning that the level of details in the components’ state machines can have an important impact on what the user can achieve. If the state based behaviors are specified with too much detail, the control flow graph will likely be very large and difficult to use for devising test cases (structural testing techniques, for instance, based on control flow graphs are not known to scale up very well). Note also that too many details would likely reveal vendor proprietary information, which contradicts one of our working hypotheses. If on the other hand the behavior is provided at a high level of abstraction, state machines will likely not be very useful to the user to devise interesting test cases. If the component is large and complex, the statechart is in any case likely to be unwieldy and difficult to use, especially for users having no detailed design information about the component. As a result of the above issues, this paper will not make use of state machine representations to test the components’ behavior.

2.4 Component Metadata to Support Testing

A standard was suggested in [8, 22] as an extensible way for the component vendor to provide varying kinds of artifacts to be used by the component user, which are referred to as metadata. It is argued that any software engineering artifact used in component based development (e.g., testing artifacts) can be a metadatum for a given component as long as 1) the component vendor is involved in its production, 2) it is packaged with the component in a standard way, and 3) it can be processed by automated development tools and environments. Metadata describe either static (they never change) or dynamic (they may change at runtime) aspects of the component. Examples of static metadata include deployment information, textual descriptions, and version information. Structural coverage achieved when executing component methods is an example of dynamic metadata [22].

3. OVERVIEW

Figure 1 is a UML activity diagram depicting an overview of our COTS component testing strategy. It illustrates the roles and responsibilities of the component vendor and user (the two horizontal swim lanes of the diagram) in terms of the information that is generated and used for component testing purposes. Recall that in a UML activity diagram [21] rounded boxes denote activities, squares boxes denote objects (data, documents, or artifacts in our case), plain arrows denote control flow, and dashed arrows dependencies between activities and objects.

More specifically, the vendor (upper swim lane) generates CSPE constraints for the component interface methods (activity 1), for instance from design documents of the component. The vendor is also responsible for implementing CSPE probes (activity 2), that is, built-in methods that the user can use to increase controllability and observability during component testing (e.g., to put the component in a specific state).

On the other hand (lower swim lane), the first task of the user (activity 3) is to identify the component functionalities that are required by the component-based system being developed (e.g., from the design documentation): the user does not necessarily need to test the whole component. Depending on constraints such as budget, the user also selects a CSPE-based testing criterion among the ones we define in Section 4 (activity 4). Once activities 1 to 4 have been completed (the vertical bar denotes a join), the user can proceed with the generation of an adequate test set (automated as described in Section 5), given the component functionalities under test and the selected criterion (activity 5). Then the test scaffolding (drivers, stubs, oracles) can be produced and the test cases executed (activities 6 to 7).

Applying the approach proposed in Section 4, which is based on CSPE constraints, requires that some component metadata be available. First, to derive test sequences, we will need the component to provide us with the CSPE constraints themselves (static metadata). Note that clauses in predicates should not and need not be provided in any detail: symbols can be used as placeholders for clauses. That way, no (proprietary) details are provided to the user who can still use CSPE constraints to generate test sequences. (Note however that in the rest of the article we will use complete clauses instead of symbols to ease comprehension.) Required dynamic metadata include whether a method pre or post-condition holds at run-time and is used as an oracle during the execution of test suites. Accessing such dynamic metadata is performed through calls to specific methods that need to appear in the component interface, referred to as Built-In Test (BIT) support [5] or probes. Probes are also required to increase controllability, i.e., to make sure clauses are true when needed.

Note that BIT support is a vendor technique widely discussed (e.g., many chapters in [4] describe possible approaches). They however all assume that test cases be defined by the vendor and directly embedded into the component, thus preventing any
tailoring of the testing activity by the user, besides enabling auto-
testing.

Figure 1 captures the framework to be used when reasoning on 
component vendor and user responsibilities to ease component 
testing in the context of a particular component deployment. It’s 
worth noting that this framework, although described in the 
specific context of CSPE-based testing, is generic enough and can 
be used with other component-oriented testing techniques. 
Although vendors and users may consider another testing 
technique, the following activities are still required: the vendor 
must provide testing information and built-in test support 
(activities 1 and 2); the user must be able to target a specific 
subset of the component functionalities and use the information 
provided by the vendor to generate test cases from testing criteria 
and execute them (activities 3 through 7).

Providing strategies for all these activities is a long term effort 
and this article is mostly concerned with activities 3, 4, and 5. 
Existing approaches can be considered to help automate activities 
6 and 7 [24].

4. COTS COMPONENT TEST STRATEGY

Our COTS component test strategy is described by first revisiting 
a number of additional issues regarding CSPE constraints (Section 
4.1). We then provide definitions for our CSPE-based test 
adequacy criteria (Section 4.2) which combine CSPE constraints 
and the testing of logical expressions. Section 4.3 shows how the 
user can identify the parts of the component to be tested. A simple 
example illustrates those concepts in Section 4.4.

4.1 Revisiting CSPE Constraints

Since a test case has to begin with a constraint void of a preceding 
method, a method labeled # is used to represent the start of a test 
case [7]; for instance, constraint (#, m1, true) indicates that 
component interface method m1 can always be executed at the 
start of the test case. In the context of component testing, this also 
corresponds to component instantiation.

With this in mind, a component of \( n \) interface methods, then has 
\( n^2+n \) constraints to be considered: the first term accounts for 
constraints of ordered pairs of methods, whereas the second 
accounts for constraints with preceding method #.

Note that these CSPE constraints can be derived from a number of 
different sources such as: contracts in OCL [29] (or any formal 
language), sequence diagrams, statecharts, or even partially from 
 textual documentation of how the component is to be used. 
Therefore, CSPE-based testing can be used in different contexts 
and in a variety of situations.

4.2 CSPE-Based Test Adequacy Criteria

Because the predicates involved in CSPE constraints can be 
complex, and because we need to make sure that testing exercises 
the various situations under which two methods can execute in a 
sequence, CSPE-based testing criteria need to account for both 
the constraint type and the logic of the constraint predicate. We 
thus combine CSPE criteria (Section 2.1) and logical expression 
testing criteria (Section 2.2), and define a total of 14 CSPE-based 
test adequacy criteria reported in Table 1.

We therefore have five ways to account for constraint types (first 
two columns in Table 1): A, AP, N, NP, and ANP. Then, for AP, 
NP, and ANP, which involve Possibly Valid and Possibly Invalid 
constraints, we also account for varying ways of exercising 
predicates, using criteria PC, IC, PIC, and CoC (third column). 
Table 1 (last column) shows the acronyms for our 14 CSPE-based 
test adequacy criteria. In practice, a subset of these would be 
considered for use based on the testing requirements and budget 
(the criteria range in effectiveness and cost).

In some cases (for constraint types AP and NP), the definitions 
indicate that only a subset of the test cases required by the 
selected logical expression testing criterion is used. The reason is 
that these CSPE-based criteria exercise either true or false values 
of the predicate (but not both).

Table 1. Definition of CSPE-based Test Adequacy Criteria

<table>
<thead>
<tr>
<th>Definition</th>
<th>Constraint type(s)</th>
<th>Predicate Criterion</th>
<th>CSPE-based criterion</th>
</tr>
</thead>
<tbody>
<tr>
<td>The criterion requires that every Always Valid constraint be covered at least once.</td>
<td>A</td>
<td>N/A</td>
<td>A</td>
</tr>
<tr>
<td>The criteria require that every Always Valid constraint be covered at least once, and that every Possibly Valid constraint be covered according to one of the predicate criteria Predicate Coverage (PC), Implicant Coverage (IC), Prime Implicant Coverage (PIC), Combinatorial Coverage (CoC). Only solutions to the selected predicate criterion leading to a truth value of the predicate are selected.</td>
<td>AP</td>
<td>PC</td>
<td>AP–PC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IC</td>
<td>AP–IC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PIC</td>
<td>AP–PIC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CoC</td>
<td>AP–CoC</td>
</tr>
<tr>
<td>The criterion requires that every Never Valid constraint be covered at least once.</td>
<td>N</td>
<td>N/A</td>
<td>N</td>
</tr>
<tr>
<td>The criteria require that every Never Valid constraint be covered at least once, and that every Possibly Valid constraint be covered according to one of the predicate criteria Predicate Coverage (PC), Implicant Coverage (IC), Prime Implicant Coverage (PIC), Combinatorial Coverage (CoC). Only solutions to the selected predicate criterion leading to a false value of the predicate are selected.</td>
<td>NP</td>
<td>PC</td>
<td>NP–PC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IC</td>
<td>NP–IC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PIC</td>
<td>NP–PIC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CoC</td>
<td>NP–CoC</td>
</tr>
<tr>
<td>The criteria require that every Always Valid and every Never Valid constraints be covered at least once, and that every Possibly Valid constraint be covered according to one of the predicate criteria Predicate Coverage (PC), Implicant Coverage (IC), Prime Implicant Coverage (PIC), Combinatorial Coverage (CoC).</td>
<td>ANP</td>
<td>PC</td>
<td>ANP–PC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IC</td>
<td>ANP–IC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PIC</td>
<td>ANP–PIC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CoC</td>
<td>ANP–CoC</td>
</tr>
</tbody>
</table>
4.3 Targeting Functionalities
Recall that the component user should be able to specify which parts of the component, i.e., functionalities of the component, is under test, since the component-based system being built by the user may not use all of the component. We have identified complementary partitioning strategies, referred to as Required Methods, and Required Constraints.

The Required Methods partitioning consists of two steps. First, the user specifies a subset of the component interface methods that are actually used. (The initialization method, labeled #, is always part of this set.) This can be done manually from the user understanding of how the component-based system uses the component, or automatically from UML sequence diagrams showing how that system interacts with the component. The second step is to select a subset of the CSPE constraints based on this subset of component interface methods. Only constraints that involve preceding and succeeding methods identified during the first step are selected. In other words, if component methods m1 and m2 appear in UML sequence diagrams, then constraints (m1, m2, p) and (m2, m1, p’) are selected.

For the Required Constraints partitioning, the user directly selects CSPE constraints to be exercised. This can be done manually or automatically too. For instance, in the latter case, if UML sequence diagrams show two interface methods m1 and m2 called always in that order, then only constraint (m1, m2, p) should be selected.

Once constraints have been selected, according to the two partitioning strategies described above, the user can use any of the CSPE-based criteria to further select constraints.

4.4 Example
We illustrate the concepts introduced above with the simple Queue component example in Figure 2. Figures (a) and (b) are the component class diagram and a possible usage by a user, respectively. Figure (c) recalls the methods that appear in Figure (a) but also provides pre and postconditions in OCL for the component interface methods. Figure (d) is an excerpt of the CSPE constraints for Queue: a total of 56 constraints, including 10 Always Valid constraints, 15 Never Valid constraints, and 31 Possibly Valid constraints.

The Queue component wraps and provides access to a linked list of arbitrary objects. It requires initialization before use, including setting its initial state. The component has seven interface methods: init, empty, eque, dque, top, getLock, and rlsLock. Method init initializes the component before it can be used, empty checks if the Queue is empty, eque and dque add and remove an item to the queue respectively, and top fetches the last item off the queue without removing it.

The Queue component exposes two provides interfaces (diagram not shown): Producer, and Consumer. Each is an interface to a subset of the seven interface methods of the component. The Producer interface is meant to be used by a client program that initializes the Queue component and en-queues to it, while the Consumer interface is meant to be used by a client program that checks for the status of the queue and de-queues from it. Some interface methods are exclusive to each of the interfaces (eque appears in Producer, whereas dque appears in Consumer), while others are accessible from both (empty, getLock, rlsLock).

Methods getLock and rlsLock are used to get and release a synchronization lock on the Queue instance making the component thread safe in a multi-threaded environment: methods that modify the Queue component require that the Queue component be in either an unlocked state (for a single-threaded client), or that the calling thread hold the current lock (for a multi-threaded client).

The constraints reported in Figure 2 (d) are devised from the component OCL contracts in Figure 2 (c). For instance, pre and post-conditions show that calling empty immediately after init is always possible (c9) whereas calling init twice is never valid (c8). Calling eque immediately after init requires that the client holds the current lock or there is no locking mechanism as specified in the two clauses of predicate p1 (c10).

Selecting the Always Valid constraints for testing (A) requires the exercise of 10 constraints. Selecting Always Valid and Possibly Valid constraints (e.g., with AP-PC) requires the exercise of 41 constraints. In combination with AP, PC requires that each Possibly Valid predicate (e.g., p1) be exercised once. Variants of ANP (e.g., ANP-CoC) require exercising all 56 constraints. CoC, in combination with ANP, implies that predicate p1 (for instance) be exercised four times (four truth value combinations of the two clauses of p1).

Assuming the usage of Queue specified in Figure 2 (b) and following the Required Methods partitioning strategy, constraints c10 and c22 are selected. On the other hand, following the Required Constraints strategy would lead to the selection of only c10 if no sequence diagram shows eque being executed before init.

5. AUTOMATION
We have automated the generation of adequate test sets for the criteria defined in Section 4.2. This involves representing constraints in a graph (Section 5.1), associating costs to arcs in the graph (Section 5.2), and devising algorithms to traverse the graph (Section 5.3).

5.1 Graph Representation
We decided to adopt a graph representation for constraints that have to be covered according to the selected CSPE criterion. In this graph, nodes are component interface methods (including #) and arcs represent CSPE constraints.

Each Always Valid constraint c - (m1, m2, true) is modeled as an arc from node labeled m1 to node labeled m2; and the arc is labeled c. Each Never Valid constraint c - (m1, m2, false) is modeled as an arc from node labeled m1 to an error node for method m2, labeled m2, and the arc is labeled c. There is only one way to leave an error node, which is to follow an arc from this node to node #. This arc is labeled ‘-‘ and represents the re-instantiation of the component. In order to facilitate the identification of short test cases, such re-instantiation arcs are added to all the nodes in the graph. Then, during the traversal of
If three Boolean variables are involved in the predicate and the graph, looking for an adequate test set, those arcs can be taken if it means reducing the overall cost of the test set. Each Possibly Valid constraint c = (m1, m2, p) leads to two different arcs because both valid and invalid outcomes might need to be exercised according to the selected criterion. Additionally, Possibly Valid constraints are exercised possibly several times according to the selected predicate criterion (e.g., CoC), for each selected combination of clause truth values resulting in a true value of the predicate there is one arc, labeled “c [true, n]”, from node ml to (error) node m2, where n is the integer value of the clause truth value combination.

These notations are illustrated on the Queue example in Figure 3. For instance, Figure 3 (a) shows Always Valid constraint c9, modeled as an arc directed out of node init and into node eque, labeled c9. The figure also shows three Never Valid constraints: c11, c12 and c22. Each constraint’s succeeding method is modeled as an error node: dque, top, and eque, respectively. Those nodes have only one outgoing arc, a re-instantiating arc labeled “~”, leading to node #. As discussed before, in order to facilitate the construction of less expensive test cases, nodes init and eque also have such outgoing re-instantiating arcs.

Figure 3 (b) additionally shows nodes and arcs for Possibly Valid constraints on one example, namely constraint c38. Assuming that the selected predicate criterion is Prime Implicant Coverage (PIC), and considering that the three-clause predicate of constraint c38 is of the form A and (B or C), two combinations of clause truth values must be selected for the true value of the predicate, and two more combinations must be selected for the false value of the predicate. This results in two arcs from node top to node.

---

3 For instance, assuming that the test case being constructed currently ends with method m and constraint (m, m, p) has to be exercised, it might be less expensive to exercise (m, #, true), that is the re-instantiation arc, followed by a path from # to m (followed by m), than a path from m to m.

4 If three Boolean variables are involved in the predicate and values 0, 1, and 1 (for the three variables respectively) are selected according to the predicate criterion, the integer representation is 3 (0.2^2 + 1.2^1 + 1.2^0).
5.2 Cost Measures

When building an adequate test set, the objective is often to minimize costs. When associating costs to arcs in the graph, different solutions can be considered. A first possible measure for arc cost is to account for the execution time of the succeeding methods in the corresponding constraint, based for example on the complexity of the method, or its possible interactions with a database or a network. A second, simpler measure, often considered in the literature and used in this article, is to count 1 for each succeeding method execution. A third possibility is, in addition to the cost of method executions, to account for the cost of satisfying predicates when exercising Possibly Valid constraints: e.g., the more complex the predicate, the more costly the driver that ensures the predicate is true (or false).

5.3 Algorithms

Once the graph has been built, it can be used to generate an adequate test set. This can be done by traversing the graph and ensuring that the required arcs (as specified by the selected partition and criterion) are indeed covered. Additionally, since a main objective is to have an adequate test set of reasonable cost, we allow the algorithm to traverse non-required arcs if they help reduce cost or must be traversed to reach a required arc. This is exactly the formulation of the (NP-complete) Directed Postman Problem (DRPP), which is an extension of the Chinese Postman Problem, itself similar to the well-known Traveling Salesman Problem [12].

We adapted a solution to the DRPP problem presented in [9]. This graph-based heuristic transforms the original graph into a reduced Eulerian graph and then computes an Euler tour (i.e., a traversal of the graph that goes through all the arcs of the graph exactly once). The transformations aim at identifying which of the non-required arcs will be used in the final solution to the DRPP, and how many times each arc (including required arcs) will be used. These transformations require, among other things, the use of solutions to two standard, well-known graph problems, namely the Shortest Spanning Arborescence Problem [13] and the Minimum Cost Maximum Flow Problem [6]. The algorithm is solvable in polynomial time, and therefore will scale well for very large components. The interested reader is referred to the listed references for additional details.

A prototype tool implementing our approach, called PrestoSequence, has been implemented (29 classes and 1200 LOC). It reads component metadata and test specification provided by the component vendor and component user and outputs test sequences based on the DRPP solution. This tool was used in the following case study.

6. CASE STUDY

The two main objectives of this section are to investigate the fault detection effectiveness of the CSPE coverage criteria we proposed in Section 4.2 and the benefits of using the DRPP-based algorithm presented in Section 5.

The component we use for this case study is part of the Petstore system found in Sun’s J2EE Blueprints collection [27].

Specifically, it is the order fulfillment component based on the Petstore excerpt found in the book “JUnit in Action” [20]. It is responsible for handling the creation of new orders and their processing until fulfillment. The component is implemented using the Enterprise Java Bean (EJB) component model [26] and is deployed on the JBoss Application Server [16]: it is therefore a representative example of component that was developed independently from the current work. We modeled the component with UML through use case scenarios, UML class and component diagrams, and OCL contracts which are defined for each of the component’s interface methods.

An example OCL contract for the fulfillOrder() interface method specifies that the order to be fulfilled (identified by the parameter of type Integer) should exist and be in state VERIFIED before executing fulfillOrder(), and that fulfillOrder() changes the order’s state to FULFILLED:

```
context: Petstore::fulfillOrder(id:Integer)
pres: Order.allInstances->exists(o: Order | o.orderId = id and o.state = #VERIFIED)
post: Order.allInstances->exists(o: Order | o.orderId = id and o.state = #FULFILLED)
```

In this contract, Order.allInstances represents all the orders recorded by the Petstore, uniquely identified by orderId.

From the interface method contracts, we derive the CSPE constraints as described in Section 4.1. An example constraint is c12=(createOrder,printOrder,p3), where predicate p3 is OCL expression Order.allInstances->exists(o | o.orderId = id), that is, the argument passed to printOrder(), namely id, must correspond to an existing order.

One of our objectives is to assess the reduction in cost brought by the DRPP algorithm. This raises two questions. Since we do not have cost data for test suites, what surrogate measure can we use? Second, assuming we have such a measure, what do we compare the results of DRPP with? Regarding the first question, we use test suite size, measured as number of executed methods, as a surrogate measure for cost. It is reasonable to expect the cost of developing and running test suites to be proportional to the number of methods executed. Though different methods usually entail varying costs, this is expected to average out over an entire test suite and is therefore considered a good approximation at the test suite level. As for the second question, we compare DRPP test suites with the algorithm by Karçali and Tai (KT) [17] which
was briefly introduced in Section 2.1, as this is the only possible point of comparison in the literature.

Test sequences were generated for all the different CSPE constraint criteria for a fixed CSPE constraint predicate criterion: Predicate Coverage. Since KT’s approach does not address the coverage of predicates and simply covers Possibly Valid constraints once, this corresponds to our Predicate Coverage criterion. By fixing the CSPE constraint predicate criterion to Predicate Coverage, we thus make sure that we can compare the two techniques and investigate without bias whether the DRPP algorithm is good at minimizing test sequences. Using more complex predicate criteria would only increase the relative improvement of DRPP over KT. Sequences were therefore generated for A, AP-PC, NP-PC, and ANP-PC.

Before reporting our results, we first need to clarify the issue of repeatability of our case study. When building an adequate test set, random variations can come from the construction of the initial Eulerian graph and from the computation of the Euler tour (e.g., there are different ways to traverse the Eulerian graph to produce the Euler tour). As a result, since there exist many ways to build an adequate test set for any of our criteria, is the cost-effectiveness we observe always comparable? A first answer is that, using various randomly generated problems of varying sizes, the heuristic solution to the DRPP problem we use was shown to be very often close to the optimal in [9]: more precisely, the heuristic based solution is on average within 1.4% of the optimal solution (determined with a branch and bound algorithm). This implies that the cost of adequate test sets will not vary substantially across adequate test sets. As for the effectiveness at detecting faults, we computed nine other Euler tours, thus building a total of 10 adequate test sets for ANP-PC. We observed that the fault-detection results we report below were consistent over those 10 sets and can therefore be trusted as representative.

Figure 4 is a chart depicting the size of the DRPP-based approach and the size of the KT algorithm on Petstore in terms of number of methods executed in the test suites. The DRPP-based test sequences clearly show a significantly lower size for all criteria which corresponds to relative improvements ranging from 26% to 53% over the KT algorithm.

![Figure 4. Costs of DRPP-based vs. KT Test Sequences](image)

Let us now turn to the analysis of the fault-detection effectiveness of our CSPE-based coverage criteria. Due to size constraints, the results reported here are for the most complete criterion: ANP-PC. From a practical standpoint we want to determine whether deriving such test suites based on interface method contracts brings any significant advantage over simpler test strategies. In order to investigate this question, a number of issues have to be addressed. First, what kind of faults do we consider in our experiment? Second, since we have no known faults on the component we use, how do we seed faults? Third, with what other test strategy should we compare our CSPE results?

In the context of COTS component testing possible faults belong to one of three categories: errors in the component under test, mismatches between the component under test and the component platform, and mismatches with other COTS components in the platform. We do not consider faults in the first category since we assume that the component under test is delivered by the vendor without faults. This is not necessarily the case in practice but should be representative in most situations with professional vendors. Also, we do not consider faults in the third category since the testing of component compositions is outside of the scope of our current research. In our current work we therefore only consider mismatches between the component under test and the component platform.

We will seed faults corresponding to component-platform mismatches using mutation operators; this is common approach in testing research when investigating the effectiveness of testing techniques [2]. To seed the types of faults described above, the most straightforward solution is to seed mutants in the component platform. This turns out to be at best very difficult because of the complexity of the deployment platform. Another simpler solution is to simulate platform mismatches (e.g., the platform fails to throw an exception when expected), which is the approach we adopt here by simulating such failures directly in the component.

Based on the types of faults under investigation, a set of mutation operators were then selected to manually and randomly seed faults through the entire component code. For example, interface mutation operators [15] were not selected as we assumed that this interface was correct. In our experiment, we used both generic mutation operators [19] and others specifically defined for Java [18]. (Note that these mutants were manually seeded before any test suite was generated.) For example, in the following mutant, a relation operator == is replaced with another relation operator >=:

Original: if (orderState == OrderState.UNVERIFIED)
Mutant: if (orderState >= OrderState.UNVERIFIED)

The all methods and exceptions (AME) test strategy was used as a baseline of comparison to assess the generated CSPE test suite. AME requires that all of the methods and all of the exception instances be covered at least once by a compliant test suite. An exception instance is defined as an exception type thrown by a method, i.e., the same exception type may be thrown by a number of methods and each one is an exception instance. AME [15] is a simple but thorough testing technique applicable without source code, and is therefore a good baseline of comparison.

Out of the entire set of 102 mutants, 21 were identified as equivalent [2], 81 were killed by the CSPE test suite, and 76 were killed by the AME test suite. Therefore, the CSPE test suite killed all non-equivalent mutants, five more than the AME test suite. However, the number of methods executed in the 45 test cases of the CSPE test suite is 149, a much higher number than the 22 methods contained in the 5 AME test cases. The questions that now arise are: (1) How can we characterize the mutants killed only by CSPE and are they expected to correspond to critical, frequent faults in practice? (2) Can we decrease the cost of CSPE
test suites while retaining (most of) their fault detection effectiveness?

![Statechart of the Petstore clearing functionality](image)

Regarding question (1), a careful analysis indicates that these mutants were all found in two specific areas of the Petstore component: where the order processing functionality (1 mutant) and the Petstore clear and confirm/cancel functionality (4 mutants) of the component are implemented. Both of these areas in the component share a common characteristic in that they implement part of the state-based behavior of the component. As an example, the clear and confirm/cancel functionality, is used to clear the Petstore database of all of the orders. This is done in two steps through the clearPetstore and, and either the confirmClearPetstore or cancelClearPetstore methods. The clearPetstore method returns a confirmation code, and clearing the database requires that the same confirmation code be provided in the confirmClearPetstore method. In the same way, the clearing of the database can be revoked by calling the cancelClearPetstore method with the confirmation code. (A set of confirmation codes is maintained in the component—confirmCodes— and stored in the database.) This behavior can be modeled by the (incomplete) state diagram in Figure 5 (Transitions have been numbered for convenience.) If a code does not exist when calling confirmClearPetstore or cancelClearPetstore (while the Petstore is not empty), then an exception is thrown (transition 2 or 3).

As a result of the mutants the incorrect behavior results in confirmClearPetstore and cancelClearPetstore failing to properly remove the valid confirmation code from the set of maintained codes after their execution. This corresponds in Figure 5 to removing the actions circled with dashed lines. As a result, if a call to confirmClearPetstore is made with a code that has already been used, for instance in a previous call to confirmClearPetstore, then transition 4 is taken clearing the database again. The second call to confirmClearPetstore should have triggered transition 3, resulting in an exception being thrown (the code exists in the set of valid codes when it should not). The Petstore can be cleared or the clearing can be cancelled multiple times with the same confirmation code. Note that these mutants emulate a deployment failure as this would correspond to a scenario where the container does not forward the request properly to the database or where the database does not properly remove the code from its record.

The reason why only CSPE killed these mutants is that it ensures that the faulty behaviors were exercised (i.e., trying to execute confirmClearPetstore twice) whereas the AME technique only verifies the basic Petstore clearing functionality by asserting that the database is empty after a confirmClearPetstore with valid confirmation code, that it has been unaltered after a cancelClearPetstore with valid confirmation code, or that it has been unaltered and an exception has been thrown after a confirmClearPetstore with an invalid confirmation code.

If we now generalize, the mutants missed by AME and killed by CSPE are faults related to the state behavior of parts of the component and AME does not fully exercise this state behavior. From what we know on testing OO systems [5], detecting state behavior-related faults is crucial as many critical functionalities in such systems exhibit such behavior. Many test techniques relying on detailed design information or code are state-based [5] and are defined in terms of state model coverage. This clearly reflects the fact that detecting such faults is of practical importance. We can therefore conclude that our CSPE-based approach is likely to be useful in detecting certain types of faults, namely state-based faults, and this is going to be of practical importance. Note that our approach achieves such results without providing any explicit state model of the component to the user (Section 2.3).

If we now come back to question (2) above, can we find a way to select a subset of CSPE test cases that would retain a high fault detection effectiveness with a cost comparable to that of AME? To achieve this, we devised the following test case selection heuristic based on mutant analysis:

1. Seed representative mutants in the component (following the procedure described above)
2. Evaluate each individual CSPE test case with respect to its mutation score
3. Order the test cases in descending score order.
4. Select the first n test cases, n being determined by available test resources

The idea is to use mutant analysis as an indicator of test cases with high fault detection capability so as to select them for testing. The tasks of creating and executing test suites on mutants can easily be automated, as done in this case study, and the effort overhead should therefore not be an issue. But the question remains of whether such a heuristic is likely to work in practice. Continuing with our case study, we determined based on the average number of methods per test case in CSPE and AME test suites, that we would need to select a subset of 7 CSPE test cases to ensure the size of the resulting test suite would not be larger than that of the AME test suite. In order to obtain realistic results, we needed to set up the experiment so that the heuristic would be based on a subset of mutants (learning set) and be evaluated on a different set (evaluation set). We randomly generated 10 pairs of such sets to be able to statistically compare the results of AME with those of CSPE subsets. Over 10 evaluation sets the average number of mutants killed by AME and CSPE were 37.9 and 40.5, respectively. Given that 40.5 is the maximum possible average 5, 100% of the mutants were systematically killed by our subsets of 7 CSPE test cases whereas AME killed an average of 94% of the mutants. A statistical test of significance, whether we use a paired t-test or a non-parametric Wilcoxon signed rank test [11], indicates that the difference is significant with a p-value < 0.0001. Furthermore, in terms of cost, the average number of methods in CSPE subsets was smaller than in the AME test suite: 19.2 and 22, respectively. We can therefore conclude that the heuristic presented above can be effective at selecting a subset of CSPE test cases so as to significantly reduce its cost while retaining its fault detection capability.

5 Five of the evaluation sets contain 40 mutants and another five contain 41 mutants.
7. CONCLUSION
This paper presented a systematic methodology for the testing of deployed COTS components based on a careful analysis of interface method contracts used to generate interface method sequential constraints. This methodology does not require the component design or code but implies that the component vendor provides a specific test interface as part of the component’s public interface so as to allow the component to be controllable and observable, two basic testability requirements. This paper also proposed an optimization algorithm to generate test sequences of minimal length to fulfill the proposed coverage criteria, thus reducing the cost of testing. A case study using mutation analysis, performed on a representative component, shows that the proposed test strategy is effective at killing mutants and that its cost can be significantly reduced by using a simple test selection heuristic based on mutation analysis. More precisely, a careful analysis revealed that it is particularly effective at killing mutants affecting the state behavior of parts of the component thus suggesting it is particularly effective at detecting state-related faults, a category of faults very common in object-oriented software. Future directions of this research include full automation of the test generation method, and the extension of the testing technique to address component composition testing.

ACKNOWLEDGEMENTS
This work was partly supported by a Canada Research Chair (CRC) grant. Lionel Briand and Yvan Labiche were further supported by NSERC operational grants. This work is based on M. Sówka’s masters thesis and is part of a larger project (www.sce.carleton.ca/Squall).

REFERENCES