Investigating the Use of Analysis Contracts to Improve the Testability of Object Oriented Code

L. C. Briand, Y. Labiche, H. Sun

Software Quality Engineering Laboratory
Carleton University
Systems and Computer Engineering
1125 Colonel By Drive
Ottawa, ON, K1S 5B6, Canada
e-mail: {briand, labiche}@sce.carleton.ca

ABSTRACT

A number of activities involved in testing software are known to be difficult and time consuming. Among them is the definition and coding of test oracles and the isolation of faults once failures have been detected. Through a thorough and rigorous empirical study, we investigate how the instrumentation of contracts could address both issues. Contracts are known to be a useful technique to specify the precondition and postcondition of operations and class invariants, thus making the definition of object-oriented analysis or design elements more precise. It is one of the reasons the Object Constraint Language (OCL) was made part of UML. Our aim in this paper is to reuse and instrument contracts to ease testing. A thorough case study is run where we define OCL contracts, instrument them using a commercial tool, and assess the benefits and limitations of doing so to support the automated detection of failures and the isolation of faults. As contracts can be defined at various levels of details, we also investigate the cost and benefit of using contracts at different levels of precision. We then draw practical conclusions regarding the applicability of the approach and its limitations.

Keywords

Object-Oriented Testing, Testability, Contracts, Object-Oriented Analysis
1 INTRODUCTION

One important issue in testing software systems is to develop test oracles that can automatically determine, within the test driver, whether a test case is successful or not. Developing such test oracles manually is expensive and represents a major cost when writing test drivers. Furthermore, there is no general solution or methodology for their construction, thus making them frequently complex and error-prone [3, 4]. Another issue, which is especially critical in object-oriented systems, is to perform a diagnosis of a failure in order to determine the fault(s) that caused it. Again, such a debugging activity is extremely time-consuming as it is not uncommon for developers to spend days or weeks isolating faults in a large system. This problem is made more acute in object-oriented designs where functionality is spread across large numbers of objects.

A common practice, first introduced in the Fusion methodology [6] during object-oriented Analysis, as opposed to the Design or Code level [16], is to define contracts for all public methods and classes in the Analysis class diagram, i.e., pre- and postconditions for methods, class invariants for classes [15]. The goal is to make the Analysis model more precise and complete. In the context of the Unified Modeling Language (UML), such contracts can be expressed in the Object Constraint Language [25], as illustrated in [17]. The question is then whether we can reuse such contracts to address both the test oracle and diagnosability issues. In short, can we leverage Analysis contracts for the purpose of supporting testing as well? More precisely, we want to investigate the following questions:

- Can instrumented contracts be used as a substitute to hard-coded test oracles in the test drivers?
- Can instrumented contracts be used to significantly lower the effort of locating faults after the detection of a failure?

Furthermore, because contracts can be defined at various levels of precision (and therefore complexity), we need to investigate whether defining and instrumenting very precise contracts brings real benefits considering the additional cost they entail.

The paper first discusses related works and motivates our study. Then Section 4 provides detailed information about the way contracts were defined. Section 5 describes the overall strategy we followed.
to investigate the above questions. Section 6 reports on the detailed setting and procedures of our case study. Section 7 provides detailed results. The validity of our study is discussed in Section 8 and conclusions are drawn in Section 9.

2 RELATED WORKS

A number of authors have discussed the use of executable assertions to improve testability and practitioners have been using them for decades. Several directions have been investigated with respect to the definition and usage of assertions. Rosenblum [20] reports on several experiments that led to a classification of the assertions that were most effective at detecting faults in C programs. An automatic process for the identification of assertion insertions is described in [21], the objective being to find statements where faults are likely to hide and then specify an assertion for these statements. In this pioneering work on assertions, Voas proposes a technique that is based on mutation testing and does not require an oracle. However it requires very large numbers of executions, even for small programs. As an example, [23] reports on a program that computes a solution to the quadratic equation \( ax^2 + bx + c \) and requires tens of thousands of executions. In [7], to the opposite end of the spectrum, the authors investigate abstract testing, an extension of program static analysis aiming at verifying user-provided specifications by abstract interpretation of the program semantics. This strategy does not require program execution (only static analysis) and can detect errors such as readings of non-initialized variables, buffer overflows, and invalid arithmetic operations (e.g., division by zero). However, functional failures that do not result from these kinds of errors cannot be detected.

In the context of object-oriented systems, Binder [4] discusses the advantages of so-called Built-In Test (BIT) oracles, based for example on class invariants and responsibility-driven executable assertions. One issue that he raises is though an assertion can detect out-of-range and some in-range – but incorrect – inputs, it is not clear that most incorrect results would necessarily cause a run-time assertion violation. In other words, Binder indirectly poses the question of whether assertions, for example based on contracts,

\footnote{In [20], Rosenblum briefly explains how assertions have been used as an aid to software development for more than 30 years.}
can be used as effective test oracles. In such a case, the advantage would be that the oracles could be derived from the contracts in the Analysis documents and instrumented using an assertion tool supporting contracts. The investigation of failure detection effectiveness of instrumented contracts is one important objective of our paper.

A strategy for implementing BIT support was already proposed in [24], showing how class test drivers and oracles could be embedded in the class itself. The basic idea developed in [10] is similar but they go further by suggesting a procedure to use Design by Contract [15] to implement more effective BIT support by instrumenting contracts as assertions. In order to clearly identify the benefits of instrumenting contracts, [2] proposes definitions of measures for two quality factors, namely robustness and diagnosability, though the definition of the latter one is not precise and operational. Robustness captures the degree to which the software is able to recover from internal faults that would otherwise have provoked a failure, and diagnosability captures the fault isolation effort after the occurrence of a failure. We further discuss the definitions and measurements of robustness and diagnosability in the following section.

Several case studies are briefly reported in [2]. Based on the three systems analyzed, the authors suggest that robustness is quickly enhanced by the introduction of contracts. It is however unclear how this was measured and on which quantitative basis these conclusions are drawn. What is the rationale followed for the definition of contracts? Are contracts exclusively based on analysis information or also included design and implementation information? It is therefore important to perform a well-designed, precise study that looks into the matter. Furthermore, measuring improvements in robustness due to contracts is important but not sufficient. We need to better characterize the mechanisms by which contract assertions fail to detect failures and determine whether some strategies could be devised to instrument contract assertions in a more effective way.

As a side issue, we believe that Robustness is a misnomer as the only thing we consider here is the probability of detecting a failure through contract assertion violation. A better term would be observability, one of the two components of testability [4], that we will use in the remainder of this text.
In this paper, we perform a carefully designed and reported case study that furthers existing work in four ways:

- It provides a precise, operational measure for diagnosability;
- It investigates the effectiveness of contract assertions in improving observability and diagnosability in object-oriented systems by following a precise contract definition procedure, exclusively based on Analysis information;
- It assesses the impact of simplifying contracts on observability and diagnosability, for the sake of easing instrumentation;
- It investigates the mechanisms by which failures remain undetected (not observed) or are detected far away from their corresponding fault location (low diagnosability);
- It analyzes the best ways to instrument contracts with a particular emphasis on Java systems.

3 OBSERVABILITY AND DIAGNOSABILITY

In the following subsections, we go more in depth in discussing the two testability factors we are assessing. We review the work of [2], perform a sensitivity analysis of an observability model, and propose a new rationale to define diagnosability.

3.1 Observability

A mathematical model of Observability is provided in [2]. The Observability of a system (called global Observability) composed of a set of interconnected components, is defined as the probability that an internal fault is detected by any one of the components. The weakness of a system is the opposite probability: the probability that a fault is not detected. Let $P_i'(S)$ being the probability that a fault in component $i$, part of system $S$, is not detected in $S$. $P_i'(S)$ is the weakness of system $S$ on faults in $i$. The mathematical expression of $P_i'(S)$ then relies on the following observation: an internal fault in a component plugged into a system can be detected either by the component itself or by one of its clients (or children). For simplicity, the authors only consider faults that are detected by a component directly
dependent on the faulty one, even though dependences are transitive. Then, \( P_i'(S) \) is the probability that the fault is not detected by component \( i \), nor by its direct clients and children: \( P_i'(S) = P_{ii} \prod_j P_{ij} \) (we assume here that \( P_{ij} \) are independent) where \( P_{ij} \) is the probability that a fault in component \( i \) is not detected by the direct client \( j \) of \( i \). Then the weakness of system \( S \) is \( P'(S) = \sum_i P_{fail}(i) \prod_j (1 - P_{ij}) \) where \( P_{fail}(i) \) is the probability a failure comes from executing a fault in component \( i \). Let \( P(S) \) being the Observability of system \( S \), and \( P_{ij} \) being the probability that a fault in component \( i \) is detected by a direct client \( j \) of \( i \). The previous expression can be transformed as follows:

\[
1 - P(S) = \sum_i P_{fail}(i) \prod_j (1 - P_{ij}) \]

(1)

In [2], the authors investigated their Observability measures (\( P_{ij} \) and \( P_{ii} \)) on three different software systems. In these three systems, \( P_{ij} \) was observed to be proportional to \( P_{ii} \): \( P_{ij} = c \times P_{ii} \), where \( c \) is the proportionality coefficient between the fault detection effectiveness of a class (in their study, components were classes) and the fault detection effectiveness of one of its server classes where a fault is assumed to be located. The authors assumed that a client class is less effective than its server classes in detecting faults that are in these server classes, thus leading to \( 0 \leq c < 1 \). Based on fault seeding and empirical analysis, Baudry et al estimated \( c \) to average at 0.8 (though \( c \) values are expected to vary across pairs of client/server classes).

As a simplification [2], assuming that (i) the probability the failure comes from component \( i \) is the same for all the other components (\( P_{fail}(i) = 1/n, \forall i \), where \( n \) is the number of components), and (ii) the probability that a fault in component \( i \) is discovered by contract assertions in \( i \) is the same for all the components (\( P_{ii} = P_{jj} = P_0, \forall i, j \)), the authors were able to demonstrate, as expected, a significant
increase in Observability of the three systems with the improvement in Observability of their components: \( P(S) \) is a function in \( P_0 \).

Given assumptions (i) and (ii), the results on the three different systems show that not using contracts results in systems that are not robust, but that adding simple contracts improves significantly their Observability\(^2\). In addition, it is noted that the three Observability curves for the three systems are similar and the most plausible explanation seems to be that similar dependency densities between classes can be found across the systems the authors used. It was also shown on one system, that a change in \( c \) (e.g., 0.2 instead of 0.8) had a small impact on the system Observability \( P(S) \).

Though a model was proposed, no systematic sensitivity analysis of \( P(S) \) against these two factors was performed: coefficient \( c \), and class dependency density \( k \), that is, the number of clients of class \( i \). Based on the above simplifying assumptions, and further assuming that dependency densities are identical for every class \( i \) (e.g., we can use the average class dependency density), we derive from (1) the following formula:

\[
P(S) = 1 - (1 - P_0) \ast (1 - c \ast P_0)^k
\]

\( (2) \)

\(^2\) A definition of ‘simple’ contracts is not provided in [2] though.
Based on (2), Figure 1 shows how $P(S)$ relates to $P_{ii}$ for varying values of $c$ and $k$. In Figure 1 (left), $c$ is constant and $k$ takes values 1, 3, and 6. In Figure 1 (right), $k$ is constant and $c$ takes values 0.2, 0.5, and 0.8. From these figures, it is clear that $P(S)$ is very sensitive to these two factors. One important conclusion we can draw from the sensitivity analysis is that $P_{ii}$ does not need to approach 1 for contracts to ensure a high probability of failure detection. After a certain threshold, in many $(c, k)$ configurations, $P(S)$ reaches a plateau close to 1 for values of $P_{ii}$ that are significantly below 1. For example, when $c=0.8$ and $k=3$, we reach a plateau when $P_{ii} \sim 0.75$. This suggests that contracts do not need to be highly precise or complete to be useful. Considering that $k$ may vary significantly from system to system and that $c$ is very difficult to estimate, we can conclude that sensitivity analysis gives us little insight into the potential effectiveness of instrumented contracts. Furthermore, it is difficult to assess the impact of the simplifying assumptions we had to make above on the validity of the results we obtained. Therefore, empirical investigations are crucial if we want to answer any question about contracts and observability.

Based on the analysis of three systems, [2] focused on class testing and suggested that diagnosability is quickly enhanced by the introduction of contracts. For instance, they improve contracts for one of their systems and observe an increase in observability from 58.5% to 87.5%. Similar results are also reported in [1]. However, the authors do not provide the rationale for the determination of the first set of contracts nor the procedure applied for their improvement. It is also unclear whether contracts were exclusively based on analysis information or also included design and implementation information. It is therefore important to perform a well-designed, precise study that looks into the matter. Section 5.1 describes how we measure observability in the context of our case study.

---

3 It is reasonable to expect classes to show, on average, at least one dependency and at most six. The average class dependency density in the three cases studies described in [2] are 1.9, 2.6, and 2.8, respectively.

4 It depends, to a large extent, on how precise the contracts are, the type of dependencies between clients and servers, the functionality of the classes.
3.2 Diagnosability

Regarding diagnosability, [2] does not provide a mathematical model similar to that of Observability, but only briefly comment the main ideas and results. Once a failure has been revealed diagnosability relates to the ease with which the corresponding fault(s) can be isolated. With this definition in mind, diagnosability can be measured as the effort involved in isolating faults. Since this is in practice difficult to measure and is subject to random fluctuations depending on the type and location of faults, diagnosability can indirectly and objectively be estimated based on the maximum size of the code to be analyzed to ensure the fault will be detected. This is a reasonable indicator as the isolation effort is related to the size of the sets of potentially faulty methods or statements.

[2] proposes to base their estimation of diagnosability on the concept of indistinguishability set. This is the set composed of statements bounded by two consecutive contract assertions in an execution flow. The intuitive idea is that, when a failure is observed due to a contract assertion violation, the statements to be investigated (which cannot be distinguished in terms of their likelihood to contain a fault) are between the last contract assertion to execute successfully and the contract assertion violated. The larger the indistinguishability set, the more expensive the fault diagnosis.

Since [2] only provides the main ideas and results, some elements of the approach remain to be clarified. For instance, it is not clear whether the definition of indistinguishability set is adequate to measure
diagnosability. The statements in this set may turn out not to be faulty as the fault(s) may have been executed before the last successfully executed contract assertion. In other words, the incorrect internal state of the system can be due to a fault outside of the indistinguishability set, as contract assertions cannot be expected to be perfect test oracles. Measuring diagnosability that way is at best a rough approximation. Measuring diagnosability must involve an estimation of the statements to be analyzed based on the location of the fault(s). Furthermore, it has to be measured based on an actual set of failures and corresponding faults, not just based on contract information.

In this paper, we will perform an in-depth investigation of the impact of contract executable assertions (i.e., instrumentation of contracts in code) on diagnosability. As defined in Section 5.2, we are to measure diagnosability in a way that will enable systematic comparisons between programs instrumented with contracts and programs that exclusively use classical test oracles. This measure is based on an estimation of the size of the diagnosis work to be performed by measuring a ‘distance’ between the location of failure detection and the location of the fault that caused the failure.

4 THE DEFINITION OF CONTRACTS

We are investigating the use of contracts to support testing. But what do we mean by contract? Though the notion of Design by Contract [15] is well established, the level of detail of those contracts and the way they are defined need to be specified as they may vary a great deal in practice. Therefore, assessing the impact of contracts on testing is not meaningful if we do not define precisely how those contracts are devised. In other words we need to define precisely what we are evaluating. Based on experience, we define a set of practical rules we have been using and refining to define our contracts. We also provide some representative examples, and explain what have been the implementation issues and procedures to instrument those contracts. Recall that we consider contracts to be defined during Analysis, in our case by using the Object Constraint Language (OCL) in the context of the UML notation [18, 25]. This choice was made because UML is now a de-facto standard notation for object-oriented Analysis and
OCL is the most natural way to precisely define contracts in that context\(^5\). Defining contracts during Analysis entails that our contracts must be based exclusively on application domain knowledge and are instrumented in the application domain classes of the system and their public methods.

### 4.1 Contract Definition Rules

In order to be systematic and complete, it is recommended that contracts be defined by following some guidelines that can be formalized as rules. It is particularly important in the context of this paper to precisely define the conditions under which we obtained the results presented in Section 7. The rules defined below apply in the context where OCL contracts are reused from Analysis documents. Contracts have three types of components (class invariant, method pre-condition, method postcondition) and may involve five kinds of elements: class attributes, class associations, method input or output parameters, and method return values. We systematically identify what kind of elements may be needed for a given type of contract, as described in Table 1.

<table>
<thead>
<tr>
<th>Types of elements</th>
<th>Contracts</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Invariant</td>
</tr>
<tr>
<td>Attribute</td>
<td>X</td>
</tr>
<tr>
<td>Input parameter</td>
<td>X</td>
</tr>
<tr>
<td>Output parameter</td>
<td></td>
</tr>
<tr>
<td>Return value</td>
<td></td>
</tr>
<tr>
<td>Association</td>
<td>X</td>
</tr>
</tbody>
</table>

Table 1 – Involvement of attributes, parameters and return values in contracts

Class attributes are systematically involved in class invariants, pre- and postconditions as checking the value of attributes and their relationships is a key element of contracts. To be more precise, the usual rules followed to define contracts based on attributes and parameters are:

\(^5\) This also addresses the difficult challenge, mentioned in [22], of finding/designing a language in which pre and post-conditions are specified by the user.
(1) Check whether attributes and input and output parameters are within the allowed domain of values;

(2) Check whether expected relationships among attributes’ and input and output parameters’ values hold.

Furthermore, postconditions typically describe the addition or destruction of association links whereas pre-conditions verify that certain links are present or absent so as to enable the correct execution of a method. Invariants may also check whether certain association links are present or absent for the system to be in a legal state.

Output parameters and return values are part of the methods’ behavior to be modeled by postconditions. Since neither class invariants nor pre-conditions involve the modeling of method effects, they do not use output parameters or return values.

4.2 Issues

In addition to the above general rules, several remarks should be made regarding the level of detail in postconditions and the use of delegation in methods.

4.2.1 Level of detail in postconditions

A postcondition may involve returned values, a number of modifications of attributes or output parameters, and these may vary according to different conditions under which the method is executing. For instance, suppose the specification of method `int Result (int x, int y)` indicates that: when `x > y`, the return value is 1; when `x < y`, the return value is 2, and when `x == y`, the return value is 3.

In that situation, like in many others, we can define the corresponding postcondition at different levels of precision, as illustrated in Table 2. It is clear that these postconditions show very different levels of detail and complexity, but both are correct in the sense that they make a true statement about the effect of the `Result()` method.
if \((x>y)\) then result = 1
else if \((x<y)\) then result = 2
    else result = 3
endif
endif
(Note that this is close to the code structure)

result >=1 and result <=3

**Table 2 – Level of detail in postconditions (example)**

It is difficult to come up with general guidelines for selecting an appropriate level of precision for specifying postconditions. Nevertheless, we can consider the following issues:

- A weak postcondition may not be able to detect an illegal system state (as defined by the states of its objects and their associations);

- Specifying in great detail a postcondition will help detect failures and locate faults but more effort is required to define the postcondition, check its correctness (the postcondition is more likely to contain faults due to its complex logic), and instrument it. Furthermore, since we do not know how many faults are present and can potentially be caught, the cost-effectiveness of investing in a more precise postcondition is not known beforehand.

Only experience and practice, through case studies such as the one presented here, will tell us about the potential gains of using more detailed contracts. In our study, in order to *systematically* investigate the impact of contract precision on observability and diagnosability, we defined contracts at three levels of precision (see Table 3 for an abstract example):

- The highest level of precision possible, keeping in mind we define Analysis contracts. Here every distinct condition, possibly resulting from a different set of inputs or system state, is distinguished in the postcondition. The motivation in this case is to assess the *maximum*, potential benefit of contracts.

- An intermediate level of precision that only distinguishes conditions for the standard situation (expected execution) from exceptional situations that are also addressed by the method.
- A low level of precision that just defines the ranges/enumerations of values expected as resulting from executing the method.

<table>
<thead>
<tr>
<th>if $A_1$ then $B_1$</th>
<th>if $A_2$ then $B_2$</th>
<th>$B_3$ or $B_1$ or $B_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>else if $A_1$ then $B_1$</td>
<td>else $B_1$ or $B_2$</td>
<td>// $A_1$ is assumed to be</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// the standard situation</td>
</tr>
<tr>
<td>Highest level</td>
<td>Intermediate level</td>
<td>Lowest level</td>
</tr>
</tbody>
</table>

Table 3 – Three levels of details for contracts, an example

We provide examples of contract definition at the three levels of precision in Appendix B. The above definitions allow us to work at consistent levels of precision across the system and therefore be systematic in our investigation.

4.2.2 Delegation and postconditions

When a (client) method returns a value that merely transmits the return value of another (server) method, one may ask whether a postcondition, which is the exact copy of the server’s postcondition, is to be defined for the client. In other words, in the following code chunk, is it necessary to check the return value’s validity in the postcondition of `valueOfBalance()` since this was checked in `getBalance()`?

```java
public int valueOfBalance () {
    return currentAccount.getBalance();
}
```

We can identify three possibilities. The postcondition is defined and instrumented (1) only for the server, (2) only for the client, or (3) both for the server and the client. We evaluate each of these three alternatives below according to two criteria: instrumentation cost and fault detection:

- If the return value’s validity is only checked in the server we avoid redundancy at run-time when executing assertions. However, we cannot detect errors in the client, e.g., a wrong method call. We then rely on a side effect to detect the error.
- If the return value’s validity is only checked in the client we also avoid, to some extent, redundancy in the assertion executions. Diagnosability is improved over the previous case since when the return value is invalid, it is either because of an error in the call or an error in the client.

- If the return value’s validity is checked in both the server and client(s), we have redundant assertion executions. The consequence is additional assertions to be written and maintained in the code. However, diagnosability is improved since when a contract assertion is violated, we know precisely whether it is in the client or the server.

We chose the latter solution since it provides the best fault detection and diagnosability capabilities, and considering all the assertions involved in such an experiment, redundant assertions represent a small overhead. We also discuss below the problems introduced by delegation when instrumenting contracts in the source code (Section 4.4.2).

4.3 Contract Examples from the ATM case study

As mentioned above, we use the Object Constraint Language (OCL) to define our contracts. We provide here, in Table 4, some contract examples from the ATM case study (at the highest level of precision). As a writing convention for contracts, we introduce first the context (a class in the case of an invariant, plus a method in the case of pre- and postconditions) as well as a label for the contract component being defined: inv for invariant, pre for pre-condition, and post for postcondition. From these examples (and Appendix B), we can see that the case study involves contracts that are far from being trivial and this is something we have to expect in actual systems. We need instrumentation tools that support OCL to the maximum extent possible and well-defined implementation rules to instrument such contracts systematically in the context of the selected tool. The further the tool’s instrumentation language from OCL, the higher the cost overhead of using contracts as implementation rules. This issue is discussed in the next section.

The meaning of the four example contracts in Table 4 is provided by order of appearance:

Contract 1: A Bank object must satisfy the following: For the current transaction, the card and account must exist and the available balance must be less or equal than the current balance.
Contract 2: Method `Bank::finishDeposit(...)` requires that the `ATMnumber` exists and that the `serialNumber` be positive. Then, the method ensures that, provided that the parameter `successful` is true, the account is credited with the amount of the transaction.

Contract 3: The method ensures that, according to the value entered with the keyboard, the correct transaction is initiated (deposit, transfer, ...).

Contract 4: Method `ATM::checkIfCashAvailable(...)` verifies whether the requested amount is less than the amount available in the ATM (it asks the `cashDispenser`).

<table>
<thead>
<tr>
<th>context:</th>
<th>Bank</th>
</tr>
</thead>
<tbody>
<tr>
<td>inv:</td>
<td>self._currentTransactionCard-&gt;notEmpty() and</td>
</tr>
<tr>
<td></td>
<td>self._currentTransactionAccount-&gt;notEmpty() and</td>
</tr>
<tr>
<td></td>
<td>self._currentTransactionAccount._availableBalance.getValue() &lt;=</td>
</tr>
<tr>
<td></td>
<td>self._currentTransactionAccount._currentBalance.getValue()</td>
</tr>
</tbody>
</table>

| context: | Bank::finishDeposit(ATMnumber:int,serialNumber:int) : void |
| pre:     | self.aTMMachineProfile->exists(_ATMNumber = ATMnumber) |
|          | and serialNumber >0 |
| post:    | self._currentTransactionAccount._currentBalance.getValue() = |
|          | self._currentTransactionAccount._currentBalance.getValue@pre+ |
|          | _currentTransactionAmount.getValue() |

| context: | Transaction::chooseTransaction(session:Session, atm:ATM, bank:Bank): Transaction |
| post:    | let option : integer = atm.getMenuChoice ("Please choose a transaction type:", 4, TransactionMenu) |
|          | (option = 1 and result oclIsTypeOf WithdrawlTransaction) or |
|          | (option = 2 and result oclIsTypeOf DepositTransaction) or |
|          | (option = 3 and result oclIsTypeOf TransferTransaction) or |
|          | (option = 4 and result oclIsTypeOf InquiryTransaction) |

| context: | ATM::checkIfCashAvailable(amount: Money):Boolean |
| post:    | result = not (self.cashDispenser.currentCash().less(amount)) |

Table 4 – Contract examples for the ATM
4.4 Selection of an instrumentation tool and Implementation rules

The use of an instrumentation tool has a significant impact on the cost-effectiveness of using contracts to support testing. Its selection is therefore important as well as its effective usage following instrumentation guidelines.

4.4.1 Selection of an instrumentation tool

A tool named Jcontract [9], from Parasoft, was selected to instrument the contracts. Though a few other tools exist, it was selected for its ability to handle local variables within contracts (see discussions below). Jcontract provides a tool that instruments and compiles Java code so that extra byte code is added to check pre-conditions and class invariants at method entry and postconditions and class invariants at method exit during run time. If a pre-condition, postcondition, or class invariant is violated then Jcontract may be set up so that a message may be sent to a monitor managed by the tool, an exception can be raised, and/or the program may be terminated. The language used to instrument contracts (DbC) is similar to OCL⁶ and provides specific Javadoc tags that can be used to specify all required aspects of contracts in the code without side effects. We will use @pre, @post, @invariant, @assert tags as shown in Figure 2 and Figure 3. The information provided by the Javadoc comments is then used by Jcontract to instrument the code. The tag @assert provides a way to check assertions within the method block and this is how we intend to check constraints involving local variables. Local variables are out of the scope of all other tags, though method parameters, method return value (post-condition only), and attributes can of course be accessed.

---

⁶ But it is not nearly as complete and is closer to implementation details as it is used to instrument the code.
Using Jcontract, contracts assertions are evaluated during the execution of a method as follows:

- Evaluation of the pre-condition (defined by @pre) and the invariant (defined by @invariant) at method entry, just before the first statement of the method;

- Evaluation of the postcondition (defined by @post) and the invariant (defined by @invariant) at method exit, just after the last statement of the method;

- Evaluation of the assertions (defined by @assert) at their location in the body of the method.

### 4.4.2 Instrumentation guidelines

#### Postcondition assertions

A first practical issue when instrumenting contracts is the use of method calls in contract assertions (see examples 3 and 4 in Table 4). Note that OCL expressions are side effects free since they can only use method with isQuery being true [18] (the isQuery property specifies whether the execution of the method leaves the state of the system unchanged). But we need to pay attention to method calls that
require inputs from or send outputs to a user or device. For example, in the post-condition of method
\texttt{Transaction::chooseTransaction()} (example 3 in Table 4) method \texttt{getMenuChoice()} requests a
choice to be entered from the keyboard. If that method is executed again during the evaluation of the
postcondition, the choice would need to be re-entered, thus creating the opportunity for inconsistencies.
During testing, user or device inputs are typically obtained from a stub, simulating a user entering
information on the keyboard. We therefore need to find an instrumentation solution for postconditions
containing those types of method calls.

The solution involves removing, in some way, the method call from the postcondition and substituting it
with something preserving the semantics of the postcondition. A solution consists in storing the result of
the method call during execution so that it can be reused in the postcondition. From a practical
standpoint, this can be done in two ways:

1. We can add an attribute to the client class (e.g., of type integer, for calling
\texttt{Transaction::chooseTransaction()}), use that attribute to record a relevant result after the
execution of the server (e.g., \texttt{getMenuChoice()} which returns an integer) in the body of the
client, and use that attribute in the postcondition.

2. We can add a local variable to the client method in order to record the server’s result and use an
\texttt{@assert} tag to access that local variable and check the postcondition’s clauses involving that
variable.

The first solution is not recommended as it requires a modification to the system class diagram, thus
leading to confusion and a probable cluttering of the diagram. The second solution only requires the
introduction of a local variable (see Figure 3 for the implementation of the postcondition of example 2 in
Table 4). Such a variable may already exist and coding standards might also be used to ensure all return
values are stored in local variables. However, not all tools supporting contracts enable the check of local
variables. This capability was one of the major reasons for us to select Jcontract as it became clear, early
in our study, that this was unavoidable if the postconditions were to be reasonably complete and
effective.
When a local variable must be checked, a good practice is to put the `@assert` tag as close as possible to either each return statement (usually just before it) or at the end of the method when there is no return value. This ensures that the value of the variable checked in the assertion is exactly the one that was intended to be checked in the postcondition (but could not because of a method call involving inputs or outputs).

The modification required in the `@post` tag and the addition of an `@assert` tag must be performed carefully. Consider for instance the situation in which the postcondition of a method is $A \lor B$, where $A$ and $B$ are clauses possibly involving method calls. It implies that, if at method exit the Boolean expression $A \lor B$ is false (i.e., both $A$ and $B$ are false), an exception is raised. Let us consider the case where $A$ is the only clause that contains a method call involving inputs/outputs. Then an `@assert` tag...

```java
public static Transaction chooseTransaction(Session session, ATM atm, Bank bank){
    ...
    int option = atm.getMenuChoice("Please choose a transaction type:", 4, transactionMenu);
    Transaction result; //added variable for the implementation of the contract
    switch (option) {
        case 1: result = new WithdrawlTransaction(session, atm, bank);

        // the original statement was ‘return new WithdrawlTransaction(...);’
        break;
        case 2: result = new DepositTransaction(session, atm, bank);
        break;
        case 3: ...
        case 4: ...
        default: result = null;
    }

    /** – Javadoc comment –
    Assertion defined before (as close as possible to) the return statement
    @assert (option == 1)&&( result instanceof WithdrawlTransaction ) 
    || (option == 2)&&( result instanceof DepositTransaction )
    ...
    */
    return result;
}
```

Figure 3 Instrumentation of a postcondition using the `@assert` tag with Jcontract
verifying A is added just before the end of the method, and clause A is removed from the postcondition (the \@post tag). But we also have to remove clause B from the postcondition (the \@post tag) and add it to the \@assert tag since, otherwise, the Boolean expression evaluated is implicitly A and B rather than A or B: if B is evaluated alone in tag \@post, an exception is raised when A is false (in the \@assert), but not when both A and B are false.

An important limitation is that the use of an \@assert tag, even placed just before method exit, might not be able to detect a failure due to the execution of a fault in the return statement. This is the case if (part of) the contract involving the return expression is verified before the return statement. In Figure 4, an example shows a case where the postcondition is expressed in terms of result, the returned variable, and instrumented as an \@assert tag. Let us assume this method returns a probability between 0 and 1 and that a fault results into returning result+1 instead of result. Such a fault in the return statement cannot be detected by the postcondition instrumentation as:

- The Boolean expression result >= 0 and result <= 1 cannot be checked in the postcondition that does not have access to local variables or method return values;
- The assert tag is located before the return statement and hence cannot detect failures due to the fault in the return statement itself.

**Access to attribute values**

Other instrumentation issues concern the necessary built-in test support to enable sufficient observability of classes’s concrete state and the translation process from OCL to the language used by the instrumentation tool (Jcontract). Regarding the former issue, get() methods for non-public attributes as well as methods verifying the state invariant can be necessary and can greatly simplify the intrumentation of contracts. This is discussed and referred to as built-in test support in [4]. For example, in the ATM system we often use method Money::getValue() for verifying amounts (private attribute in Money) in contracts that are not in class Money. But this requires that some standard is followed when defining class interfaces such that get() and state invariant check methods are available.
Collections

The translation from an OCL expression of the contract to its implementation using a specific instrumentation tool language (in Jcontract: DbC, a language close to OCL) also requires some attention. Collections are used in OCL to link an object of a class to a set of objects of another class when the multiplicity of the association between these two classes is greater than 1. Then contracts can, for instance, involve assessing the size of a collection or selecting a particular object in a collection. Such collections are eventually implemented with data structures in the source code, thus requiring the translation of the OCL contract into an expression that handles these data structures. This is easily done in our study since contract assertions in DbC have access to attributes, method parameters, and local variables, and any Java code (e.g., a for statement browsing a data structure) can be used in a DbC contract assertion. In general, this issue should be carefully considered when selecting a tool to support contract instrumentation.

5 THE ANALYSIS OF CONTRACTS AS A TOOL FOR TESTABILITY

We describe in turn how we plan to estimate observability and diagnosability in our case study.

5.1 Contracts as a Substitute to Test Oracles

One of the most expensive activities during testing is the writing of test oracles in the test drivers. This is also a key activity as we want to ensure we detect all failures and do not waste time on false-positives,
that is test cases indicated as failed by the test driver that turn out to pass after further investigation. As discussed in previous work [4], contracts could be instrumented and used to serve as test oracles as they define conditions under which the system’s state can be considered corrupt. The advantage would be the use of instrumented contracts as test oracles, thus eliminating the need to develop test oracles in each test driver, for each test case. For this to be possible, we need effective instrumentation tools to generate, in the code, executable assertions based on contracts. Furthermore, the contracts need to be reused from Analysis or design, depending on the development methodology in place. They should not be subsequently refined or modified for the sake of improving the test oracles as the economic argument of leveraging them to improve testing would be less compelling.

The procedure we use to investigate whether contracts are effective test oracles is as follows:

- We define, on an example system, contracts in OCL. Those contracts are intentionally defined to be at the level of what could be expected during Analysis (e.g., in a Fusion Analysis). In other words they are based on application domain knowledge, not implementation knowledge. We also derive simplified versions of the contracts at their intermediate and low levels of precision.

- We implement the contracts using the language provided by a contract instrumentation tool. In our case study, we selected Jcontract from Parasoft [9], which provides a language (DbC) close to OCL. Contract assertions are defined through Javadoc comments, in order to ensure they do not affect the execution of the delivered system.

- We randomly seed faults in the system, using a set of predefined mutation operators for Java [11, 12]. When seeding faults, the probability of selecting a mutation operator over another depends on the characteristics of the code, e.g., extent of inheritance usage, that determine what the opportunities are to consider a specific operator. For each seeded fault, we generate four mutant programs (i.e., faulty program versions): with (at the three levels of precision) and without contract assertions (i.e., using test oracles in test drivers). Test oracles here are implemented by

---

7 Ideally, we would like such a language to be identical to OCL, to ease the transition from the analysis document to the code. But current tools, though improving, have not reached that stage yet.
comparing the mutant program outputs (i.e., an ordered set of strings) with the output of the correct program.

- We develop test cases based on a black-box technique, e.g., category-partition [19]. Our motivation is to ensure the functionalities of the system are reasonably exercised to provide contract assertions with the opportunity to execute seeded faults and detect failures.

- We run the test cases with the four mutant programs. We measure effectiveness of failure detection in each case and, for the instrumented system, check how many failures are caught by contract assertion violations. This is used to assess the suitability of contract assertions as a substitute to test oracles and potential improvements in observability due to contract instrumentation.

- When failures are not detected by contract assertions, we investigate the root cause and try to generalize from all instances.

5.2 Contracts as a Tool to Improve Diagnosability

Isolating faults in an OO system is time-consuming as functionality is typically distributed across many objects. In that context, contract assertions can be helpful in order to reduce the number of methods and source code statements one has to look at to locate faults when failures are detected. When contracts are instrumented, the detection point — the contract assertion that raises an exception — is expected to be significantly closer to the faulty statement(s) than an output statement making the failure visible to the user when contracts are not used. Moreover, since most methods in OO systems usually consist of a few lines of code, it is expected that contract assertions should be violated not long after faulty statements are executed. This should help failure diagnosis and fault isolation.

In order to assess how effective contracts are to improve testability, we need to define a diagnosability measure that enables objective, quantitative comparisons. Such a measure must be defined based on the Diagnosis Flow that determines the sequence of methods one has to investigate from the location in the program where a failure has been revealed (e.g., by the Oracle) to the location of the faulty statement(s). Note that this has been identified as one of the root causes that make debugging difficult [8]. We define
the diagnosis flow as a model of what would be common practice when testers detect a failure and search for faulty statements: They know where the failure has been revealed (i.e., the violated contract assertion or the output statement revealing the failure), and then they recursively browse the flow of methods executed from the method that contains the violated contract assertion or output statement, say method A, back to the clients of A. Admittedly, like any model, this is a simplification of reality where a developer may have so much expertise that she uses shortcuts to identify faulty statement(s). But such simplifications are necessary for performing a systematic, objective study of diagnosability.

We use the following rules to determine the methods that appear in the diagnosis flow:

**Starting method in the diagnosis flow:**

The starting method depends on the location of the raised exception. As illustrated in Figure 5 (Figure 5.a), an exception can be raised in a method (method $e$) at one of the following locations: (1) At method’s entry, i.e., the pre-condition or the invariant as evaluated at method’s entry is violated; (2) During the execution of the method$^8$ (an assertion is violated); (3) At method’s exit (the postcondition or the invariant as evaluated at method’s exit is violated). In case (1), the starting method in the diagnosis flow is the method that called method $e$, because the exception is raised before the execution of any statement of $e$. On the contrary, for cases (2) and (3), the faulty statement can be in method $e$ and the starting method in the diagnosis flow is therefore method $e$ itself. If we use oracles instead of contract assertions, the detection of the failure is an output statement. The starting method in the diagnosis flow is then the method that contains this output statement.

**Sequence of methods in the diagnosis flow:**

When a method is investigated, one starts the investigation at the beginning of the method and then looks at the different flows of control in the method. During that process, each time a method call is encountered, the called method is (recursively) investigated. When the end of the

---

$^8$ Jcontract, unlike other tools supporting contracts, enables the check of assertions in the body of methods.
method is reached, the investigation continues with the client method. The diagnosis flow stops when a faulty statement is discovered in a method. Note that, if a method is involved several times in the diagnosis flow, we assume that the method is not investigated more than once.

The diagnosability measure we associate with the diagnosis flow determines the distance, in terms of number of methods investigated (that appear in the diagnosis flow), between the detection point (the violated contract assertion or the output statement) and the faulty statement (where the mutation operator was applied). The measure can be further refined according to the complexity of the methods encountered, to give more weight to the more complex methods (e.g., with many control paths). This is an open issue that will be investigated in future work. However, though an approximation, we think the number of methods is a reasonable indicator since methods usually consist of a small number of lines of code.

The use of a sequence diagram defined during Analysis or Design (or its construction if it does not exist) can help the determination of the diagnosis flow, as well as its visualization. We use this representation in this article, both for examples (as in Figure 5.b) and for the case study.

![Sequence Diagram](image)

(a) Location of the violated contract  
(b) Diagnosis flow as a sequence diagram

Figure 5 Diagnosability measure: example

---

9 This is a simplification justified in the context of our study since we insert faults using mutation operators and each failure can be traced back to a unique fault.
According to this procedure, the diagnosis flows and the corresponding diagnosability measures for the example in Figure 5.b are determined as follows:

- If the exception is raised at the entry of method \( e \), the diagnosis flow begins with method \( a \). Then methods \( b \) and \( c \) are investigated in sequence. Method \( d \) is not investigated because the faulty statement is discovered before the call. The diagnosis flow is sequence \([a, b, c]\) and the diagnosability measure equals to 3.

- If the exception is raised during the execution of \( e \), but before the call to \( f \), the diagnosis flow begins with method \( e \). Method \( f \) is not investigated because the fault is revealed before its call in method \( e \). The diagnosis flow is sequence \([e, a, b, c]\) and the diagnosability measure equals to 4.

- If the exception is raised in \( e \) after the call to \( f \), then \( e \) starts the diagnosis flow, and \( f \) is investigated. The diagnosis flow is sequence \([e, f, a, b, c]\) and the diagnosability measure equals to 5.

Given our measure for diagnosability, the procedure we use to determine how effective contract assertions are to improve the diagnosis of failures is as follows:

- We use an example system, its associated contracts and test cases, as discussed in Section 5.1. The test case executions give (i) the mutant programs killed when contract assertions are used (at the three levels of precision), and (ii) the mutant programs killed when test oracles are used. Based on test executions, for all the mutant programs that are killed by both contract assertions and oracles, we determine the diagnosis flows and compute the diagnosability measures for both mutant versions with contract assertions and with test oracles only, respectively. Since several test cases may kill the same mutant, and since they can involve different outputs (when the oracles are used alone) or violated contract assertions (when contracts are used alone), we may obtain different diagnosability values for one mutant program. Therefore, we define the diagnosability associated with a mutant as the average diagnosability of all the test cases that kill it.

- We compare the distributions of diagnosability values for both mutant versions in order to determine whether contract assertions make a practically significant difference.
6  CASE STUDY DESCRIPTION

We first present the system we used as a case study and justify its selection. Then, we present our strategy for seeding faults and the experimental procedure we followed to investigate the benefits of contracts in terms of failure detection and diagnosability.

6.1  The ATM System

An Automated Teller Machine (ATM) system was selected as an example since it was deemed of adequate complexity and the application domain is intuitive, thus facilitating the definition of contracts. Furthermore, the system was not designed or coded by the authors and represents a complete set of functionalities. Furthermore, the implementation language is Java, for which a number of contract instrumentation tools exist. The ATM is composed of 21 classes (see class diagram in Figure 6), a variety of associations (including aggregation, inheritance, use dependencies), and the Java source code contains 2200 LOCs (without comments).
6.2 Seeding of Faults

Faults were seeded according to a set of mutation operators proposed both for procedural [14] and object-oriented programs [11, 12] (previously used in an experiment [13]). Thus, some of these mutation operators are general and can apply to all imperative programming languages (e.g., the substitution of two operators) whereas others are specific to object-oriented languages (e.g., the removal of an overriding declaration). While seeding our program with faults, we tried to cover all mutation operators in a balanced way given the constraints of the code. As a result we seeded 112 faults, covering 17 different mutation operators (see Figure 7). That number was deemed sufficient to observe significant
differences in fault detection and diagnosability. It is also important to note that, to avoid any positive bias in the results, faults were seeded randomly before the test cases were designed. The rationale is that when designing test cases it is very difficult to recall where faults were seeded whereas it is easier to “guess” whether a fault is likely to be found by a test strategy that was previously applied to a system and implemented in a test driver. Ideally, these two activities (seeding of faults and test case generation) should be performed by different persons, or faults should be automatically seeded according to the characteristics of the source code and a set of mutation operators [5].

Moreover, in order to obtain unbiased results, only faults that cannot be detected by the compiler and that can possibly be found by running test cases were considered. For example, if we take the example of the Access Modifier Changes (AMC) mutation operator [12], where an access mode is changed with another one (e.g., from private to public), it is difficult to imagine how a test case can uncover such a fault. We therefore did not use this particular mutation operator. Such faults should be addressed by systematic design and code reviews.

Figure 7 provides the distribution of seeded faults per mutation operator. The mutation operators are denoted by acronyms and have self-explanatory names: Argument Number Decrease (AND), Argument Order Change (AOC), Arithmetic Operator Replacement (AOR), Control Flow Disruption (CFD), Comparable Array Name Replacement (CNR), Constant Replacement (CRP), Do Statement End Replacement (DER), Hiding Field Variable Addition (HFA), Class Instance Creation Expression changes (ICE), Logical Connector Replacement (LCR), Method name Replacement Operator (MRO), method Parameter Order Change (POC), Relational Operator Replacement (ROR), Return Statement Replacement (RSR), Scalar Variable Replacement (SCR), Statement Deletion (SDL), Statement Swap Operator (SSO).

Most of these operators are not specific to the object-oriented paradigm (e.g., DER). The differences regarding the proportion of faults seeded per category is due to the characteristics of the system code under study. For instance, the larger the number of overriding methods, the larger the number of potential AND faults. We only have two AND faults in our case study since the ATM system includes only two instances of method overriding in class Money.
Figure 8 provides the distribution of seeded faults per class, as well as class sizes (by means of number of lines of code without comments). Due to the characteristics of the classes under study, only 11 classes out of 21 were seeded with faults. The class diagram shown in Figure 6 comes from Analysis (where we defined our contracts), and some of the classes were ultimately not implemented as Java classes in the source code, i.e., classes ATMMachineProfile, Card, and Account are implemented with arrays. Classes that represent the parts of an ATM machine (e.g., Keyboard) only represent the interface with hardware devices, and thus only contain simple methods as the device driver code is missing here. But classes that participate in the core logic of the system (e.g., ATM, Bank, …) show larger numbers of seeded faults.
6.3 Experimental Procedure

The procedure we used in our case study can be described as follows:

- We devise contracts in OCL (at three levels of precision), following clearly defined rules (see Section 4). These contracts are then instrumented using Jcontract (see Section 4.4), leading to three instrumented versions of the system.

- We seed faults according to clearly defined mutation operators in both instrumented and non-instrumented programs (see Section 6.2). We thus generate mutant programs for four program versions (oracles only, oracles plus contract assertions at three levels of precision).

- We run the non-instrumented versions of mutant programs and compute the fault detection effectiveness and diagnosability. These program versions only use test oracles (that check program outputs) to detect faults.

- We run the instrumented mutant programs (for all three levels of contract precision) and compute again the fault detection effectiveness and diagnosability, based on both instrumented contracts and oracles (the same drivers are used for instrumented and non-instrumented programs).
We compare the fault detection effectiveness and diagnosability of instrumented and non-instrumented mutant program versions, for all three levels of contract precision.

7 ANALYSIS RESULTS

In this section we provide a complete reporting of results. We first focus on the detection effectiveness of contracts with respect to faulty system states and then investigate how helpful they can be to support the diagnosis of failures.

7.1 The Detection of Failures

Recall that we are comparing the number of mutants killed among three versions of the ATM system including contract assertions and another version having only regular, hard-coded test oracles checking program outputs. We are looking at both the percentage of non-equivalent mutants killed and the subset of mutants killed by the contract assertions as opposed to the test oracles.

<table>
<thead>
<tr>
<th>Contracts</th>
<th>Equivalent Mutants</th>
<th>Live Mutants</th>
<th>Mutants Killed by Oracles</th>
<th>Non Equivalent Mutants Killed by Contracts</th>
<th>Equivalent Mutants Killed by Contracts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Without</td>
<td>7</td>
<td>6</td>
<td>99</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Highest</td>
<td>7</td>
<td>6</td>
<td>20</td>
<td>79</td>
<td>5</td>
</tr>
<tr>
<td>Intermediate</td>
<td>7</td>
<td>6</td>
<td>24</td>
<td>75</td>
<td>5</td>
</tr>
<tr>
<td>Lowest</td>
<td>7</td>
<td>6</td>
<td>25</td>
<td>74</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 5 Mutant Detection Results for the ATM system

The 6 live mutants represent less than 10% of non-equivalent mutants and are due to imperfect black-box testing. For each of the three contract precision levels, 79, 75, and 74 non-equivalent mutants are killed by contract assertions and this represents 80%, 76%, and 75% of non-equivalent mutants killed by oracles, respectively. So using contracts as a substitute to oracles would lead to less faults being detected but the difference might, under certain circumstances, make economic sense. Furthermore, using contracts at lower levels of precision results in some mutants not being detected but these differences are small (in the worst case, 5 additional mutants remain undetected). Note that this result was obtained
from contracts defined during the analysis and following precise guidelines (Section 4.1). It is
interesting to note that it is similar to the results reported in [2] where a mutant detection effectiveness of
87.5% was reported for class testing. The specifics of the contracts with which this result was obtained
are not known but we know their contract assertions were based in part on implementation information
and this may explain the slightly higher percentage they obtained.

It is worth noting (Table 5) that 5 of the 7 equivalent mutants were killed by contract assertions (at any
precision level). This therefore suggests that using oracles and contracts together improves fault
detection and this can be explained by the fact that contract assertions check intermediary object states
and can therefore detect additional errors. Let us take two examples of equivalent mutants killed by
contract assertions in the ATM. First, mutant 90 initializes attribute _currentCash in class
CashDispenser to a wrong value (a value different from zero), thus breaking the post condition of the
constructor. However, this attribute is never used before being set a value when the ATM is started. As a
consequence, the mutant cannot be killed by test cases. The second example, mutant 73, is more
complicated and is illustrated by Figure 9, which contains the (simplified) relevant parts of the source
code for classes Session and Transaction. In the original program, when
Session::doInvalidPINExtension returns INVALID_PIN when called in
Transaction::doTransactionUseCase, it sets the state of the session (attribute _state) to
ABORTED (i.e., a wrong PIN has been entered three times). In this situation, doTransactionUseCase
also returns INVALID_PIN in its call in Session::doSessionUseCase, resulting in the initialization
of the state of the session to ABORTED (again). As a consequence, mutant 73, which consists in setting
_state to a wrong value, has no effect on the result observed by oracles because the state is set twice.
However, the post-condition of doInvalidPINExtension is broken.
We also investigated the reason why some mutants could not be detected by contract assertions but were detected by test oracles. We identified five distinct mechanisms (see the Appendix A for complete examples) for which we report the number of instances we encountered:

- The order of parameter changed but the precondition on those parameters is the same and no computation is performed with them (1 instance).

- The contract was not precise enough to catch the failure (10 instances) and we checked it was not possible to define the contract at such a precision so as to find the fault. For instance, postconditions often verify whether values are within acceptable ranges, and it is not possible to perform a more refined check (e.g., a function that computes a probability returns a value between 0 and 1). If the
effect of the mutant still results in values that are within-range, then the postcondition check does not report a problem.

- The postcondition is checked once the execution of the method is completed. However, if the method itself raises an exception as the result of executing faults, and there is no catch block for that exception in the method, then the execution does not get the opportunity to check the postcondition (5 instances). A question is whether these cases should be included in our failure detection effectiveness percentage for instrumented mutant programs as assertions may not get the opportunity to catch the failures due to certain faults.

- Some mutants are related to constants being assigned incorrect values. However, when these constants get involved in the definition of contracts, the contract implementations are also faulty and cannot be violated in the mutant programs (3 instances).

- The mutant changes a message sent to one of the actors (i.e., the GUI if we refer to a human actor). We consider it more practical that messages sent to actors be the responsibility of an oracle that, for example, check output log files. Therefore, when output messages to actors were specified in the post-conditions, we chose not to instrument this aspect of the contract. This category should not be considered when looking at the effectiveness of contracts to kill mutants (1 instance).

7.2 The Diagnosability of Faults

We now investigate whether using contracts can help reduce debugging effort by facilitating the isolation of faults in object-oriented systems. We only consider here the faults that were detected by contract assertions as for the others it is obvious that no diagnosability improvement can be measured. We thus consider 79/75/74 mutants out of the 99 mutants killed by oracles, for the highest, intermediate, and lowest contract precision levels, respectively.

Average diagnosability results are presented in Figure 10. Without resorting to statistical inference testing, it is clear that the difference in diagnosability between the ATM system with contract assertions (at any level of precision) and the one without contract assertions is very significant as it approaches one order of magnitude. This should translate, in practice, into significant effort savings during debugging.
However, in order to assess such savings precisely, an experiment would need to be run in a realistic context with actual software developers. We can also see that simple contracts, at the lower level of precision, are sufficient to improve the average diagnosability, though on a slightly lower number of mutants.

<table>
<thead>
<tr>
<th>Assertions</th>
<th>Number of Mutants</th>
<th>Average Diagnosability</th>
</tr>
</thead>
<tbody>
<tr>
<td>With Oracle only</td>
<td>79 mutants</td>
<td>13.30</td>
</tr>
<tr>
<td>Highest Precision</td>
<td>79 mutants</td>
<td>1.35</td>
</tr>
<tr>
<td>Intermediate</td>
<td>75 mutants</td>
<td>1.36</td>
</tr>
<tr>
<td>Lowest</td>
<td>74 mutants</td>
<td>1.37</td>
</tr>
</tbody>
</table>

**Figure 10 Average Diagnosability**

But averages do not tell the whole story. Five cases of poor diagnosability can be observed when using contracts whereas a number of minimum diagnosability values (1) can be observed without contracts. Figure 11 shows diagnosability values (for both the contract and oracle cases) on the Y-axis and mutants are ordered (X-axis) according to their diagnosability when using hard-coded oracles. This allows us to compare results graphically and identify cases where diagnosability did not improve as much as expected. The poor diagnosability cases when using contracts correspond to five different mutation operators (POC, CRP, HFA, CFD, SCR). Note that two of them (POC and CRP) correspond to the two cases discussed above (mutants not killed by contract assertions) where mutants involve a compatible parameter order change and a constant value change. In these five cases, as a side effect and probably by chance, contract assertions get broken in other methods that are far away, in terms of diagnosis flow, from the methods where faults were seeded.
In our experiment, faults are mostly detected in the classes where they lie or are not detected at all (in their clients), except in five instances. For the highest precision oracles, the diagnosability is on average 1.35, is equal to 1 for 75 mutants (the fault is detected in the class that contains it) and lies between 2 and 10 for five mutants. Results are nearly identical for contracts of lowest and intermediate precision. This contradicts, to some extent, the results by [2] who estimated coefficient $c$ to average at $0.8^{10}$ (Section 3.1). Our results suggest that, in our system, $c$ is small as failures are detected by client classes’ contract assertions only in a couple of cases. This issue clearly needs to be further investigated.

---

$^{10}$The procedure followed to obtain that estimation is not provided in [2].
8 VALIDITY

Based on our sensitivity analysis, we have shown that no theoretical model can, with sufficient precision, tell us whether instrumenting analysis contracts is likely to be a good solution for automating test oracles. We therefore resorted to carefully designing and performing a case study.

Any case study, despite the substantial effort it involves, only represents one data point. Such a problem is not specific to software engineering case studies but is encountered in many other fields such as social sciences. So the usual question is: to which extent are the results obtained externally valid, that is generalizable? Though this is inherently a difficult question, one way to alleviate the problem is to look beyond the numbers to understand the mechanisms by which we obtain the quantitative results we report. In our study, we investigated every mutant that remained undetected by contract assertions or oracles. We also looked at every mutant that showed a poor diagnosability. We identified, across mutants, the common mechanisms which would explain poor observability (Section 7.1) or diagnosability (Section 7.2), and assessed whether such mechanisms should be commonly encountered in other programs as well. This gave us a good insight into whether our results were likely to be encountered in many other situations, e.g., different software systems.

Following traditional testing research, we identified a general fault taxonomy (mutation operators) for Java programs and made sure we randomly seeded a wide variety of faults (Section 6.2) in the program to be tested. We used a well-known traditional black-box software technique (Section 6.3) to generate test cases. We also made sure the program selected (Section 6.1) for experimentation had no specific peculiarity that could hinder us from getting meaningful results. We also ensured that contracts were defined using well-defined rules, at a realistic level of detail, so that the results could be interpreted in a well-defined context.
9 CONCLUSIONS

From the results of our case study, it seems clear that contract assertions detect a large percentage of the failures, ranging from 75% to 80%, depending on the level of precision of the contract definitions. If we include the 5 instances where run-time exceptions are raised in the program\textsuperscript{11}, the percentage of failures detected rises to 85%, 81% and 90% for the three levels of precision, respectively. In other words, in roughly 80 percent of the cases, contracts are good enough substitutes to hard-coded oracles in test drivers. Though defining and instrumenting very precise contracts helps with the detection of failures, the difference does not seem practically significant and probably does not justify the cost entailed with instrumenting precise contracts.

One issue is that to find 20% of the faults we still require test oracles to detect the corresponding failures and, more importantly, we would need to know beforehand which test cases require hard-coded test oracles. If this were the case, we could save a large percentage of the effort of defining and coding them. Future research has to provide a way to identify, during the coding of the test driver, the test cases that will need the hard-coded test oracles. One avenue to be explored is to identify weak contracts through fault seeding (mutation) and require oracles for those test cases that exercise weak contracts. Another strategy worth investigating is to compensate for the small percentage of failures that remain undetected when not using hard-coded oracles with a larger number of test cases. This may still be economically viable as the corresponding test drivers are less expensive to develop when they do not contain oracles.

In terms of instrumentation, our experience is that with a tool like Jcontract, precise implementation rules can be devised to facilitate the transition from Analysis contracts (in OCL) to Javadoc contracts (in DbC). The complexity of the instrumentation depends, of course, on the complexity of the contracts, e.g., whether they manipulate sets and therefore require the traversing of complex data structures. As a consequence, an important future research direction is to investigate the impact of using approximate, simpler contracts instead of the precise contracts instrumented here. This is especially relevant in light of the sensitivity analysis results, suggesting that contracts need not be perfect to be effective.

\textsuperscript{11} They cannot be detected by instrumented contracts, but provide information regarding where the exception was raised.
Another important result that shows to be very strong and of high practical significance is that diagnosability, as we measure it, improves nearly an order of magnitude between mutant programs without contract assertions and the ones with contract assertions, regardless of the level of precision. This suggests significant savings during debugging, as faults will be much easier to locate if contracts are being instrumented and checked during execution.

In conclusion, our overall results suggest that instrumented contracts have a strong potential in terms of decreasing the cost of testing. Furthermore, they do not require to be defined and instrumented at a high level of precision to be useful. Those results are the outcome of a carefully planned and designed study and can be replicated on other system examples, using well-defined procedures and a commercial tool. Replication is essential in order to confirm the results we obtained here.

ACKNOWLEDGEMENTS

This project was supported by a NSERC CRD grant in collaboration with Mitel Networks, within the framework of CSER. It is part of the TOTEM project which focuses on the testing of object oriented systems, at all testing levels (http://www.sce.carleton.ca/Squall/Totem/Totem.html). We are also grateful to Daniel Amyot for his review of earlier drafts of this report.

REFERENCES


APPENDIX A: EXAMPLES OF MUTANTS NOT KILLED BY CONTRACT ASSERTIONS

In this Appendix, we illustrate the five distinct mechanisms that can explain why mutant programs were not killed in our ATM case study (see Section 7.1). We provide a detailed example for each of the five cases.

Situation 1: The order of parameter changed but the precondition on those parameters is the same and no computation is performed with them.

Mutant 1 (mutation operator AOC) consists in swapping two parameters (cardNumber and serialNumber) in a call to method printReceipt() (class ReceiptPrinter). Method printReceipt() only verifies the range of values of these two parameters and issues a receipt of the transaction (no computation). Since the two parameters have the same range, the mutant program is not killed.

```java
// In class ATM
/** @pre serialNumber>0
 * @pre balance.getValue() >= availableBalance.getValue()
 */
public void issueReceipt(int cardNumber,int serialNumber,String description,Money amount,Money balance,Money availableBalance) {
    String [] receipt = new String[7];
    _receiptPrinter.printReceipt(_number, _location, serialNumber, cardNumber, description, amount, balance, availableBalance,receipt );
    // this is the mutant: the two parameters are swapped
    for (int i = 0; i < 7; i ++)
        _display.receiptPrint(receipt[i]);
}

// In class receiptPrinter
/** @pre cardNumber >=0
 * @pre serialNumber >0
 * ...
 */
public void printReceipt(...) {
    // only outputs from the parameters
}
```

Figure 12 – Mutant program not killed: example of Situation 1
Situation 2: The contract was not precise enough to catch the failure.

Mutant 57 (mutation operator MRO) consists in replacing the call to `getAmountEntry()` (method in class ATM) with a call to method `startupOperation()` (method in class ATM). Statement `_amount = _atm.getAmountEntry()` is changed into `_amount = _atm.startupOperation()`. This change is done in method `getTransactionSpecificFromCustomer()` in class `TransferTransaction`, which asks the customer to choose two accounts (for the transfer) and an amount. The postcondition does not (and cannot) involve the amount since it is entered by the user who can provide any number: whether the amount is available is checked elsewhere, and whether the amount is within acceptable range (i.e., the amount is a positive integer) is the responsibility of class `Money`'s invariant. Since `_atm.getAmountEntry()` and `_atm.startupOperation()` return an amount which can be within expected range (positive integer), the mutant is not killed.

```java
/** @post $result == Status.SUCCESS */
public int getTransactionSpecificsFromCustomer() {
    _fromAccount = _bank.chooseAccountType("transfer from", _atm);
    _toAccount = _bank.chooseAccountType("transfer to", _atm);
    _amount = _atm.startupOperation();  // mutant
    return Status.SUCCESS;
}
```

Figure 13 – Mutant program not killed: example of Situation 2

Situation 3: An exception is raised before the evaluation of the contract assertion.

Mutant 59 (mutation operator CRP) changes a case label in class `WithdrawalTransaction`: in method `getTransactionSpecificFromCustomer()`. The case label “case 5: …” is changed to “case 8: …” (the statements executed are not changed). The corresponding switch creates an instance of class `Money` according to the amount entered by the customer. Then, the amount is checked against the available amount in the selected account. In the mutant, the `Money` instance corresponding to amount 5 is not created, and since there is not default label in the switch, the reference used in the check is null, thus raising an exception. The postcondition does not have the opportunity to execute.
Situation 4: Mutants are related to constants being assigned incorrect values.

Mutant 45 (mutation operator CRP) changes the value of constant variable `RUNNING` in class `Session`. The original value is 0 and the new value is 1. However, `Session`’s invariant, the pre- and postconditions of `Session`’s methods, as well as the source code, use the name of the constant (`RUNNING`) instead of the value. Then, the mutant is not killed.
Figure 15 – Mutant program not killed: example of Situation 4

Situation 5: The mutant is related to messages sent to actors

Mutant 49 (mutation operator RSR), in method Session::doFailedTransactionExtension(), changes a string. This change eventually has an impact on what is displayed on the screen. As explained above, we considered that verifying messages sent to actors can be better performed with an oracle (that check output log files) than with instrumented contracts.

```java
/**
 * @post _state==RUNNING
 * @post _PIN == 0
 * @post _atm == atm
 * @post _bank == bank
 * @post _cardNumber == cardNumber
 * @post _currentTransaction == null
 **/
public Session(int cardNumber, ATM atm, Bank bank) {
    _cardNumber = cardNumber;
    _atm = atm;
    _bank = bank;
    _state = RUNNING;
    _PIN = 0;
    _currentTransaction = null;
}
```

Figure 16 – Contracts for method Session::doFailedTransactionExtension()

Mutant 49 changes string "Envelope not deposited - transaction cancelled" with string "Transaction OK".
/**
 * @pre reason == Status.DAILY_WITHDRAWL_LIMIT_EXCEEDED || ... || reason == Status.TOO_LITTLE_CASH
 **/
public boolean doFailedTransactionExtension(int reason) {
    switch(reason) {
    case Status.TOO_LITTLE_CASH:
        return _atm.reportTransactionFailure("Sorry, there is not enough cash available to satisfy your request");
    case Status.ENVELOPE_DEPOSIT_TIMED_OUT:
        return _atm.reportTransactionFailure("Envelope not deposited - transaction cancelled");
    default:
        return _atm.reportTransactionFailure(_bank.rejectionExplanation(reason));
    }
}
APPENDIX B: CONTRACTS AT DIFFERENT LEVELS OF PRECISION

We provide below three post-conditions for method `Bank::initiateWithdrawl()`, one at each level of precision, in decreasing order of precision and according to the rules provided in Section 4.2.1.

```
post: if not self.card->exists(_cardNumber = cardNumber) then
    result = Status.UNKNOWN_CARD
else
    if PIN <> self._currentTransactionCard._PIN then
        result = Status.INVALID_PIN
    else
        if not self._currentTransactionAccount._available then
            result = Status.NO_SUCH_ACCOUNT
        else
            if from = SAVINGS then
                result = Status.CANT_WITHDRAW_FROM_ACCOUNT
            else
                if self._currentTransactionAccount._availableBalance.less(amount) then
                    result = Status.INSUFFICIENT_AVAILABLE_BALANCE
                else
                    if (MAXIMUM_WITHDRAWL_AMOUNT_PER_DAY.less(Money.add(self._currentTransactionCard_withdrawlsToday, amount)) then
                        result = Status.DAILY_WITHDRAWL_LIMIT_EXCEEDED
                    else
                        result = Status.SUCCESS
                        and
                        _currentTransactionAmount.equals(amount)
                        and
                        newBalance.getValue() =
                            self._currentTransactionAccount._currentBalance.getValue() -
                            amount.getValue()
                        and
                        availableBalance.getValue() =
                            self._currentTransactionAccount._availableBalance
                        .getValue() - amount.getValue()
                    endif
                endif
            endif
        endif
    endif
endif
```

Figure 18 – Postcondition for `Bank::initiateWithdrawl()` at the highest precision level
post: if 
    (self.card->exists(_cardNumber = cardNumber))
    and
    (PIN == self._currentTransactionCard._PIN)
    and
    (self._currentTransactionAccount._available)
    and
    (from <> SAVINGS)
    and
    (not self._currenttransactionAccount._availableBalance.less(amount))
    and
    (not MAXIMUM_WITHDRAWL_AMOUNT_PER_DAY.less(Money.add(
        self._currentTransactionCard_withdrawlsToday, amount)))
  ) then
  result = Status.SUCCESS
  and
  _currentTransactionAmount.equals(amount)
  and
  newBalance.getValue() =
    self._currentTransactionAccount._currentBalance.getValue() -
    amount.getValue()
  and
  availableBalance.getValue() = self._currentTransactionAccount._availableBalance
    .getValue() - amount.getValue()
else
  result = Status.UNKNOWN_CARD
  or
  result = Status.INVALID_PIN
  or
  result = Status.NO_SUCH_ACCOUNT
  or
  result = Status.CANT_WITHDRAW_FROM_ACCOUNT
  or
  result = Status.INSUFFICIENT_AVAILABLE_BALANCE
  or
  result = Status.DAILY_WITHDRAWL_LIMIT_EXCEEDED
endif

Figure 19 – Postcondition for Bank:initiateWithDrawl() at the intermediate precision level
post: result = Status.UNKNOWN_CARD
    or
    result = Status.INVALID_PIN
    or
    result = Status.NO_SUCH_ACCOUNT
    or
    result = Status.CANT_WITHDRAW_FROM_ACCOUNT
    or
    result = Status.INSUFFICIENT_AVAILABLE_BALANCE
    or
    result = Status.DAILY_WITHDRAWAL_LIMIT_EXCEEDED
    or
    ( result = Status.SUCCESS
        and
        _currentTransactionAmount.equals(amount)
        and
        newBalance.getValue() =
            self._currentTransactionAccount._currentBalance.getValue() -
            amount.getValue()
        and
        availableBalance.getValue() =
            self._currentTransactionAccount._availableBalance.getValue() -
            amount.getValue() )

Figure 20 – Postcondition for Bank:initiateWithdraw() at the lowest precision level