Contract Driven Development = Test Driven Development – Writing Test Cases

Andreas Leitner, Ilinca Ciupa, Manuel Oriol, Bertrand Meyer
Chair of Software Engineering
ETH Zurich, Switzerland
{firstname}.{lastname}@inf.ethz.ch

Arno Fiva
Chair of Software Engineering
ETH Zurich, Switzerland
fivaa@student.ethz.ch

ABSTRACT

Although unit tests are recognized as an important tool in software development, programmers prefer to write code, rather than unit tests. Despite the emergence of tools like JUnit which automate part of the process, unit testing remains a time-consuming, resource-intensive, and not particularly appealing activity.

This paper introduces a new development method, called Contract Driven Development. This development method is based on a novel mechanism that extracts test cases from failure-producing runs that the programmers trigger. It exploits actions that developers perform anyway as part of their normal process of writing code. Thus, it takes the task of writing unit tests off the developers' shoulders, while still taking advantage of their knowledge of the intended semantics and structure of the code. The approach is based on the presence of contracts in code, which act as the oracle of the test cases. The test cases are extracted completely automatically, are run in the background, and can easily be maintained over versions. The tool implementing this methodology is called Cdd and is available both in binary and in source form.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools (e.g., data generators, coverage testing)*

General Terms

Verification

Keywords

unit testing

1. INTRODUCTION

Unit tests are an important instrument in software engineering. This is a generally recognized fact, but it does not change the cumbersome, time-consuming, and boring nature of the process of writing meaningful unit tests.

Consequently, researchers have studied ways to reduce this burden on the developer, while maintaining or improving the quality of the testing process. One possible solution is automated testing, illustrated in tools like Agitator [4], Parasoft's Jtest [1], AutoTest [13], Korat [5], TestEra [12], DSD-Crasher [6], Eclat [20], Symstra [24], DART [9]. Such tools bring a great advantage by the degree of automation that they provide, but they do face several problems:

- Automated testing strategies cannot make up for the insights that a human tester has into the semantics of the software under test and the relationships between the different units.
- In the absence of explicit, executable specification tools cannot distinguish between meaningful and meaningless input data.
- The quality of the generated tests can be estimated by different means and measures (such as code coverage, mutation testing, number of bugs found, time to first bug, proportion of the fault-revealing tests out of total generated tests, etc), but the exact measures to use or combination thereof depends on the characteristics of the project under test and is very hard, if not impossible, for a tool to determine automatically.

This paper shows how to reduce the burden of writing test cases without interfering with the programmer, while still leveraging the insights that he has into the semantics, structure, and possible weak spots of the software. The starting point of this work is the observation that developers actually create and run test cases even if they do not create comprehensive test suites. A developer typically adds some features to a program and then runs the program in such a way that the new feature is used. By placing assertions along the way and by watching the output of the program in general, he checks whether the program works correctly. A developer triggers the execution of a newly added feature via one or both of the following actions:

- Providing the right input.
- Changing parts of the program to force a certain path to be taken.

Each execution of the program in such a way tests certain aspects of the program. During the lifecycle of a program many such implicit test cases are created and run. One test case evolves into the next, often being an only slightly different variant of its predecessor. These implicit test cases are created by humans and do not face the problems mentioned above that automated synthesizers face. Such test cases have a very serious drawback however: they are implicit; usually they exist only for one or very few runs and cannot be kept for later automatic re-execution because:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ESEC/FSE'07, September 3–7, 2007, Cavtat near Dubrovnik, Croatia.
Copyright 2007 ACM 978-1-59593-811-4/07/0009 ...$5.00.

- If the developer had to provide inputs, the test case cannot be rerun without the developer providing the inputs again. This requires manual intervention and the developer needs to remember the exact inputs.

- If the developer changed parts of the program to force a certain path to be taken, then this change is unlikely to persist as it limits the generality of the application. Usually such a change is undone or altered yet again to create the next implicit test case.

Such test cases are easy to run and create and, since they are not permanent, don't need any maintenance. These are the most likely reasons why developers write and use them so often.

This paper presents a method that captures these implicit test cases (including their oracle) and makes them explicit and persistent. The captured test cases do not require user input, are stable with regards to system evolution, and are efficiently minimized (and hence their execution time is improved). Without further constraints on the development method the resulting test suite will provide a regression suite for mistakes made by the developers in the past. If the developer, before implementing a feature, writes its contract and runs the application the resulting test suite will have similar properties to a test suite resulting from test driven development. In general the developer chooses which test cases should be created by running the application with the corresponding inputs.

The *Cdd* tool[1] is an implementation of this method. The acronym expands to *Contract Driven Development*. Cdd targets Eiffel code, because Eiffel natively supports contracts and real world source code equipped with contracts is broadly available. Cdd is available both in binary and source, and was integrated into the EiffelStudio development environment. Cdd observes program executions and, when a failure occurs, Cdd detects the last uninfected state and takes a snapshot of this state. This snapshot is then recreated and serves as the starting point of the extracted test case. Cdd chooses the time at which it takes this snapshot so that it is early enough for the state not to be infected but also late enough to reduce execution time. Also, in order to make the test case more robust with regards to system change, the snapshot does not include that part of the state that is irrelevant for reproducing the failure.

The rest of this paper is organized as follows. The next section motivates the approach. Section 3 gives the intuition behind how Cdd works in practice with the help of a use case. Section 4 explains the main abstractions behind Cdd, namely traces, failures, and test cases. Section 5 presents the implementation, Section 6 outlines future work, and Section 7 concludes.

2. MOTIVATION

Writing unit tests during the development of software systems brings obvious benefits. Although developers are aware of these benefits, they still write only very minimal unit tests. In order to find the reasons for this, we conducted a small scale study on Computer Science students from the ETH Zurich, asking them various questions about their unit-test-writing habits. Their degree of experience in writing software varied widely, from 6 months to 9 years. So did the number of software projects that they had worked on until then, from 1 to 5. 45% of the students said that they never wrote a unit test case before the implementation and 36% do it only very seldom. After implementation, only 18% of the students always write unit tests. 54% do it very often. We also asked the students to rank (on a scale from 1 to 5, where 1 represents total disagreement and 5 total agreement) the causes that prevent them from writing unit tests. The causes and associated average ranks were:

- "Writing unit tests takes too much time": 4.4
- "It takes too much effort to maintain the unit tests": 3.6
- "Writing unit tests is too much effort for the provided benefits": 3.2
- "It takes too much time to run the unit tests": 2.1
- "Unit tests are not useful": 1.8

The results of our study show that the time and effort involved in writing and maintaining unit tests are the most often occurring causes for the developers' dislike of unit testing, as also indicated by other studies. Various research groups have tried to tackle these problems from different stand points. Many approaches try to take the burden off the developers' and testers' shoulders by promoting test *automation* as a solution. Some fully automated or close to full automation testing tools are already available; among them are DSD-Crasher [6], DART [9], FindBugs [10], AutoTest [13], Jartege [15], Eclat [20], Symstra [24], JTest [1], Agitator [4], and Java PathFinder [23]. All these tools have the potential of being a great support to developers and testers, but they lack the insights into the semantics of the tested applications that a human has and hence are very likely to miss bugs that a human would find easily. This paper presents an approach which aims at filling this gap that automated testing leaves open: while our tool also does not get into the way of the developer, it leverages actions that the developers perform anyway as part of the process of writing software.

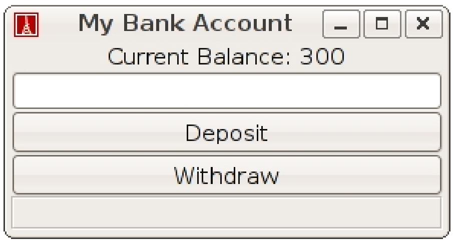
Other approaches also rely on this principle of least interference in established development processes. This is the case for the Agitator [4] tool (currently called AgitarOne). While this tool achieves a high degree of automation, it also allows the developer to improve the testing process by interacting with the tool in a way which only slightly affects his work flow.

Yet other research directions try to shift this burden of writing tests onto other activities. For instance, several research groups have investigated test generation from test specifications [3], [18], from formal software specifications [14] and from specification in the form of (abstract) state machines [8]. By introducing parameterized unit tests, Tillmann et al. [22] simplify the problem of automated input data synthesis and allow the developer to write more expressive oracles.

Several recent publications discuss the relationship between Design by Contract and Agile methods: Ostroff et al. [19] highlight the complementary nature of the techniques and Feldmann [7] shows the interplay of contracts and refactorings.

The relationship between contracts and test driven development is of particular interest: developing with contracts has the advantage of making use of a practical, lightweight, and executable form of specification that is able to express more about the intended semantics of the program than a finite number of test cases can. However, test cases are automatically executable; hence, they can be used for instant and continuous verification. Programs equipped with contracts do not have this property, because they lack concrete instances satisfying the preconditions of the methods under test. While the preconditions implicitly specify all valid inputs, automatically finding actual instances that fulfill these preconditions require tools such as constraint solvers or model checkers and involve translation of the program from the implementation level to a more abstract level and back. Satisfying the preconditions present in real software systems on the implementation level is beyond the

[1]http://eiffelsoftware.origo.ethz.ch/index.php/CddBranch/



Current Balance: 300

Figure 1: Screenshot of the bank account application. User is able to deposit and withdraw money and check his balance.

```
class BANK_ACCOUNT
inherit
  ANY
    redefine
      default_create
    end
feature
  default_create
    do
      balance := 300
    end
  balance: INTEGER
  deposit (an_amount: INTEGER)
    do
    ensure
      balance_increased: balance > old balance
      deposited: balance = old balance + an_amount
    end
  withdraw (an_amount: INTEGER)
    do
      balance := balance − an_amount
    ensure
      balance_decreased: balance < old balance
      withdrawn: balance = old balance + an_amount
    end
  ...
invariant
  balance_not_negative: balance >= 0
end
```

Listing 1: Main class of running example, representing a bank account.

capabilities of today's tools. The automated test case extraction described in this paper is able to provide this missing part and hence unify the advantages of contracts (thorough specification) and test cases (concrete and automatically verifiable).

With the contract driven development method, the developer is freed from the task of writing explicit test cases, but required to provide contracts instead. Contracts do provide benefits besides the ability to extract test cases: contracts allow for more precision during design, they serve as documentation throughout the lifecycle by more clearly specifying the semantics of interfaces and they increase chances of detecting failures closer to their source.

3. A USE CASE

To give the intuition behind our proposed method and the tool that implements it (Cdd), this section presents an example of practical use of the tool during the development of an application.

Our approach builds on the work on continuous testing [21], by trying not to affect the developer's work flow, but rather to exploit actions that the developer performs as part of the normal development process. An additional advantage of this approach is that it

```
class TEST_CASE_1
feature
  test
    local
      ba: BANK_ACCOUNT
    do
      ba := new_object ("BANK_ACCOUNT")
       set_field (ba, "balance", 300)
       check_invariant (ba)
      ba.deposit (30)
    end
end
```

Listing 2: Class *TEST_CASE_1*, automatically extracted from run of application that deposited money.

```
class TEST_CASE_2
feature
  test
    local
      ba: BANK_ACCOUNT
    do
      ba := new_object ("BANK_ACCOUNT")
       set_field (ba, "balance", 300)
       check_invariant (ba)
      ba.withdraw (20)
    end
end
```

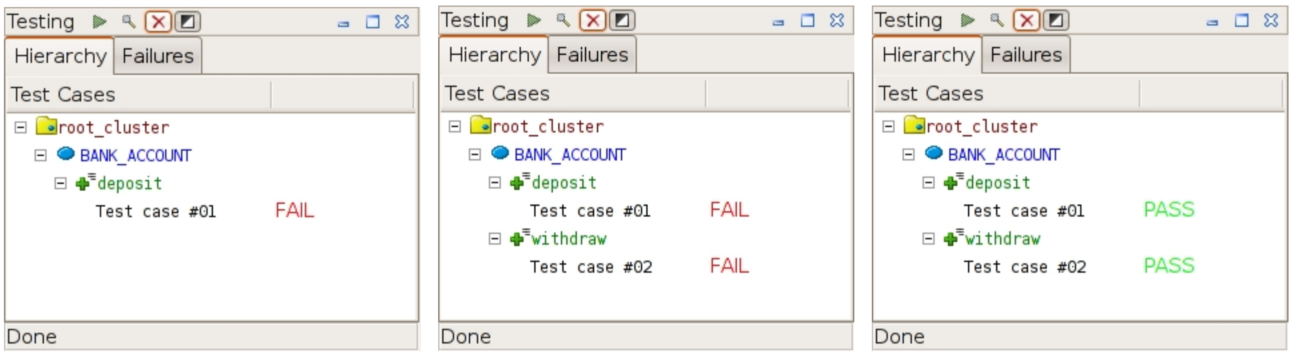
Listing 3: Class *TEST_CASE_2*, automatically extracted from run of application that withdrew money.

creates unit test suites for those developers that did not intend to write unit tests (due to time constraints or other reasons). While such test suites might not be complete, they are likely to provide a good foundation for regression testing, since every test case from the suite proved to fail at least once during the development cycle.

Let us consider an application written in Eiffel providing the means to deposit and withdraw money from a bank account. Listing 1 shows the main class of this application and Figure 1 shows a screenshot of the running application. In addition to class *BANK_ACCOUNT* (shown in Listing 1), which implements the main business concept of the software system, the actual application also contains:

- Class *MAIN_WINDOW*, which represents a GUI window showing the current account balance and allowing the user to enter an amount that he can then deposit or withdraw.
- Class *INTERFACE_NAMES*, which contains some global application constants.
- Class *APPLICATION*, which serves as the entry point of the application. It creates a bank account and a main window, passes the account to the main window, displays the main window, and starts the event loop.

Note that this example represents an application currently under development; it contains both incorrect and unfinished code. However, the developer tests the application by launching it as it is. He starts out with an empty unit test suite. He runs the application from within the debugger of his IDE (with Cdd support installed) and enters through the GUI that he wants to deposit 30 EUR. The GUI invokes the method *deposit* of class *BANK_ACCOUNT*, which throws a postcondition error and stops the application. As usual, the debugger indicates the line of the violation, the current stack frame



(a) After one failure (b) After two failures (c) After fixes applied

Figure 2: Test-cases automatically extracted from failures (Screenshot of test-case-window as integrated into EifelStudio IDE)

and the values of the variables in scope. Without the Cdd extraction mechanism, the developer would have to fix the bug immediately or the failure information would be lost. When the Cdd extension is installed, the test case extractor becomes active when a failure is observed and extracts, saves, and runs a test case in the background automatically. This test case is added to the previously empty test suite, which now looks as shown in Figure 2(a). The extracted test case referenced in this window is depicted in Listing 2. The actual test case does not provide an oracle, since the postcondition takes over this responsibility.

The reason why this failure occurs is that there is no implementation yet for method *deposit*. This is very much in the spirit of test driven development, where a test case (here only the contract) is written before the implementation.

As this example shows, the developer has to provide the inputs triggering the postcondition violation only once; then Cdd:

- Automatically extracts a test case.
- Minimizes it to that part of the application relevant for the failure.
- Frees it from external state (the GUI) and non-determinism (the user input).

Similarly, had the user circumvented the GUI programatically and hard-coded somewhere an invocation of *deposit*, Cdd would have extracted the same test case.

Since the failure is now reproducible via a test case, the developer no longer has to fix the fault immediately. He can go on and test another aspect of the application. For example, he can try to withdraw something from the bank account. If he does that, the debugger will once again stop the application and signal a fault in the postcondition. In this case the method was implemented correctly, but the postcondition contains an error. Cdd again automatically extracts a test case (depicted in Listing 3) for this failure. Then it adds this test case to the test suite, and compiles and runs it in the background. The test case status window is hence updated to show two failing test cases 2(b).

Suppose the developer now adds a correct implementation for method *deposit* and fixes the postcondition of method *withdraw* . Since Cdd employs continuous testing [21], the test cases are flagged with `PASS` automatically, as shown in the test case status window (Figure 2(c)).

4. MODEL

This section explains how test cases are extracted from failures. The first part of the section introduces the notions of trace, failure, and failure-recipient. The second part explains how test cases are represented in this model, how their oracle works, and how test cases can be executed and extracted from failures. The section concludes with an overview of possible applications to debugging.

Note that throughout the section each notion is presented through the use of mathematical functions that return the needed information, in order to keep notions as simple and language-agnostic as possible.

4.1 Traces

The test case extraction process is based on abstractions of traces of programs. Traces are what the developer produces when running the program in the debugger of the IDE.

The trace abstraction is based on a tree that captures what called what. Every node in the called_by tree is an instruction invocation, i.e. the invocation of an instruction at a given point in time during the execution that produced the trace at which we are looking. An instruction invocation is a pair of an instruction and the context in which it was executed. The called_by-tree only knows the following three kinds of instructions:

- Object creation (including data allocation and constructor execution).
- Method call.
- Delegate call.

The purpose of the tree is to enable test case abstraction and not to model all details of a trace. This is why only the above three kinds of instructions need to be looked at.

An example graph can be seen in Figure 3. This graph is based on a trace of the bank account example introduced in Section 3.

Listing 4 provides some more details of the bank account application introduced previously. We use this example throughout this section. Class *APPLICATION* contains the main event loop (which we consider to be the application's entry point for our discussion) in method *event_loop*. This method is responsible for calling the corresponding subscribers for each observed event. The method uses the Eiffel agent mechanism (which is similar in intent to the C# delegate mechanism), where each event keeps a list of its subscribers. A subscriber is just another method (that must have been registered before with the event). The *event_loop* method consists of two nested loops: the outer loop is executed once for every event, while the inner loop iterates over the subscribers of each event and calls these subscribers. Methods *deposit_amount* and *withdraw_amount* respectively subscribe to the events associated to the two buttons of the application ("deposit" and "withdraw", as seen in Figure 1). These methods in turn read the amount entered via a text entry box and then call the corresponding method from class *BANK_ACCOUNT*.

```
class APPLICATION
feature
 ...
  event_loop
    do
      ...
      from
      until
        should_quit
      loop
        wait_for_event
        from
          ev. subscribers . start
        until
          ev. subscribers . after
        loop
          ev. subscribers .item. call
          ev. subscribers .forth
        end
      end
    end
 ...
end
class MAIN_WINDOW
feature
   ...
  amount: TEXT_FIELD
  account: BANK_ACCOUNT
  deposit_amount
    do
      account. deposit (amount. to_integer )
    end

  withdraw_amount
    do
      account.withdraw (amount. to_integer )
    end
 ...
end
```

Listing 4: Partial source for example application. It shows the root class (*APPLICATION*) and the class representing the main window (*MAIN_WINDOW*)

The context of an instruction invocation contains the program state in which the instruction has to be executed, plus the bindings of the instruction. For example a method call requires one target-object and one object per argument. Let I be the set of instructions and C the set of contexts, then the set of instruction invocations is $I \times C$. An instruction invocation is said to be well formed if its context is such that the instruction can be executed from it without any syntax or typing errors:

$$\text{wellformed} : I \times C \quad \rightarrow \quad \mathbb{B}$$

Every instruction invocation executes on a target object. In the case of a method invocation, the target can either be explicit (qualified call) or implicit (unqualified call). In the case of a creation instruction the object being created serves as the target object. Let O be the set of objects. Then the signature of target is:

$$\text{target} : C \quad \rightarrow \quad O$$

The edges of the called_by-tree indicate which instruction invocation was called by which other instruction invocation. Let $< i1, c1 >$ and $< i2, c2 >$ be two invocations. Then $< i2, c2 >$ is called by $< i1, c1 >$ if and only if $i1$, while executing in context $c1$, directly (i.e. not indirectly via some other instruction invocation) invokes $r2$ in $C2$:

$$\text{called\_by} : I \times C \quad \rightarrow \quad I \times C$$

Any node in the tree can potentially trigger a failure, i.e. the execution of the instruction invocation directly triggers a failure. Failures occur due to a contract violation, method call on void target, operating system signals or other kinds of exceptions. Each programing language will have its own set of causes. For Eiffel the list is given in the Eiffel ECMA standard Section 8.26.1.

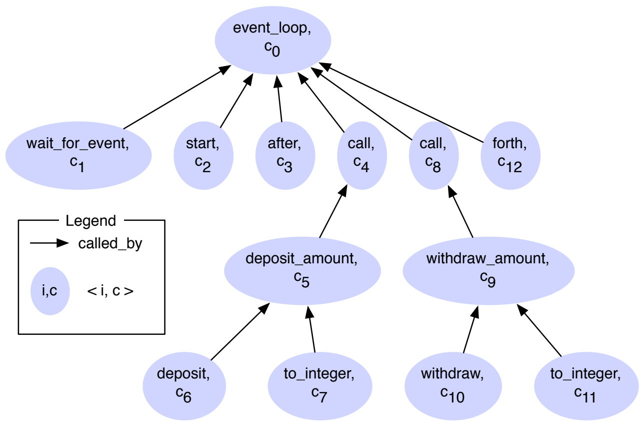


Figure 3: called_by**-tree showing which methods in what contexts called which other methods and in what context.**

In the presence of contracts every failure not only has an origin (the instruction invocation that immediately triggered the failure), but also a recipient. Intuitively the recipient is the method responsible for the failure. In most cases the recipient and origin are the same instruction invocation.

For Eiffel the recipient is defined in the ECMA Standard Section 8.26.10. The semantics of recipient is extended to not only mean the receiving method, but also its context. With F being the set of failures, the signature of recipient becomes:

$$\text{recipient} : F \quad \rightarrow \quad I \times C$$

4.2 Test Cases

In the present work, a test case is a particular (and hence deterministic) invocation of an instruction and the corresponding contracts (which serve as oracles).

At first, the notion of an invocation as a test case might seem too restrictive. Tools from the xUnit family (jUnit, nUnit, pyUnit, Gobo Eiffel Test, VSUnit, etc.) share the convention of having test methods contained in test classes. Test cases often consist of many instructions involving control flow, object creation, method invocation, and assert instructions. Traditional test methods must be created argument-less and deterministic. The developer has to provide the corresponding set-up and arguments for the element under test, turning to mock-objects when the set-up becomes too complicated. The instructions used here are perfectly capable to represent such test cases. A method call that first creates the test object, invokes the set-up method and finally invokes the test method.

Conventionally, the oracle for unit tests is provided by certain library calls or special keywords (e.g. *assert*). Similar to many recent approaches, the present work relies on the presence of embedded and executable specification as oracle instead. Such a specification subsumes the traditional approach, as the library calls or keywords providing the oracle in traditional unit tests easily integrated with a contracted oracle [11]. In addition to that inspection points can also be in the middle of the test case: the contracts are interleaved into the entire program, and not just present at the level of the test case method.

Let $< i, c >$ be a test case. It is executed in the following way:

1. Recreate context c.
2. Check invariant of c (if violated $\rightarrow$ invalid test case).
3. Check precondition of i in c (if violated $\rightarrow$ invalid test case).
4. Run instruction i in c (if normal termination and postcondition is satisfied $\rightarrow$ pass, otherwise $\rightarrow$ fail).

More formally the oracle of a test case can be defined in the following way:

$$\begin{aligned}\mathrm{tc}_{\mathrm{valid}} : I \times C &\rightarrow \mathbb{B}\\ \mathrm{tc}_{\mathrm{valid}}(i,c) &\triangleq \mathrm{wellformed}(i,c) \wedge \mathrm{inv}(i,c) \wedge \mathrm{pre}(i,c)\end{aligned}$$

$$\begin{aligned}\mathrm{tc}_{\mathrm{passing}} : I \times C &\rightarrow \mathbb{B}\\ \mathrm{tc}_{\mathrm{passing}}(i,c) &\triangleq \mathrm{tc}_{\mathrm{valid}}(i,c) \wedge \text{n-terminates}(i,c) \wedge\\ &\quad\ \mathrm{post}(i,c)\end{aligned}$$

$$\begin{aligned}\mathrm{tc}_{\mathrm{failing}} : I \times C &\rightarrow \mathbb{B}\\ \mathrm{tc}_{\mathrm{failing}}(i,c) &\triangleq \mathrm{tc}_{\mathrm{valid}}(i,c) \wedge \neg\, \mathrm{tc}_{\mathrm{passing}}\end{aligned}$$

Here *precondition* is the predicate that is defined as the result of the evaluation of the precondition of an instruction. For method and delegate calls this is equivalent to checking the precondition of the called method. With object creation it is equivalent to checking the precondition of the invoked constructor.

postcondition is the predicate that is defined as the evaluation of the postcondition of an instruction. This is equivalent, similarly to the precondition, to the evaluation of the postcondition of the called method or constructor. Note that the postcondition will not be evaluated if the method does not terminate.

The predicate n-terminates is true if and only if the method terminates normally. In the case of Eiffel (and Spec#) an abnormal termination (such as a null pointer dereference, division by zero, operating system signal, etc.) does not guarantee any postcondition, which is the case described by the formalism above.

In JML there is a pair of pre- and postconditions for normal termination and separate pairs for different kinds of abnormal termination. The above formalism can be easily adapted to this case. Similarly to the way one big pre- and postcondition pair is formed for theorem proving JML annotated programs, a big pre- and postcondition pair can be used for the oracle predicates above.

It might be confusing that above the invariant is applied not only to the context, but to the whole instruction invocation. At first sight, one might be tempted to require the invariant to hold for all objects in the scope. However, there is a need to temporarily break the invariant in order to allow for object state to change. The exact way in which this is implemented depends on the contract-enabled language. In Eiffel the following rule fulfills this purpose: the object which is the target of the currently executing method is allowed to have its invariant temporarily violated. A method can trigger the execution of another method. Consequently, more than one object at any given point in time can have a violated invariant.

It is incorrect to check the invariant of all objects in the scope. Runtime assertion monitoring is typically implemented so that the invariant is checked at the beginning and end of each method execution, also due to performance reasons. This approach is not applicable for our setting either, since an invariant breach due to a method call in the middle of a method call operates not on the initial heap, but on a potentially modified one. Hence it is not clear whether the invariant violation was caused by the instruction invocation or was part of the original context. This distinction is important. For example, a test case might be extracted at first with a context containing a set of objects that satisfy invariants, but then, as a result of changes in the program, the invariant of a class is strengthened and the extracted context may contain objects that do not satisfy the new invariant.

Neither complete invariant checking nor the checking employed by traditional assertion monitoring is appropriate. Instead, the scope of the invariant check must be broadened to include the information of the executing instructions and their called_by information. A first intuition is to check the invariant of all those objects in the scope, except those which are target to any of the methods currently executing (e.g. the targets of the instructions in the transitive closure of the called_by relation to the current invocation). In order to express this, the notion of the *target set* of an instruction invocation is handy. It is the set of all objects serving as target to any of the currently executing methods (where called_by$\star$ is the reflexive transitive closure of called_by):

$$\begin{aligned}\text{target\_set} : I \times C &\rightarrow \mathbb{P}(O)\\ \text{target\_set}(i,c) &\triangleq \{\mathrm{target}(c')|\\ &\quad\ \langle i', c'\rangle\ \text{called\_by}\star\langle i,c\rangle\}\end{aligned}$$

However, requiring all objects not in the target-set to have a valid invariant is overly protective for the purpose of test case execution. The context might perfectly well contain objects with broken invariants that are not needed for the execution of an instruction invocation. In such a case (caused by natural program evolution) one should not be required to throw away the test case. We hence use a notion of *necessary* objects of a program invocation: an object is necessary for an instruction invocation if and only if the execution accesses the object.

$$\mathrm{necessary} : I \times C \times O \rightarrow \mathbb{B}$$

Based on these notions, the final definition of the invariant check is:

$$\begin{aligned}\mathrm{inv} : I \times C &\rightarrow \mathbb{B}\\ \mathrm{inv}(i,c) &\triangleq \forall o \in O|\\ &\quad (\mathrm{necessary}(i,c,o) \wedge \neg(o \in \text{target\_set}(i,c)))\\ &\quad \Longrightarrow \mathrm{inv}_{\mathrm{obj}}(o)\end{aligned}$$

where $\mathrm{inv}_{\mathrm{obj}}$ is the invariant of an object as defined in the class's contracts.

Given these definitions, extracting a test case from a failure becomes very simple:

$$\begin{aligned}\mathrm{testcase}_{\mathrm{failure}} : F &\rightarrow I \times C\\ \mathrm{testcase}_{\mathrm{failure}}(f) &\triangleq \mathrm{recipient}(f)\end{aligned}$$

4.3 From Test Cases to Debugging

Failure test suites and the fault lifecycle. One of the advantages of automatic test case extraction is that a developer observing a failure has the choice of fixing the fault either immediately or later, since the failure is automatically reproducible. Furthermore, Cdd can also provide a benefit that extends into the process of *fixing* the fault. Most non-trivial faults need to be fixed in several places, not just in one, and Cdd-style test case extraction can provide guidance for this process. For example, an observed failure stemming from a null pointer dereference in a method *m* can be fixed by either changing the implementation of *m* in a way so that the null case is treated via a special path, or by strengthening the precondition of *m* to exclude the case in which the reference is null to begin with. Note that, if the developer chooses this latter fix, now the calling site of *m* violates *m*'s precondition. The extracted test case does not prove this since it only starts with the execution of *m*. This is just one of several failure evolution scenarios, all of which can be coped with by extracting not just one test case per failure but a whole test suite, that is one test case per method on the call stack:

$$\begin{aligned} \text{testsuite}_{\text{failure}} : F &\rightarrow \mathbb{P}(I \times C) \\ \text{testsuite}_{\text{failure}}(f) &\triangleq \{tc(i', c') | \\ &\quad \text{recipient}(f) \text{ called\_by}\star\langle i', c'\rangle\} \end{aligned}$$

Extracting unit test suites from system level tests. So far test case extraction was always based on the presence of a failure. Via the recipient the failure determines the method for which to extract a unit test case. The approach works however equally well if the method for which to extract a test case is known through other means. This is achieved by generalizing the test suite notion from above from failures to instruction invocations:

$$\begin{aligned} \text{testsuite}_{\text{invoc}} : I \times C &\rightarrow \mathbb{P}(I \times C) \\ \text{testsuite}_{\text{invoc}}(i, c) &\triangleq \{tc(i', c') | \\ &\quad \langle i, c\rangle \text{ called\_by}\star\langle i', c'\rangle\} \end{aligned}$$

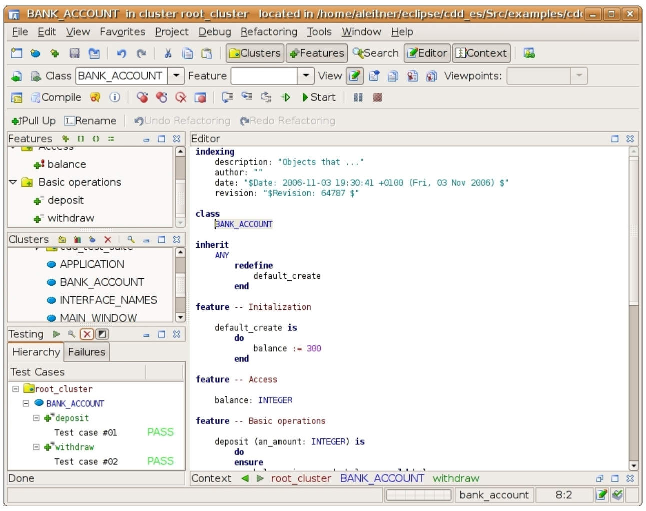
The method selection criteria can stem from a coverage criterion. It can also be used to extract unit tests from system level tests. Developers are often more inclined to write system level tests (i.e. black box tests that exercise the whole program) rather than unit level tests (i.e. tests that exercise one method at a time). The reason is that a few system level tests can achieve a reasonably high coverage and hence require less effort for creation and maintenance. However, modern development practices rely on instant feedback to the developer.

With the presented test case extraction mechanism it is possible to extract short running unit level tests automatically from existing system level tests in just one step. When the IDE signals that the developer is about to change a certain module (in the sense of class or package), relevant system level tests can be executed automatically and for each method invocation of the targeted module a test case is extracted.

While the developer changes the module, he gets instant feedback about whether he broke anything, without the added overhead of unit test suite maintenance.

Deep invariant checks for traditional debugging.

The traditional approach to runtime monitoring of invariants (checking the invariant at method entry and exit) is a compromise between performance and correctness. It captures many invariant violations, but methods accidentally violating the invariant of objects other than the target object can lead to an infected state that is not discovered at the time of the infection.


BANK_ACCOUNT in cluster root_cluster located in /home/aleitner/eclipse/cdd_es/Src/examples/cdd
File Edit View Favorites Project Debug Refactoring Tools Window Help
Clusters Features Search Editor Context
Class BANK_ACCOUNT Feature View Viewpoints:
Compile Start
Pull Up Rename Undo Refactoring Redo Refactoring
Features
balance
Basic operations
deposit
withdraw
Clusters
APPLICATION
BANK_ACCOUNT
INTERFACE_NAMES
MAIN_WINDOW
Testing
Hierarchy Failures
Done
Editor
Context root_cluster BANK_ACCOUNT withdraw
bank_account 8:2

```
Test Cases
root_cluster
  BANK_ACCOUNT
    deposit
      Test case #01      PASS
    withdraw
      Test case #02      PASS
```

```
indexing
    description: "Objects that ..."
    author: ""
    date: "$Date: 2006-11-03 19:30:41 +0100 (Fri, 03 Nov 2006) $"
    revision: "$Revision: 64787 $"

class
    BANK_ACCOUNT

inherit
    ANY
        redefine
            default_create
        end

feature -- Initalization

    default_create is
        do
            balance := 300
        end

feature -- Access

    balance: INTEGER

feature -- Basic operations

    deposit (an_amount: INTEGER) is
        do
        ensure
```

Figure 4: Screenshot of Cdd-integration in the EiffelStudio IDE

The deep invariant check proposed in Section 4.2 can be used in such cases, to selectively check the invariant in situations where one suspects an infected state.

5. IMPLEMENTATION

As described in Section 2, the main motivation of the model developed in Section 4 is to extract test cases in a completely automated way while developers program applications. The approach fundamentally relies on the presence of contracts. We targeted the Eiffel language with our implementation because contracts are first level citizens in this language and practitioners using the language are known to provide contracts in real world settings. This provides for a good setting in which the tool can be validated.

Our implementation is a modified version of EiffelStudio[2], the predominant IDE for Eiffel development. The resulting tool, Cdd, supports Contract Driven Development as described in this paper. The prototype[3] is available for download in both binary and source form under an open source license. A screenshot of EiffelStudio integrating Cdd can be seen in Figure 4.

5.1 Using Cdd

Test case extraction. Cdd tightly integrates with the debugger of EiffelStudio in order to extract test cases during the regular development cycle of an application. When the developer runs an application and an exception is raised, the debugger stops the execution and shows the developer the source code line where the exception was raised, the current call stack, and the content of the variables in scope. In addition to that, the test case extractor of Cdd becomes active and tries to extract a test case that is able to reproduce the current failure. First the test case extractor determines which method to extract a test case for. This is often (but not always) the method that raised the exception. As described in Section 4, Cdd chooses the method receiving the failure as the method under test.

The extractor proceeds to extract a snapshot of the state that is required to invoke the method under test. The target object and all method arguments and their transitive reference closure are serialized. This is an efficient over-approximation of the set of neces-

[2]http://www.eiffel.com

[3]http://eiffelsoftware.origo.ethz.ch/index.php/CddBranch/

sary objects. The result is a test case as can be seen in Listings 2 and 3. The current implementation of Cdd extracts a test case only for the failure-receiving method. To improve flexibility, future releases will extract one test case for each method of the current call stack as described in Section 4.3.

The Cdd implementation poses no runtime overhead during debugging, since the extractor becomes active only at the time of an observed failure. At this point the debugger stops the application anyway, so extraction takes place when the application is not running. The application under test is not instrumented or altered in any way. This has the clear advantage of interfering as little with the developer's working habits as possible.

It is not possible to know *a priori* that a given call will fail, so Cdd extracts the state after the failure. Instrumenting every method to capture its prestate has prohibitive performance overhead. In most cases, extracting the state after the failure is sufficient in order to obtain a test case that exhibits the same error. In some cases though, it is not sufficient and the only possibility is to replay the program with a pre-state capture. Recent advances in capture and replay [17, 16] promise finer grained control and much better performance. We are currently working on integrating such a selective capture replay mechanism into Cdd.

It should be noted that the extracted test cases are implicitly minimized. Instead of re-executing the whole trace that led to the failure, the resulting unit test executes only the method directly responsible for the observed failure. Such test cases are only feasible in the presence of contracts, which serve as oracle. They are only able to cover robustness related failures and functional failures, where the functional behavior has been expressed via contracts.

Test case visualization. The extracted test cases are displayed in a tree where they are grouped by the class and the method that they are testing (see Figures 2). The developer can choose to see all test cases or only failing ones. It is also possible to disable the background extraction and execution of test cases, as well as select an individual test case for debugging (the latter is described in more detail below). Each test case node displays the status of its last execution and the assertion violation raised, if any. The developer can use each test case node to navigate to the source code of the test case or to the receiving method.

Test case execution. Whenever the IDE finds a compilable system, all extracted test cases for that system get compiled and executed. For each test case, Cdd first recreates the context, then checks the invariant, and finally invokes the method under test with the created context. The current version of Cdd only checks the invariant of the object under test (the thorough invariant check described in Section 4.2 is work in progress). If the invariant was found to be violated the test case is flagged as invalid and this test case is not executed. Otherwise the method under test is executed with the recreated target object and arguments.

During execution, assertion monitoring is enabled and violated assertions are reported in the form of exceptions back to the test case executor. If the precondition of the method under test has been violated, Cdd flags the test case again as invalid and does not execute it further. There is an important point to note here: only the violation of the outermost precondition flags an invalid test case; a precondition evaluated as part of a method call triggered directly or indirectly from the method under test flags a failing test case instead. This is also true if the method under test is recursive. All exceptions (e.g. invariants, postconditions, preconditions, check instructions, segmentation faults, division by zero, etc.) other than the outermost precondition and invariant check signal a failing test case.

Background execution of test cases allows the developer to always see the latest state of the test cases. However, in addition to this, the developer may want to more clearly understand why a particular test case fails. For this case Cdd allows him to execute a test case in the regular debugger. When doing this on a failing test case, the debugger will automatically stop at the exception being raised and thus allow the developer to inspect the concrete values. Additionally the developer can set breakpoints and thus step through the test case (including the method under test) line by line and inspect the state at method entry point and how this state evolves.

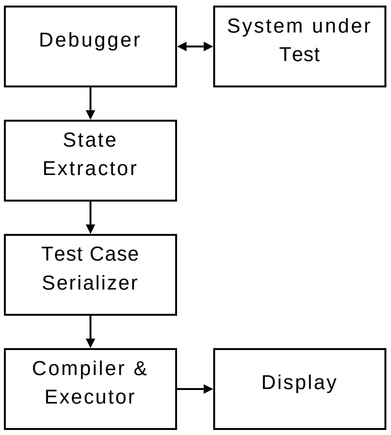


Figure 5: Architecture of Cdd implementation

5.2 Architecture

Cdd is implemented as a modification to the EiffelStudio IDE. The implementation consists of approximately 50 new classes totaling to around 6000 lines of code. The addition is relatively small compared to EiffelStudio itself (1400 classes and 2 million lines of code) and, to keep maintenance efforts reasonable, we kept the number of existing classes that we modified to a minimum. The classes of the extension can be roughly divided into groups achieving the following:

- Model (internal representation of test cases)
- State and code extraction using the interface of the debugger
- Test case serialization
- Compilation and execution of the serialized test cases
- Test harness (simple unit testing framework)
- Visualization and user interface
- Example code

Figure 5 shows the basic control flow. The debugger is in control over the system under test (the application that the developer is working on). When a failure occurs, the state extractor retrieves information from the debugger about the current state of the application. The test case serializer saves this state into a compilable unit test case. The resulting test case is then compiled and executed and finally the results are displayed.

6. FUTURE WORK

Our current implementation does not cope with concurrent applications. The SCOOP [2] mechanism extends the semantics of contracts to the concurrent case. A contract equipped language supporting SCOOP will allow us to handle concurrent applications with minimal modifications on our implementation.

The recently introduced selective capture and replay [17, 16] mechanism, promises much increased performance of both the capture and replay phase. It is based on the idea that a program is divided into two parts, an observed part and an external part. Instead of capturing state or state changes, all in and outgoing events are recorded instead. In and outgoing data only needs to be recorded in a shallow fashion. For replaying a run, the replay-harness is able to replace the external part completely, freeing the application from all dependencies that this part introduces. We are currently working on a selective capture and replay implementation for Eiffel, which will provide significant benefits in the following areas:

Prestate extraction. To correctly extract a test case from a failure, the state (i.e the relevant objects on the heap) have to be captured at the right before the recipient of the failure is called. We currently capture the state at the time of the failure, which is too late. At the time right before the recipient is called it is not clear yet whether the call will result in a failure or not and once the failure has been detected the original pre-state might be already changed. Capturing the real pre-state for all method calls is prohibitive due to its performance and memory overhead. A solution would be a posteriori user-guided extraction, but this would require manual intervention. Given the right border, selective capture and replay makes it possible to capture all executions by default, while inflicting minimal performance and memory overhead. Replays can then be run completely automatically in the background, which would remove the need for the user's intervention, but still capture the correct prestate.

Non-determinism. While most programming languages do not provide a source for non-determinism directly, programmers can typically use the foreign function interface to acquire external inputs (user input, network, database, etc.), and this can be considered a cause of non-determinism within the program. With selective capture and replay it is possible to put all sources of non-determinism into the external part, which makes replays completely deterministic. Even if a failure-producing trace was dependent on certain GUI inputs, network connections, or data base state, the replay is completely freed from these dependencies.

External state. The foreign function interface also introduces references to outside data (e.g. window handles, file handles, pointer to partially untyped C values) and this data that cannot be reflected. Such state is never directly manipulated within the program; external code is invoked instead. The implicit state minimization of selective capture and replay removes this dependency on the external state (again given that all external code is part of the unobserved code).

7. CONCLUSIONS

This article explains the fundamentals of the Contract Driven Development approach. A tool autonomously observes the developer while he is working on a program and extracts test cases from failures either provoked by the developer (in the spirit of test driven development) or by mistake (leading to a regression test). The approach is novel in that complete test cases are extracted not only from the information provided by the system under test, but also from non-permanent clues given by the programmer during development.

The approach is introduced by a case study and explained via a language agnostic model, applicable to arbitrary contracted code. The extracted test cases are both fast executing, small, and stripped of many dependencies. To aid the debugging process, failure test suites can be extracted and give the developer guidance during multi-step correction cycles. As a corollary to test case extraction based on failures, we show how the approach can be used to automatically extract fast executing unit test from slow executing system level tests.

The Cdd tool implements the idea of contract driven development. Cdd is integrated into EiffelStudio, a major Eiffel IDE. Developers can use the tool without changing their development process, since the approach is completely non-intrusive. Cdd offers the advantages of automatic test case extraction from actions that developers perform anyway while writing source code, and thus builds up a comprehensive unit test suite and offers support for functional testing, debugging, and regression testing.

7.1 Acknowledgements

We thank Jocelyn Fiat and Manu Stapf for their support and insights on the compiler and IDE. We are especially grateful to Andreas Zeller and Martin Burger with whom we had many discussions that helped shape this work. We thank Bernd Schoeller for many fruitful discussions.

8. REFERENCES

[1] Jtest. Parasoft Corporation. http://www.parasoft.com/.

[2] ARSLAN, V., EUGSTER, P., NIENALTOWSKI, P., AND VAUCOULEUR, S. SCOOP - concurrency made easy. *Dependable Systems: Software, Computing, Networks* (2006).

[3] BALCER, M., HASLING, W., AND OSTRAND, T. Automatic generation of test scripts from formal test specifications. In *TAV3: Proceedings of the ACM SIGSOFT '89 third symposium on Software testing, analysis, and verification* (New York, NY, USA, 1989), ACM Press, pp. 210–218.

[4] BOSHERNITSAN, M., DOONG, R., AND SAVOIA, A. From Daikon to Agitator: lessons and challenges in building a commercial tool for developer testing. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis* (New York, NY, USA, 2006), ACM Press, pp. 169–180.

[5] BOYAPATI, C., KHURSHID, S., AND MARINOV, D. Korat: automated testing based on java predicates. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2002), Rome, Italy* (2002).

[6] CSALLNER, C., AND SMARAGDAKIS, Y. Dsd-crasher: A hybrid analysis tool for bug finding. In *International Symposium on Software Testing and Analysis (ISSTA)* (July 2006), pp. 245–254.

[7] FELDMAN, Y. A. Extreme design by contract. In *Fourth Int'l Conf. Extreme Programming and Agile Processes in Software Engineering (XP 2003)* (2003), Springer Verlag, pp. 261–270.

[8] GARGANTINI, A. Using model checking to generate fault detecting tests. In *Tests and Proofs (TAP 2007). Proceedings* (2007), Y. Gurevich and B. Meyer, Eds., Lecture Notes in Computer Science, Springer.

[9] GODEFROID, P., KLARLUND, N., AND SEN, K. Dart: directed automated random testing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2005), ACM Press, pp. 213–223.

[10] HOVEMEYER, D., AND PUGH, W. Finding bugs is easy. *SIGPLAN Not. 39*, 12 (2004), 92–106.

[11] Leitner, A., Ciupa, I., Meyer, B., and Howard, M. Reconciling manual and automated testing: the AutoTest experience. In *Proceedings of the 40th Hawaii International Conference on System Sciences - 2007, Software Technology* (January 3-6 2007).

[12] Marinov, D., and Khurshid, S. TestEra: A novel framework for automated testing of Java programs. In *Proc. 16th IEEE International Conference on Automated Software Engineering (ASE)* (2001), pp. 22–34.

[13] Meyer, B., Ciupa, I., Leitner, A., and Liu, L. L. Automatic testing of object-oriented software. In *Proceedings of SOFSEM 2007 (Current Trends in Theory and Practice of Computer Science)* (2007), J. van Leeuwen, Ed., Lecture Notes in Computer Science, Springer-Verlag.

[14] Offutt, A. J., Xiong, Y., and Liu, S. Criteria for generating specification-based tests. In *Proceedings of the Fifth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '99)* (October 1999), pp. 119–131.

[15] Oriat, C. Jartege: a tool for random generation of unit tests for Java classes. Tech. Rep. RR-1069-I, Centre National de la Recherche Scientifique, Institut National Polytechnique de Grenoble, UniversitÂt'e Joseph Fourier Grenoble I, June 2004.

[16] Orso, A., Joshi, S., Burger, M., and Zeller, A. Isolating relevant component interactions with JINSI.

[17] Orso, A., and Kennedy, B. Selective Capture and Replay of Program Executions. In *Proceedings of the Third International ICSE Workshop on Dynamic Analysis (WODA 2005)* (St. Louis, MO, USA, may 2005), pp. 29–35.

[18] Ostrand, T. J., and Balcer, M. J. The category-partition method for specifying and generating fuctional tests. *Commun. ACM 31*, 6 (1988), 676–686.

[19] Ostroff, J. S., Makalsky, D., and Paige, R. F. Agile specification-driven development. In *XP* (2004), J. Eckstein and H. Baumeister, Eds., vol. 3092 of *Lecture Notes in Computer Science*, Springer, pp. 104–112.

[20] Pacheco, C., and Ernst, M. D. Eclat: Automatic generation and classification of test inputs. In *ECOOP 2005 — Object-Oriented Programming, 19th European Conference* (Glasgow, Scotland, July 25–29, 2005).

[21] Saff, D., and Ernst, M. D. An experimental evaluation of continuous testing during development. In *ISSTA 2004, Proceedings of the 2004 International Symposium on Software Testing and Analysis* (Boston, MA, USA, July 12–14, 2004), pp. 76–85.

[22] Tillmann, N., and Schulte, W. Parameterized unit tests with unit meister, 2005.

[23] Visser, W., Pasareanu, C. S., and Khurshid, S. Test input generation with java pathfinder. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis* (New York, NY, USA, 2004), ACM Press, pp. 97–107.

[24] Xie, T., Marinov, D., Schulte, W., and Notkin, D. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 05)* (April 2005), pp. 365–381.