Testing the Implementation of Business Rules using Intensional Database Tests

David Willmor and Suzanne M. Embury
School of Computer Science, University of Manchester,
Oxford Road, Manchester, United Kingdom
d.willmor|s.m.embury@cs.manchester.ac.uk

Abstract

One of the key roles of any information system is to enforce the business rules and policies set by the owning organisation. As for any important functionality, it is necessary to verify the implementation of any business rule carefully, through thorough testing. However, business rules have some specific features which make testing a particular challenge. They represent a more fine-grained unit of functionality than is usually considered by testing tools (programs, module, UML models, etc.) and their implementations are typically spread across a system (or perhaps some specific layer of a system). There is no convenient one-to-one relationship between programs and business rules that can facilitate their testing.

To the best of our knowledge, no tools, methods or guidelines exist for helping software developers to test the implementation of business rules. Standard testing tools can help to a certain extent, but they leave the rule-specific work entirely in the programmer's hands. In this paper, we discuss the problems of testing business rules, and elicit the key features of a good test suite for a collection of business rules. We focus in particular on constraint business rules - an important class of rule that is commonly applied to the persistent data managed by the information system. We show how intensional database tests provide a suitable platform on which to implement business rule tests rapidly, and show how existing intensional test suites can be automatically adapted to test business rules. We have applied these ideas in a case study, which has allowed us to compare the relative costs of creating and executing these augmented test suites, as well as providing some evidence of their ability to detect faults in business rule implementations.

1. Introduction

Many modern information systems implement, amongst other things, some form of business rule [13]. These rules control and constrain the behaviour of the system in terms of the business's policies and principles. Example rules include: no customer may have more than two unpaid orders; and a customer is a gold customer if they have placed more than 50 orders in the last year. These rules are central to the operation of the business and are deeply embedded within the information system; they must therefore be verified thoroughly. Additionally, business rules evolve frequently, in line with changing business focus, market opportunities and statutory regulations. Whilst the question of how to implement [2, 7, 8] and evolve [3, 10] business rules has been addressed in the literature, the verification of business rule implementations has not been studied and presents a number of unique challenges, relative to standard database testing.

What, then, are the barriers to effectively testing the implementation of a set of business rules? Time is the most significant barrier: the time required to run sufficient tests to verify each of the rules, but also the time needed for testers to design test cases that not only exercise the functionality of the system but at the same time verify that a set of business rules have not been violated. These problems are particularly acute in information systems. Such systems will need to enforce many different business rules — the implementations of which will be spread across a system. The type of implementation will also vary between rules; some will be implemented in the program's source code whilst others may be implemented in a rule repository or as a trigger within the database itself.

In this paper, we present an approach for verifying the implementation of a set of *constraint* business rules via the execution of test cases. Constraint business rule are a common form of rule that manifest themselves as constraints on the database. Our approach automatically generates *check-conditions* that can determine if a business rule has been violated. In order to relieve the tester from having to manually create test cases to verify the business rules, we utilise existing test suites and augment these with the necessary check conditions. Such test suites should (if designed correctly) cover the various different behaviours of the information system. Our approach focuses on testing business

rules that are implemented in normal program code, rather than rules that are enforced externally by a rule engine.

Our approach utilises *intensional database test cases* [18], that include pre- and post-conditions in the form of declarative queries specifying the requirements of the database both before and after executing the test. It is these pre- and post-conditions that our approach augments with the business rule check-conditions. Intensional tests improve over previous database testing approaches [4, 5, 9] (which can be characterised as *extensional* test case specifications) in that it is not necessary to specify the exact set of tuples that must be present in the database when the test is executed. Instead, the intensional approach allows testing personnel to construct high-level, declarative descriptions of test cases, which are typically shorter and quicker to specify than their extensional equivalents.

Thanks to the declarative nature of these preconditions, it is possible for the testing harness to automatically update the database state before executing the test case, to ensure that the precondition is satisfied, regardless of its initial state [18]. Because of this, intensional test cases are more flexible and less brittle than their extensional equivalents, in that they are able to run against a variety of databases without requiring manual intervention. This allows us to verify the business rules against a number of different states without requiring any modifications to either the test cases or check-conditions.

We begin, in Section 2, by discussing the problem of testing business rules and elicit the key features of a good test suite for a collection of business rules. Section 3 presents our approach for generating a database query that can determine if a business rule has been violated. In Section 4, we discuss how we can augment existing intensional database test cases with conditions required to verify if a business rule has been violated. In Section 5, we present our implementation of the business rule testing techniques — *DOT-BR*. In Section 6, we present the results of an experimental evaluation of our approach. Finally, in Section 8, we conclude and discuss avenues for future work.

2. Verification of Business Rule Implementations

In its most general form, a business rule (BR) is a statement that "defines or constrains some aspect of a business" [6]. This broad definition encompasses several types of rule[1]:

- Structural rules, such as the statement that each customer has a 6 digit identifying code, or that every order must be tagged with the identifier of the member of sales staff responsible for its processing.
- Derivation rules, such as the set of steps to be used to calculate the government sales tax that will be imposed on an order.
- Constraint rules, such as the requirement that all overdrafts of more than £200 must be authorised by a senior manager, or the policy that all Home Office customers be charged a fixed fee if they are overdrawn for more than 10 days in any financial year.
- Event rules, such as the requirement that when a customer has made three failed attempts to withdraw from an account, that account should be suspended.

As can be seen from these examples, business rules extend and complement the business processes that an organisation supports. They tend to be relatively small and self-contained units of functionality that are applicable across the whole organisation or within sub-divisions and departments. Most real-scale organisations will have many thousands of such rules, all of which must be maintained uniformly across multiple information systems and organisational divisions. Moreover, they are typically very volatile. Changes in business rules are regularly forced on organisations through the imposition of new statutory regulations, government policies or economic/market conditions. Many businesses will also modify their business rules by choice, as part of their attempts to maximise revenue, capture new market share and minimise customer churn.

The correct and uniform enforcement of business rules by information systems is therefore vital for many organisations (especially where the rules arise from statutory regulations and legal penalties may result from their violation). Where possible, rules will be centralised in rule repositories, and enforced by means of rule engines (e.g. ILOG JRules[2], InRule[3] or QuickRules[4]). However, for many applications, the performance costs of such an approach make it impractical, and business rules must be implemented by embedding them within the source code of the rest of the system. Verification of such implementations by thorough testing is extremely important, in part because of the difficulties of ensuring correct implementation of the rules and in part because of the serious consequences that can arise when rules are partially or inconsistently enforced by a computer system.

To order to understand the difficulties involved in testing business rules, it is necessary to consider what happens to such a rule when it is implemented. Business rules begin life as high-level conditions that the organisational state

1 There are many classifications for business rules. This one is based on that proposed by Shao and Pound [14].

2 http://www.ilog.com/products/jrules/

3 http://www.inrule.com

4 http://www.yasutech.com/

must adhere to. These conditions can typically be violated by several different actions. For example, a business rule that requires that "all overdrafts must be authorised" can potentially be broken by two different operations: lowering the balance of a customer's account, and cancelling an overdraft authorisation for some customer. The supporting software systems must ensure that whenever either of these actions occurs the business rule is not violated — that is, ensuring that if a debit action will take a customer's account into an unauthorised overdraft then either authorisation is obtained or the debit is not allowed to proceed. Similarly, if a request is made to cancel the overdraft authorisation for some customer, the operation can only proceed if that customer does not currently have a negative balance.

In order to implement this kind of behaviour, the high-level business rule must be fragmented into a set of simpler event-condition rules, each of which describes the semantics of the original rule in the context of one of the actions that can violate it. Each of these rule fragments must then be implemented in the system code, wherever its triggering action might occur. Thus, code implementing a single business rule will typically appear in many different programs, and may even occur several times within a single program, in several different guises[5].

All this presents a number of challenges for the tester:

- The business rule code is spread across many different code units (including, in some cases, in stored procedures and triggers within the DBMS), all of which must be tested for correct enforcement. This leads to a rapid explosion in the number of test cases needed to achieve good test coverage for business rules.
- Omission of a rule fragment is just as serious a problem for rule implementation as incorrect insertion of a fragment. Therefore, it is not enough to test only the code units that have been directly modified during the introduction of the new rule. This adds further to the test case explosion problem.
- Because many business rules are concerned with the organisation's state, i.e. the persistent data managed by the software system, it is often difficult to formulate simple unit tests for them. Instead, integration testing must be performed, in which several programs are executed in sequence to create the circumstances that might lead to a rule violation. The number of program sequences implicated in rule violation is potentially very high, adding further to the number of test cases required for full coverage.
- Since business rules cannot be tested in isolation (because of their tight integration with other system functionality, including business processes and other business rules), the task of formulating precise and efficient test cases for business rules is non-trivial and requires an excellent understanding of the semantics of the full set of business rules and business processes on the part of the software tester.

Thus, while it is certainly possible to code business rule tests using conventional testing tools such as JUnit, a considerable intellectual effort is required from testing staff to ensure test correctness and broad representative coverage. In the remainder of this paper, we consider how we can provide better support to testing personnel when verifying the implementation of business rules, focussing on the specific case of constraint business rules expressed over the persistent state of the organisation.

3. Checking a Business Rule

Constraint business rules impose conditions on the state of the information system's database. To verify whether a business rule has been correctly implemented we must be able to convert the rule (in this case, expressed as first-order logic) into some form of SQL query which can be evaluated against the database. The problem of converting first-order logic statements into SQL queries has been widely studied in the database literature [1, 11, 15] and is well understood. Essentially, all that is necessary is to negate the rule expression and convert the result into an SQL query. We will illustrate this simple process through a couple of examples.

The bodies of constraint business rules take the form $a \Rightarrow b$ where both a and b are conditions. Whenever condition a is satisfied, if the business rule is to be enforced, then b must also be satisfied. An example of such a rule is:

$$(\forall c, a)\ customer(c) \wedge age(c, a) \Rightarrow a \geq 21$$

which specifies that all customers must be 21 years or older. This business rule can be potentially violated by the following update:

$$+age(x, y)$$

if $y < 21$. Therefore, if a program correctly implements a business rule it will include guards that ensure that $y \geq 21$ and preventing the violating update occurring. The implementation can be verified by executing test cases that verify that the database state does not satisfy the following condition:

5 It is good design practice to attempt to contain business rule implementations within a single tier of an application as far as possible. However, it is rare that a complete encapsulation of the rules can be achieved, due to the performance and usability costs of this approach. In practice, many business rules are implemented in the presentation and database layers of classic three-tier systems, as well as the business logic layer that is their conceptual home.

$$(\exists c, a)\ customer(c) \wedge age(c, a) \wedge a < 21$$

This can be converted into a standard SQL query that checks for these violating tuples:

```
SELECT custID, age FROM customer
WHERE age < 21;
```

If this query is executed against a database state that does not violate the business rule it will obviously not return any tuples.

Another example of a constraint business rule is:

$$(\forall c)\ customer(c) \wedge managed(c) \Rightarrow gold(c)$$

which specifies that all managed customers are considered automatically to have "gold" status. According to the method proposed by Embury and Shao [3], there is only one update that will trivially violate the business rule, the composite update:

$$+cust(x), +managed(x), -gold(x)$$

since each of the updates refer to the same customer (as denoted by x) we know that a violation will always occur if each of the updates is executed. There are many other, simpler updates that will *potentially* violate the rule, however. For example,

$$+managed(x)$$

This update changes the status of existing customer x to be a managed customer. We cannot tell whether the rule will be violated in this case without fetching more information about customer x. If x already has gold status, then all is well. If not, the rule is violated by the update.

Programs including this update can be verified by test suites that determine that the database state does not satisfy the condition:

$$(\exists c)\ customer(c) \wedge managed(c) \wedge \neg gold(c)$$

either before or after the program is executed. The condition is easily converted into a standard SQL query that searches for violations:

```
SELECT custID, managed, gold
FROM customer
WHERE managed = true
  AND gold = false;
```

4. Utilising Existing Test Suites

Our approach to verifying business rules is based on augmenting intensional database tests with the necessary check-conditions. The use of intensional test cases in this way allows us to generate test cases for business rules without requiring significant input from testing personnel. Although the resulting test suites are unlikely to be complete, they do provide a cost-effective starting point for any business rule verification effort. The cost savings come in two forms: the cost in testing personnel effort required to construct them, and the cost of execution, since the business rule tests can be piggy-backed onto the existing business processing tests with much less overhead than if separate test cases had been constructed for them.

In this section, we firstly present the concept of an intensional database test case (Section 4.1). We then show how these test cases can be augmented with the necessary check-conditions (Section 4.2).

4.1. Intensional Database Test Cases

In this section, we introduce the notion of an *intensional database test case*, which provides the foundation for our business rule testing approach. In the remainder of the paper, the term *test case* should be understood to mean an intensional database test case, unless otherwise specified.

An intensional test case [18] is a quintuple $\langle p, i, o, DB_i, DB_o \rangle$, where:

- p is the program to be executed in the test,
- i is a tuple of n variables and literals to be used as input to p, where n is the number of input parameters expected by p,
- o is a tuple of m variables and literals that describes the expected outputs of the program p if the test case has executed correctly,
- DB_i is a set of constrained queries that together define the conditions required of the database state prior to test case execution, and
- DB_o is a set of constrained queries that together specify the conditions that must hold in the database state after execution of the query if no fault has been encountered.

By the term *constrained query*, here, we mean a normal relational query with additional constraints on size of the result set. The semantics of intensional test cases are defined elsewhere [18], but the key points can be gathered from the two example test cases shown in Figure 2. Our data-oriented testing framework (called Dot-Unit) is implemented as

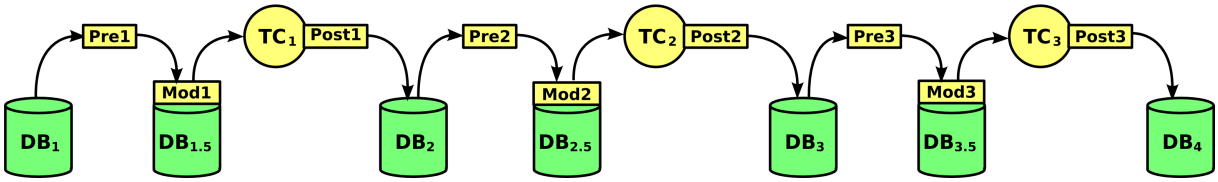

Figure 1. Intensional Testing with DOT-Unit

an extension of the well-known JUnit testing framework for Java[6]. Each test case is implemented as an individual method, which imposes the pre-condition constraints, executes the test case and checks the results for evidence of faults — this process is shown in Figure 1. For example, the first test case, `testReclassifyCustomers`, describes a test of the applications which implement a loyalty card scheme for a retail organisation. Periodically, customers are reclassified, based on their purchasing patterns, and receive certain benefits based on their classification. In this case, we wish to test that any class `A` customers whose balance is less than £200 are reclassified by the operation to class `B` customers. The pre-condition query states that there must exist at least one class `A` customer with a low balance. If the database against which the test case is executed already contains such a customer, then his or her identifier is returned as a binding for the variable `:cn`, which will be used later in the test specification. If no such customer exists, however, a special database preparation algorithm is invoked (within the `precondition` method call) in order to create one. The identifier of the newly created customer is then bound to the `:cn` variable. In this way, regardless of the state of the database at the start of the test case, it will be in a suitable state when the test method is actually executed.

After execution of the test method, the next step is to check whether the outputs match what are expected of a fault-free behaviour. In this case, we expect a (single) customer with identifier `:cn` to still exist in the database, and to have a new class of type `B`. Note the dual role of the pre-condition element of the test case, in both preparing the database for test execution and in extracting data values to be used in the remainder of the test case. Without this latter facility, it would be necessary to embed literal values into test cases (i.e., to assume that a customer with identifier "1234" exists in the database and has the correct type). By allowing the pre-condition evaluator to extract such values from the database, the test case becomes much less brittle and is able to run against any database that matches its schema constraints.

The second test case, `testAddCustomer`, checks for faults in the method which adds new customers to the database. Since we want to test the behaviour when adding a new customer, we cannot use the previous strategy of retrieving the relevant bindings from the database. For this test case, we need a customer identifier that is not already present in the database table. The test case illustrates how this can be achieved, by making use of data generation functions defined separately by the testing personnel. Note that, apart from this detail, the overall pattern of this second test case is the same as the first: impose pre-condition and prepare the database for the test case, execute the test method and finally check that the result is as expected.

Our initial experiences in using the DOT-Unit framework indicate that, for many test cases, intensional test cases are much shorter and quicker to specify than the more traditional extensional test cases. However, the declarative nature of the pre- and post-conditions also opens up further possibilities for analysis of test cases, to provide better support to testing personnel in terms of analysing the coverage of test suites [16] or, as we shall now demonstrate, they can be modified in order to verify the implementation of business rules.

4.2. Augmenting the Test Suite

A test suite, if designed correctly, should thoroughly test the behaviour of the information system for which it is designed. What exactly this behaviour is and how thoroughly it is tested, will vary between different testing personnel and types of system — but is typically based on one or more test adequacy criteria [19]. However, it is precisely this behaviour that we want to ensure does not violate any of the organisations business rules. Thus, our approach will augment just such a test suite with the check-conditions we presented in Section 3. These check-conditions play a similar role to that of assertions [12] within each individual test case[7]. Let us consider the following business rule:

6 `http://www.junit.org`

7 Conventionally, assertions are checks or annotations places within source code in order to ensure that certain invariant conditions are maintained by the code. Another approach, adopted by JUnit and ourselves amongst others, is to place the assertions not inside the code to be tested but immediately before or after the call to the code unit in question.

```
public class CustomerClassificationTest extends DatabaseTestCase {
  public void testReclassifyCustomers() {
     preCondition("ANY :cn GENERATED BY SELECT custNo FROM customer
                   WHERE customerClass = 'A' AND balanceMinimum < -200");
     SalesDB.reclassifyCustomers();
     postCondition("EXACTLY 1 :class GENERATED BY SELECT customerClass FROM customer
                    WHERE custNo = :cn");
     assertEquals(binding(":class"), "B");
  }

  public void testAddCustomer() {
     preCondition("ANY :cn, :name, :addr GENERATED BY SELECT gc.custNo, gc.name, gc.addr
                   FROM genCustomerDetails() AS gc WHERE gc.custNo NOT IN
                   (SELECT custNo FROM customer)");
     SalesDB.addCustomer(binding(":cn"), binding(":name"), binding(":addr"));
     postCondition("EXACTLY 1 :cn, :name, :addr GENERATED BY SELECT custNo, name, addr
                   FROM customer;");
  }
}
```

Figure 2. An Example DOT-Unit Test Case

```
public class OrderTest extends DatabaseTestCase {
  public void testCompleteOrder() {
    checkCondition("NO * GENERATED BY SELECT * FROM orders WHERE complete = true AND
                    balanceOutstanding != 0;");
    preCondition("ANY :orderID GENERATED BY SELECT orderID FROM orders WHERE
                  complete = 'false'");
    OrderSystem.completeOrder(binding(":orderID"));
    postCondition("EXACTLY 1 :oid GENERATED BY SELECT orderID FROM orders WHERE orderID
                   = :orderID AND complete='true'");
    checkCondition("NO * GENERATED BY SELECT * FROM orders WHERE complete = true AND
                    balanceOutstanding != 0;");
  }
}
```

Figure 3. An Example Augmented Test Case

$$(\forall o, b)\ orders(o) \wedge complete(o) \wedge balanceOutstanding(o, b) \Rightarrow b = 0$$

which states that for an order to be complete it must not have an outstanding balance (i.e., it must be paid). This business rule can be converted to the following check-condition:

```
SELECT * FROM orders
WHERE complete = true
  AND balanceOutstanding != 0;
```

This will obviously select all of the tuples of the `orders` relation that violate the business rule. In order to augment an intensional database test case with the check-condition we must convert it into a condition expressed in our extended-SQL language. Obviously, with the above example if the rule is enforced we wish for no such tuples to exist. Therefore, we can utilise the following condition:

```
NO * GENERATED BY
SELECT * FROM orders
WHERE complete = true
  AND balanceOutstanding != 0;
```

Figure 3 shows an example of an augmented test case that includes check-conditions for the business rule. Two check-conditions exist in this example — the first checks the database before executing the test; whilst the second checks the database after executing the test. The first check-condition is important, as we cannot determine if the execution of a test violates a business rule if the database already violates that rule. Therefore, we can use the existing approach for preparing a database to satisfy a pre-condition

in order to automatically remove any tuples that violate the business rule at run-time.

Managing the interactions between the existing preconditions and the augmented rule preconditions is complicated, and beyond the scope of a simple augmentation tool. For example, if we place the rule precondition first, it is possible that the existing preconditions for the test case may cause some violating data to be created before the unit under test is executed. Conversely, if we place the rule precondition after the existing preconditions, then we run the risk that data specifically needed for the test case might be deleted, because it does not satisfy the business rule. We have opted in the current version of the augmentation tool to place the rule preconditions first and to give priority to the existing test case preconditions. This must be borne in mind when interpreting test results, as anomalous behaviour might be due to unwanted interactions between the preconditions, rather than a fault in the program itself. Clearly, this is a limitation that must be overcome in future versions of the augmentation tool.

5. DOT-BR Testing Tool

Our approach to testing business rules has been implemented within a testing tool, called *DOT-BR*. This tool is part of a larger Data-Oriented Testing[8] framework that is under development at the University of Manchester [16–18]. DOT-BR takes as input a set of business rules (currently expressed in first-order logic) and an intensional database test suite. From this, it generates an augmented test suite that aims to verify the implementation of the business rules. DOT-BR will generate at least one augmented test case for every one that appears in the input suite. However, in certain situations (i.e., when the database has to be prepared in different ways in order to verify the business rule) it will produce multiple augmented versions of a test case.

The augmented test suite operates as a standard DOT-Unit [18] test suite (itself based on the JUnit[9] testing framework) that can therefore be executed within existing development and testing tool sets.

6. Experimental Evaluation

In this section, we will describe how we have used a small case study to gain a preliminary understanding of the effectiveness of our business rule testing approach and the DOT-BR testing tool. The case study we have chosen is an implementation of a shop's information system. This system is divided into three individual application programs representing: the sales, warehouse and management departments. Each application program implements business logic relating to those departments and interacts with different yet overlapping portions of the database. In its entirety, the system consists of 2681 lines of executable code spread amongst 16 classes. The test suite to be used consists of 50 intensional test cases. This system implements 10 constraint business rules — ranging from simple constraints on the value of an attribute to those spanning across a number of different relations.

The DBMS used was MySQL[10] 5.0.18 and the database schema consisted of five relations and four foreign keys. All experiments were run on a Pentium-M 2.0GHz machine with 1Gb RAM, running Ubuntu Linux.

The practicality of this approach for testing business rules largely depends on the performance overhead imposed by two factors: the augmentation of the existing test suite; and the extra time required to query (and possibly prepare the database) for the extra conditions. We must also consider the ability of our approach for detecting faults in the implementation of the business rules.

Cost of Augmenting Test Suites

Our first experiment evaluates the cost (in terms of time) of generating our augmented test suites. The process of augmenting the test suite consists of: (1) converting the set of business rules to check-conditions, and (2) inserting the check-conditions into each individual test case as pre- and post-conditions. Figure 5 shows the results of this experiment. The graph shows the cost of augmenting the test suite for increasing numbers of business rules (1, 3, 5, 10) applied to test suites of various size (5, 10, 25, 50). We can see from the graph that as both the number of tests and business rules increases, the cost of generating the augmented test suite increases. However, even as these increases occur the cost remains low — especially relative to the cost of executing the entire augmented test suite as shown in the next experiment.

Execution Time

Our second experiment evaluates the execution time of each of the previously augmented test suites in comparison to the original intensional database test suite. Figure 6 shows the results of this experiment. The graph shows the cost of executing the test suite for increasing numbers of business rules (0[11], 1, 3, 5, 10) against various different sizes of database[12] (10, 50, 100, 250, 500). We can see from the graph that increases to both the size of the relations and

8 http://www.cs.man.ac.uk/~willmord/dot/

9 http://www.junit.org

10 http://www.mysql.com

11 We include the original intensional test suite for comparison.

12 Measured as the average number of tuples in each relation.

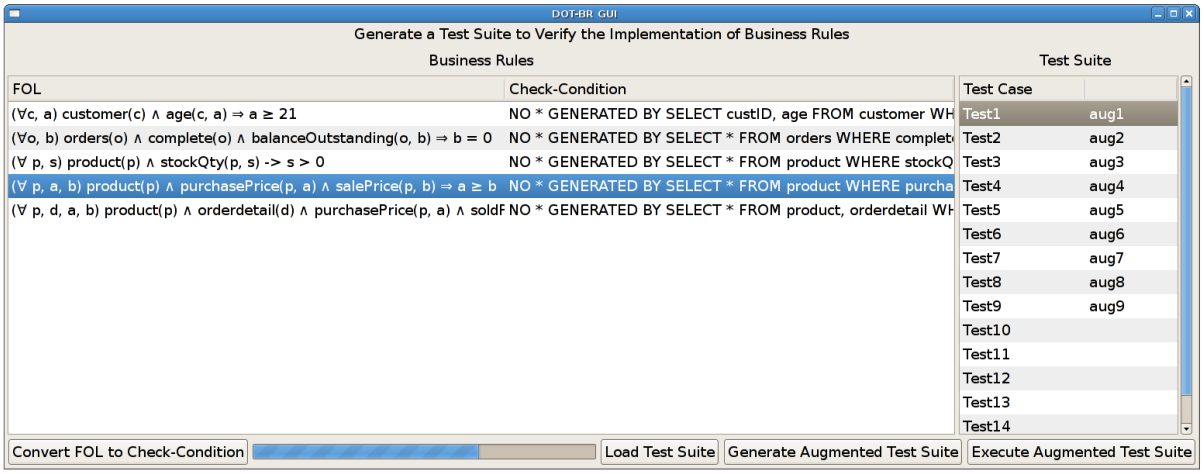

Figure 4. DOT-BR Testing Tool

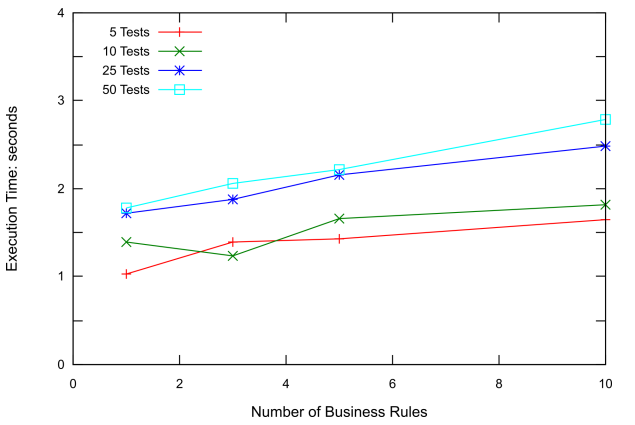

Figure 5. Cost of Augmenting the Test Suite

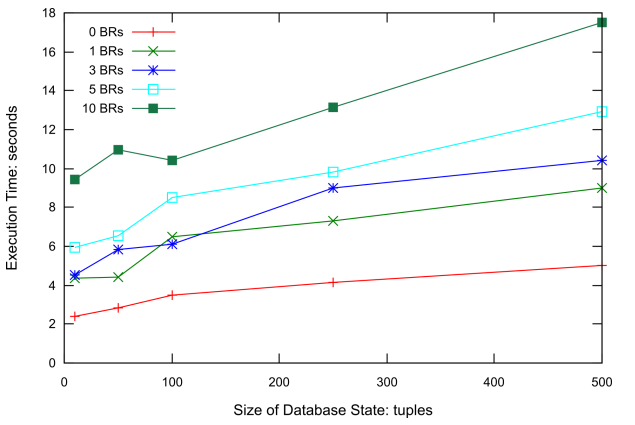

Figure 6. Execution Time of Test Suite

the number of business rules results in significant increases in the cost of executing the test suite. This increased cost can be solely attributed to the overheard involved with checking the business rules.

Fault Detection

Our third experiment evaluates the ability of our approach to detect the faulty implementation of a set of constraint business rules. The program was seeded with 20 faults — these faults were created by removing the guards around database updates that enforce the business rules. For example, consider the previously used business rule that all customers must be at least 18 years old. Consider a method that updates a customers age — typically it would contain an update guarded by a check on the age to be inserted into the database — as shown below[13]:

```
public updAge(String custID, int age) {
  if(age >= 18) {
    stmt.executeUpdate("UPDATE customer"
      +" SET age = "+age+" WHERE custID"
      +" = "+custID+";");
    }
}
```

By removing the statement `if(age >` 18)= we remove the guard on the database update. In this case, the guard was a simple integer comparison; however, other examples include database queries and more complicated nested comparisons.

13 The Java code has been simplified to only include the necessary information.

The experiment involved executing both the original and the augmented test suite against the information system. We then matched test case failures to the seeded faults — identifying the percentage of faults that were exposed by each test suite. Our augmented test suite detected 80% of the seeded faults — with 4 faults remaining unexposed. Further inspection revealed that the failure to expose these faults was not the result of an error in the augmentation process *per se*. Instead, they were caused by the particular combination of existing preconditions and starting database state, which did not result in the execution of a path containing the fault.

The original test suite detected 20% of the seeded faults, even though the test cases did not contain explicit checks for violations of the business rules. These cases resulted from a coincidental alignment of the behaviour tested for by the original test case and the semantics enforced by the business rule. One would hope that a well designed test case would in any case test for some of the important business rule semantics. The problem tackled by this paper is that the coverage of business rules in test suites is likely to be too low, rather than that we expect there to be no coverage at all.

7. Improving on our Results

Whilst the results presented in the previous section are encouraging, their main value is in pointing out areas where the approach described in this paper is unsatisfactory and in prioritising the avenues for future development of the ideas. The two main problems revealed are the poor scalability of the augmented test suites and the fact that some faults (20% in our case study) were not revealed by the augmented suites. In more detail, areas for improvement are:

- *Reducing the number of business rules to check*. Obviously, whether a test case is able to violate a business rule depends on what database updates it may execute. Therefore, by applying various source-code analysis techniques to the problem we can reduce the set of business rules that a test case must check to only those it may violate.
- *Reducing the scope of the check-conditions*. Typically, test cases operate on only a small portion of the database state, governed by the specific pre- and post-conditions imposed on it. However, at present, our augmented test cases check the entire database state for validity of the rules, when only that portion modified by the programs need be verified. This results in considerable wasted time during test execution. The scalability of the approach will be improved, therefore, if we can reduce the scope of the check-conditions to the data set affected by the test case.
- *Encouraging rule violations*. The detection of certain faults is reliant on the inputs to the test case — either a certain value must be input to the program or the database must contain certain tuples. For example, consider the rule that states 'a customer cannot have more than 3 unpaid orders' — if we are verifying a method that adds an order to the database then we are much more likely to expose a failure in the rules implementation if the customer placing the new order already has 3 unpaid orders. Ideally, we would like to modify the test case so that inputs are selected for test execution that are more likely to violate the business rule under test.
- *Taking account of historical data*. Our approach checks the whole database for validity of the business rule before making the test. Although business rules evolve over time, the data in the database is typically much more static. This means that old data may be in the database that does not satisfy the current set of business rules. It is not there in error, and cannot be changed or deleted to fit the modern business rules, as then it would no longer be a fair and accurate record of the transaction as it actually took place. Thus, the presence of historical data complicates the testing of rule implementations and provides another incentive to reducing the scope of the check-conditions.
- *Testing Rule Engines*. Many information systems as well as implementing business rules in the programs source code, may include rules centralised in a rule repository and enforced by means of a rule engine. The presence of these two separate but interacting locations of business rules not only complicates the process of testing the system but also of diagnosing what caused a particular test case to fail.

Additionally, we must expand the class of business rules we are capable of verifying. Business rules can express almost any business concept and as such the classes of business rules that may be encountered in industry is very large. Also, there exists a wide range of different languages that can be used to express a business rule (i.e., OCL[14], Business Rules Markup Language[15] and BPMN[16] to name but a few) — apart from language differences, many of these languages allow different classes of business rules to be expressed. As such, we have designed our approach around a plugin architecture allowing different languages to be integrated easily within our tool.

14 http://www.uml.org
15 http://www.research.ibm.com/rules/
16 http://www.bpmn.org

8. Conclusions

In this paper, we have discussed the problem of testing business rules and have elicited the key features of a good test suite for a collection of business. To address these requirements, we have presented an approach to testing constraint business rules — an important class of rules that are typically applied to the persistent data managed by an information system. Our approach aims to verify if a test case has violated a business rule by query the database after the test has executed. In order to relieve the tester from having to design the tests individually, our approach (and associated tool) automatically augments existing test suites with the necessary check-conditions.

Our experimental evaluation, whilst being small scale and of a preliminary nature, provided us with encouraging results. The cost of augmenting the existing test suite was minor. Whilst the cost of executing the test suite increased with the increase in the number of business rules it was not prohibitively large. These results highlight the need for further study into the practicality of our approach when applied to a large industrial case study.

Acknowledgements

David Willmor is supported by a research studentship from the UK Engineering and Physical Sciences Research Council.

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] M. Bajec and M. Krisper. A methodology and tool support for managing business rules in organisations. *Information Systems*, 30(6):423–443, 2005.

[3] S. M. Embury and J. Shao. Analysing the impact of adding integrity constraints to information systems. In *Proceedings of the 15th International Conference on Advanced Information Systems Engineering (CAISE)*, volume 2681 of *Lecture Notes in Computer Science*, pages 175–192. Springer, June 2003.

[4] F. Haftmann, D. Kossmann, and A. Kreutz. Efficient regression tests for database applications. In *Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 95–106. Online Proceedings, January 2005.

[5] F. Haftmann, D. Kossmann, and E. Lo. Parallel execution of test runs for database application systems. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*, pages 589–600. ACM, August 2005.

[6] D. Hay and K. Healy. Defining business rules - what are they really, GUIDE Business Rule Report. http://www.businessrulesgroup.org/, 2000.

[7] H. Herbst. Business rules in systems analysis: a meta-model and repository system. *Information Systems*, 21(2):147–166, 1996.

[8] R. J. K. Jacob and J. N. Froscher. A software engineering methodology for rule-based systems. *IEEE Transactions on Knowledge and Data Engineering*, 2(2):173–189, 1990.

[9] G. M. Kapfhammer and M. L. Soffa. A family of test adequacy criteria for database-driven applications. In *Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 98–107. ACM, September 2003.

[10] L. Lin, S. M. Embury, and B. Warboys. Facilitating the implementation and evolution of business rules. In *Proceedings of the 21st International Conference on Software Maintenance (ICSM)*, pages 609–612. IEEE Computer Society, September 2005.

[11] J.-M. Nicolas. Logic for improving integrity checking in relational data bases. *Acta Informatica*, 18(3):227–253, December 1982.

[12] D. S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, 1995.

[13] R. G. Ross. The business rule approach. *IEEE Computer*, 36(5):85–87, 2003.

[14] J. Shao and C. Pound. Extracting business rules from information systems. *BT Technology Journal*, 17(4):179–186, 1999.

[15] S. D. Urban. ALICE: An Assertion Language for Integrity Constraint Expression. In *Proceedings of the 13th International Annual Computer Software and Applications Conference (COMPSAC)*, pages 292–299, September 1989.

[16] D. Willmor and S. M. Embury. Exploring test adequacy for database systems. In *Proceedings of the 3rd UK Software Testing Research Workshop (UKTest)*, pages 123–133, September 2005.

[17] D. Willmor and S. M. Embury. A safe regression test selection technique for database–driven applications. In *Proceedings of the 21st International Conference on Software Maintenance (ICSM)*, pages 421–430. IEEE Computer Society, September 2005.

[18] D. Willmor and S. M. Embury. An intensional approach to the specification of test cases for database systems. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, pages 102–111. ACM, May 2006.

[19] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequay. *ACM Computing Surveys*, 29(4):366–427, 1997.