Towards Automatic Test Case Generation for Industrial Software Systems Based on Functional Specifications

Arvin Zakeriyan[1], Ramtin Khosravi[1][0000−0001−6393−0959], Hadi Safari[1], and Ehsan Khamespanah[1]

School of Electrical and Computer Engineering, University of Tehran, Tehran, Iran
{a.zakeriyan,r.khosravi,hadi.safari,e.khamespanah}@ut.ac.ir

Abstract. High-capability software services, like transaction processing systems, need to satisfy a range of non-functional characteristics such as performance, availability, and scalability. To fulfill these needs, the core business logic is usually extended with a large amount of non-domain logic in the form of frameworks, libraries, and custom code, which sometimes cannot be cleanly separated from the domain logic. So, it is nearly impossible to generate test cases for the whole system systematically guided by structural metrics on the source code. In this paper, we propose a specification-based approach to generate test cases. In this approach, the domain logic is specified in a functional notation (based on Gallina). Test cases are generated using a search-based approach where the fitness function is defined in terms of the structural coverage of the specification (measured over an equivalent Haskell implementation). An experiment on an industrial stock exchange trading engine indicates promising results in the effectiveness of our proposed approach.

Keywords: Automatic Test Case Generation · Formal Specification Based Testing · Search Based Testing

1 Introduction

In today's computing landscape, we often encounter software services that must operate under various quality constraints such as performance and availability. This happens both in enterprise information systems, or the back-ends of Internet-based public services. Examples include stock trading matching engines that must serve the requests in an extremely short response time, airline reservation systems that must handle a large load of requests before national holidays, or highly available electronic funds transfer switches handling card-based financial transactions reliably. For this kind of software service, to which we refer as *high capability software service*[1], the processing is often broken into several short-lived requests (sometimes referred to as transactions). While they are often

[1] The term "high capability" has been borrowed from the title of [7].

under critical architectural forces such as performance, scalability, and availability, other non-functional requirements such as maintainability, security, etc. must be also met as other types of software systems. As such systems often play important roles, especially in financial domains, making sure they are functionally correct is as important as satisfying non-functional requirements.

The problem is that, the force to achieve the wide range of critical quality attributes brings in many non-domain elements into the system, ranging from third-party frameworks and libraries at the architecture level (e.g., in-memory object caches) down to custom optimizations in the implementation or even code hacks at the system level (e.g., bypassing TCP stack to reduce latency). While these elements affect the non-functional characteristics of the system, their combination may impact the functional correctness of the system as well. Hence, even having separately tested the domain logic during unit testing stage, it is crucial to apply a disciplined end-to-end system test to make sure that the system behaves correctly in presence of these non-domain elements.

In this paper, we report the method we developed to test a high-capability stock trading engine, with sub-millisecond response time to trading requests over hundreds of symbols and millions of shareholders. The system is mainly implemented in Java and employs many sophisticated techniques to achieve the required quality. The system already has two sets of automated tests: about 1500 unit test case in JUnit[2], and a set of about 1100 Behavior-Driven Development (BDD) tests [1], all written by the development team. Our testing approach is completely black-box and does not depend on any knowledge about the implementation of the system, hence we believe our method is applicable to many other software systems as well.

We have taken a specification-based approach to testing for two reasons. First, as said before, our goal is to make sure the system functions correctly in presence of elements embodied to meet non-functional requirements. So, we may need relatively large test cases that drive the elements to their boundaries to explore various possible scenarios. Hence, using manual oracles is not feasible while a formal specification may serve as the test oracle. Second, to attain a certain level of confidence about the functional reliability of the system, a disciplined approach must be taken to make sure the testing has met some kind of coverage criteria. Unfortunately, as the source code of the system is relatively large and the domain logic is mixed a lot with non-domain logic, defining a suitable coverage metric on the implementation code is practically infeasible. Therefore, the specification serves as a basis for disciplined test case generation too.

We have chosen Gallina (the specification language of Coq [6]), to specify the functionality of the system for three reasons. First, the functional nature of the language makes it appropriate for specifying the order matching algorithms, which are explained procedurally in the informal specifications, compared to a pre- and post-condition specification style. Second, we can use Coq to validate the specifications against some general properties. Finally, by automatically translating the specifications into Haskell programs, we make use of various tools

[2] http://junit.org

such as coverage or symbolic execution tools. We will present more detail on the specification in Section 3.

As the system under test is not small, using symbolic execution to generate test cases systematically is infeasible. So, we take a search-based approach to test case generation. The fitness function is defined based on the specification coverage and is measured on the automatically generated Haskell code from the specification. We have used both simulated annealing and genetic algorithms to generate test cases. Our early results indicate the latter produces better results. The test case generation method is explained more in Section 4.

We have detected two important faults using our generated test cases. To further evaluate the method, we performed mutation analysis and coverage measurements. The early results, reported in Section 5, indicate that attaining the same quality as the manually written tests requires considerably less effort using our method.

In summary, the main contribution of this paper is to demonstrate the applicability of formal specification-based testing to a relatively large data-intensive software system, highly constrained to non-functional requirements. Furthermore, the use of functional notation as the formalism for specification-based testing is novel as the existing research mainly base their specification on pre- and post-conditions, algebraic specification, or automata-theoretic formalisms [12]. Finally, guiding the search-based test case generation based on a formal specification is done for the first time, to the best of our knowledge.

2 Background

The system under test in this paper is a stock market order matching engine[3] as a part of an electronic trading platform which is under development and is to be used in Tehran Stock Exchange. The matching engine supports handling orders of various types (e.g., Limit, Market to Limit, Iceberg), as well as a number of order time or volume qualifiers (e.g., 'Minimum Quantity' or 'Fill and Kill'). Also, several pre- and post-trade checks must be made (e.g., brokers' credit limit, and shareholders percentage ownership).

As other products of the same category, the matching engine is supposed to handle the incoming requests within a very low response time (sub-milliseconds). Hence, the system has a rather complicated design because of a variety of design decisions and techniques at different levels. Examples include using Disruptor framework[4] to increase performance, Project Lombok[5] to increase productivity and maintainability, and extensive use of Spring Framework as an inversion of control container to make the product configurable and testable. As using an external database increases the response time considerably, the system contains several custom data structures designed to efficiently handle a large amount of in-memory data (regarding stock symbols, shareholders, brokers, etc.). In

[3] https://en.wikipedia.org/wiki/Order_matching_system
[4] http://lmax-exchange.github.io/disruptor/disruptor.html
[5] https://projectlombok.org

addition to architecture and design level decisions, a lot of implementation-level optimizations has been made to speed up the computations.

2.1 Running Example: Matching Limit Orders

As the reader may not be familiar with the domain of stock trading, we explain a simplified matching algorithm and use it as a running example throughout the paper. *Limit order* is one among several types of orders usually supported by matching engines. "A limit order is an order to buy or sell a stock at a specific price or better."[6] This means that a buy order can only be matched with a sell order of a price no more than its limit price, and a sell order can only be matched with a buy order of a price no less than its limit price. In its simplest form, a limit order contains an order identifier, the stock symbol, the limit price, the quantity, as well as the identifiers of the shareholder and/or broker issuing the order. A limit order may not be executed immediately, as an opposite order with a matching price may not be found. In such a case, the order enters the *order book*. The order book contains two separate priority queues of the orders in the system, one for *buy* orders and another for *sell* orders. Buy orders with the highest prices and sell orders with the lowest prices rank the highest on their respective queues. The orders with the same rank are prioritized according to their arrival times.

Order book before matching

Buy			Sell		
ID	Price	Qty	ID	Price	Qty
1	50	500	4	55	500
2	40	1000	5	60	300
3	40	800	6	70	1000

Input Buy Order

ID	Price	Qty
7	60	1200

Order book after matching

Buy			Sell		
ID	Price	Qty	ID	Price	Qty
7	60	400	6	70	1000
1	50	500			
2	40	1000			
3	40	800			

Output Trades

BID	SID	Price	Qty
7	4	55	500
7	5	60	300

Fig. 1. An example of matching limit orders: the input order (ID 7) is partially matched with the two topmost orders in the sell queue, generating the two trades (on the right). The remaining quantity is added to the buy side of the order book.

As an example, the left side of Fig. 1 shows the order book of some stock symbol in the system when a new limit order arrives. The system tries to match the new order with the orders in the opposite side queue (sell queue in this case), and tries to match as much quantity as possible. In this example, the two

[6] https://www.sec.gov/fast-answers/answerslimithtm.html

topmost sell orders are matched, but the third one has a sell limit price higher than the new order's buy limit price. So, the remaining quantity of the new order is inserted into the buy priority queue. Note that a trade price is always the price of the order taken from the queue.

Another type of order is *iceberg* order type, which is similar to limit order type except that when an iceberg order enters the *order book*, only a portion of its quantity (called *disclosed quantity*) can be traded. When all of the disclosed quantity is matched, the iceberg order loses its time priority and treated as if it just entered the queue with the quantity equal to its *disclosed quantity*. This process is repeated until all of its quantity is matched. Apart from order types, an order can have a number of time or volume quantifiers. As an instance, an order with a *minimum quantity* quantifier, is executed only if a certain quantity can be traded instantaneously (i.e., before entering the queue). For example, if the new limit order in Fig. 1 has a minimum quantity attribute of value 1000, it is not executed in our case, i.e., the order is rejected and no trade is made. On the other hand, if the value of minimum quantity attribute is 500, then the result would be exactly the same as the one illustrated in the figure.

3 Functional Specification

In this section, we give an overview of the method used for specifying the system under test with the aim of test case generation. We need a specification approach that focuses on modeling computations over rich data models, provides abstraction mechanisms to enable concise specification of a large system, is supported by a mature toolset, and is acceptable by software engineers.

The first requirement ruled out automata-based notations which focus more on state-based modeling of control-intensive systems. We examined several other methods, including Event-B[7], Alloy[8], pure functional specification, and even small subsets of imperative programming languages. Based on the simple prototypes made, the functional approach seemed to best satisfy the above requirements. Features like algebraic data types, recursive computations on lists, and rich abstraction patterns based on higher-order functions enabled a readable and concise specification of the matching engine. Our first prototypes were in Haskell, and based on its success, we ported the prototypes to Gallina to enable formal analysis using Coq.

Although our basic specification pattern follows a model-based paradigm, relating the states before and after handling a request using a functional description is more understandable for the developers compared to the logical specification based on pre- and post-conditions. This is because most software engineers are familiar with functional programming, especially now, with the extensive use of the elements of functional paradigm in mainstream programming languages.

A benefit of using functional specification is that while its core concepts such as recursive functions and higher-order function are familiar to the developers,

[7] http://event-b.org
[8] https://alloytools.org

the stateless nature of the computation greatly simplifies the specification of domain logic compared to the imperative paradigm. The fact that efficiency has been a main concern in developing the system under study, made the developers prefer "update-style" over "copy style" handling of changes in the system states. At several points, the developers were surprised when they saw how simple and concise a functional specification could model a rather complicated implementation of a feature.

3.1 Basic Specification Patterns

The system specification is basically an abstract state machine. The system receives requests of various types to process. Processing a request generates a response and changes the state of the system. At an abstract level, if *Request* and *Response* represent the set of all requests and all responses respectively, and *State* represents the set of states of the system, we have handlers of type $Handler = Request \times State \to Response \times State$. Each request type is handled by a separate handler function.

Data Parameters and States To represent data parameters, we need to model basic entities in the system. The definitions needed to specify our running example is listed below.

```
Definition OrderID := nat.
Definition Quantity := nat.
Definition Price := nat.
Inductive Side : Type := Buy | Sell.

Record Order := order
{ oid : OrderID
; price : Price
; quantity : Quantity
; minqty : option Quantity
; side : Side
}.
```

```
Definition OrderQueue := list Order.

Record OrderBook : Type := orderBook
{ buyQueue : OrderQueue
; sellQueue : OrderQueue
}.

Record Trade : Type := trade
{ priceTraded : Price
; quantityTraded : Quantity
}.
```

Based on these definitions, the state of the system can be defined like:

```
Record State : Type := state
{ orderbook : OrderBook
; creditinfo : CreditInfo
; ownershipinfo: OwnershipInfo
}.
```

The first field denotes the order book which is a major part of the system state. Other parts are required to enable credit limit and percentage ownership checks which we have not included in the example. The definitions of the records for modeling requests and responses are straightforward and not listed to save space.

Request Handler Functions Each request type in the system is modeled each by a request handler function. The handler function for new order requests can be modeled conforming to the mentioned pattern for handlers. It delegates the request to the function that matches new orders.

```
Definition handleNewOrder (rq : Request) (s : State) : Response * State :=
  (* unbox and delegate the request to matchNewOrder *)

Definition matchNewOrder (o : Order) (ob : OrderBook) : OrderBook * list Trade :=
  match side o with
    | Buy ⇒
      let '(rem, sq, ts) := matchBuy o (sellQueue ob) in
        match rem with
          | None ⇒ (orderBook (buyQueue ob) sq, ts)
          | Some o' ⇒ (orderBook (enqueueBuy o' (buyQueue ob)) sq, ts)
        end
    | Sell ⇒ (* similar to Buy case, skipped in the example *)
  end.
end.
```

Finally, `matchBuy` is the function that implements the matching algorithm (here, just for limit order type). It matches a buy order $o = (i, p, q, mq, Buy)$ against the sell queue *sellq* and returns a triple whose first component is the 'remainder of o' after possible matching which is then queued by `matchNewOrder` above. Since it is possible that o is fully executed, the type of the parameter is defined as `option Order` which may be either `None`, indicating the order is fully executed, or `Some o'` where o' is the remainder of o. The second component is the sell queue after matching and the third component is the list of trades made. As the definition is rather straightforward, we do not go into more details here.

```
Fixpoint matchBuy (o : Order) (sellq : OrderQueue) :
               (option Order) * OrderQueue * (list Trade) :=
  match sellq with
    | [ ] ⇒ (Some o, [ ], [ ])
    | (order i1 p1 q1 mq1 s1) :: os ⇒
      match o with
        | order i p q mq s ⇒
          if p <? p1 then (Some o, sellq, [ ])
          else if q <? q1 then (None, (order i1 p1 (q1-q) mq1 s1)::os, [trade p1 q])
          else if q =? q1 then (None, os, [trade p1 q])
          else let '(o', sellq', ts') := matchBuy (order i p (q-q1) mq s) os
               in (o', sellq', (trade p1 q1)::ts')
        end
  end.
```

Decorators As stated in Section 2, an order may have a number of qualifiers which may change the outcome of the matching, or act as pre- or post-checks to the requests. To model the qualifiers, we use a recursive pattern similar to Decorator design pattern in object-oriented design patterns [10]. A decorator is a higher-order function of type $Decorator = Handler \rightarrow Handler$, which encapsulates an 'inner handler' which itself may be composed of a number of

decorators applied on a request handler. As an example, we define the 'Minimum Quantity' qualifier described in Section 2 as a decorator:

```
1  Definition minQuantityCheck (handler : Handler) : Handler :=
2    fun rq s ⇒
3      match rq with
4        | (newOrderRequest o) ⇒
5          let '(rs, s') = handler rq s in
6           match minqty o with
7              | None ⇒ (rs, s')
8              | Some mq ⇒
9                if list_sum (List.map quantityTraded (trades rs)) <? mq then
10                 (newOrderResponse false [ ], s) (* reject the order *)
11               else
12                 (rs, s')
13           end
14       | _ ⇒ handler rq s
15     end.
```

The decorator returns a handler function that takes a request `rq` and a state `s` (line 2). The decoration is only applied on new order requests (lines 4-13), and has no effect for other request types (line 14). The request is delegated to the inner handler `handler` and the returned response and state are stored in `rs` and `s'` respectively (line 5). In case the new order has no minimum quantity qualifier, the results are returned back (line 7). Otherwise, the sum of `quantityTraded` fields of the trades made is compared to the value of the minimum quantity (line 9). If this sum is below the minimum quantity, the order is rejected (line 10). Otherwise, the results from the inner handler is returned back.

It is important to note that the implementation of this feature in the source code is much more complicated, due to the fact that the matching algorithm changes the state of the system 'in place'. This means that after the matching is completed, the trades are checked to meet the minimum quantity constraint, and if failed, all the changes to the order book must be 'rolled back'. This makes the implementation complicated, especially for more complex order types (like iceberg order type). On the other hand, the stateless nature of the functional specification makes both versions of the state available and we 'roll back' to the previous state simply by returning `s`, as in line 10.

Using the decorator pattern described here, enables us to 'compose' various qualifiers in a modular way. In the following, four decorators are applied to `handleNewOrder` (_ ∘ _ is the notation defined for function composition):

```
Definition matchingEngine : Handler :=
  creditLimitProc ∘ fillAndKillProc ∘ minQuantityCheck ∘
  (ownershipCheck 20) ∘ handleNewOrder
```

3.2 Verifying Properties on Specification

Using Gallina as the formal specification languages enables us to use Coq proof assistant to verify correctness properties on the specification. For example, the following lemma states that `matchBuy` function conserves the total quantities.

```
Lemma quantityConservation : forall o sellq,
  (queuedQuantity sellq) = totalQuantity (mb o sellq).
```

The above lemma uses two defined functions: `queuedQuantity` which calculates the sum of the quantities in an order queue, and `totalQuantity` which calculates the sum of the queued and the traded quantities. The lemma can be proved mainly using induction, case analysis, and usual rewriting tactics.

It is important to note that the informal specifications available for the matching algorithms are generally described in a procedural manner, because they are targeted for non-expert audience. So, the properties are defined by the modelers intuitively based on their understanding from the informal descriptions.

4 Test Case Generation

Following a specification-based test case generation approach, the specification may serve both as the oracle for the test inputs, and a basis for more 'smart' generation of test cases compared to plain random test case generation. To this end, we may use *specification coverage* as a measure of fitness for a generated test suite. Ideally, we may use symbolic execution to get full coverage, but it becomes impractical as the specification gets larger. Hence, we use search-based test case generation [11,2], as a middle ground between symbolic execution and plain random test case generation.

In this research, we use a local search (based on *simulated annealing*) and a global search (based on *genetic algorithm*) algorithm to generate test cases. It should be noted that our purpose is not to compare these search-based approaches but to show the possibility of using search-based test case generation from a functional specification for a complex industrial application. Before going into the details of each technique, we briefly present the fitness function used to evaluate the test suites.

4.1 Fitness Function

As mentioned before, we use the functional specification as a basis for computing the fitness of the test suites. This is done by measuring a branch coverage on the specifications. A branch in our functional specifications happens at two points: conditional ('if') expressions and pattern matching. For example, to cover the function `minQuantityCheck` listed in Sect. 3, we need to cover the matches at lines 4 and 14, the matches at lines 7 and 8, and the expressions at lines 10 and 12. We refer to each of these parts a *case*, which is an expression appearing either as a match case or an expression in an if-then-else expression.

The *fitness function* of a test suite TS (also known as its *score*) is defined as a linear combination of the covered cases and the length of the test cases: $s = 5c - l$, where c is the number of cases covered by test suite and l is the length of test suite (i.e., the number of test cases in the test suite) that is required to avoid generating long test cases. The aim is to *maximize* the score of the test suites,

but as both algorithms of local and global search try to minimize an objective function, we use the additive inverse of the *score* of the test suites as the objective function.

To measure coverage in practice, we use the *program extraction* framework (a part of the Coq toolset) to convert the specification into a Haskell program and instrument the Haskell code (using a monadic construct) to collect the coverage information during execution. The search algorithms (explained below) execute the Haskell program on the inputs generated during the search, and use the coverage information generated to evaluate the inputs.

4.2 Local Search

We take advantage of *simulated annealing* as a local search algorithm which uses some heuristics to escape from local optima. Python package `simanneal`[9] is used to set up a search-based test case generation framework.

We define a *state* as an integer array representing a test suite, including at most 40 test cases. Each test case can have at most 10 orders, fed to system respectively.

Every order is represented by an integer tuple encoding its *broker*, *shareholder*, *price*, *quantity*, *side*, *minimum quantity*, *Fill and Kill (FaK) Qualifier* and *disclosed quantity* (for iceberg order types). Each part is chosen from a limited domain. Each test case is encoded by concatenation of the encoded version of 10 orders, prefixed by a boolean presence permission *grant* parameter such that a test case is included in the test suite this flag is true. This parameter is added to increase chance of removing a whole test case from test suite and generating smaller test suites with smaller test cases. Empty test cases (i.e., test cases with no order) are considered malformed and is deleted from test suite. The test case also includes the *initial credits* of 5 brokers and the *initial shares* of 10 shareholders. Finally, any state, representing a test suite, is generated of concatenation of 40 test cases.

Considering the aforementioned representation of state, a *neighboring state* can be reached by choosing a random element of the integer array representation of current state, and replacing it by a randomly selected element of domain of that field. This way, the framework can *move* from a state to a neighboring state. The initial state is generated by picking random values for each element of the integer array.

4.3 Global Search

We developed another framework based on a modified version of Python package `geneticalgorithm`[10] to find suitable test suites using an elitist genetic algorithm as an example of global search techniques. In this framework, we represent a test suite with a *chromosome* in the same way we defined a *state* for the local search

[9] https://github.com/perrygeo/simanneal
[10] https://github.com/rmsolgi/geneticalgorithm

algorithm. We start with a randomly generated non-seeded population as first generation. Each generation includes 100 chromosomes. We use mutation probability of 0.1 and cross over probability of 0.5, with uniform cross over type. We also use elites ratio (determining the number of elites in the population) of 1%, which means there is one elite in the population. We set a maximum number of generations of 1000. Using seven different runs of genetic algorithm, the objective function reduced from the value corresponding to the first randomly generation (about -120 to -130) to a lower value (even -155), over 1000 iteration.

5 Evaluation

To evaluate the effectiveness of our approach by executing the test suites, two important faults in the system were discovered. To evaluate the quality of our test suites we used two methods: comparing the coverage of our tests with the coverage of the already developed unit tests of the system, and mutation analysis. The source code of the system is about 80K lines of code, mainly written in Java. The development team has put a lot of effort in writing unit tests and BDD-style functional tests. There are more than 1500 test cases in the source code, devoting about 40% of the code base to test. It is estimated that the team has invested more than 30% of the development time on developing these tests over the course of two years development.

5.1 Test Execution

To eliminate the effects of randomness in our generated test suites, we generated 10 test suites using each test case generation algorithm and executed each on the system under test separately. To execute the generated test suites against the system, an adaptation layer was needed. This adapter is developed with the help of the development team. It takes a test case specification file including the test fixture and the sequence of orders with all their required attributes, initializes the system with respect to the specified fixture and feeds the orders to system. After sending all the orders to system, the adapter collects system outputs and compares them with the test oracles in the test case specifications.

Upon execution of tests generated using both Genetic Algorithm and Simulated Annealing, a number of test cases failed. As we discussed these cases with the development team, it became clear that there were actually two important faults in the matching engine and the results of the tests were correct. One of them was a case of missing a specific validation check on iceberg orders which lead to wrong matching. The other was a case of incomplete rollback of trades which leads to incorrect credit values for the brokers. These faults have not been detected by any of the unit or BDD tests.

5.2 Code Coverage Analysis

One way to analyze the quality of the generated test suites is to analyze the branch coverage of the source code obtained by our generated test suites com-

Table 1. Branch coverage percentage of two methods compared to the development team's tests

Target Class	Dev. Tests	Simulated Annealing	Genetic Algorithm
Matching Engine	60	59.5 ± 0.5	59.8 ± 0.37
Order	85	50.7 ± 2.2	52 ± 3
Iceberg	100	83 ± 0	83 ± 0
Queue	87	70.9 ± 10.3	74.8 ± 7.2
PO Observer	70	70 ± 0	70 ± 0
Credit Observer	79	77.4 ± 2	78.5 ± 1.5

pared to the the branch coverage of the unit tests of the system. By branch coverage we mean the percentage of the true and false branches of the conditional constructs in the source code (e.g., 'if' statements and loop conditions) that are executed at least once. Although there are stronger coverage criteria like prime path coverage, using these methods is impractical in our case, since to the best of our knowledge there is no automatic tool to calculate these metrics on an enterprise application scale code base. To calculate the branch coverage of the test suites, we used JaCoCo[11] which is a well-known coverage analysis industrial tool.

We ran all unit and BDD tests in the system and computed the branch coverage and mutation score for them as the baseline for comparison. It must be noted that since our specification does not yet fully cover the logic of matching engine, only those classes that implement the specified part of the logic are selected for comparison.

Table 1 shows the results of the branch coverage analysis for the three test suites: the tests developed by the development team (Dev. Tests), and the ones generated by our two algorithms. The results present the average and the standard deviation of the branch coverage across the test suites for each method. In each test suite for Genetic Algorithm there are 15 test cases on average. For Simulated Annealing this number is about 19 test cases per test suite. As the table shows, in many of the classes the branch coverage of the Genetic Algorithms and Simulated Annealing is almost equal to the branch coverage of the development team's tests. The difference of the coverage in Order, Iceberg and Queue classes is due to some methods that were not yet modeled in specification. Also, the number of test cases generated for each test suite for both methods is much less than the development test suite which reduces the execution time of test suite.

5.3 Mutation Analysis

The second method used to measure the effectiveness of test cases is mutation analysis. To this aim, we used Pitest[12] tool which is widely used for mutation testing in the domain of enterprise applications. It supports a wide variety of mutation operators, from which we took its default mutation operator set.

[11] https://www.jacoco.org
[12] https://pitest.org

Table 2. Mutation percentage of two methods compared to the development team's tests

Target Class	Dev. Tests	Simulated Annealing	Genetic Algorithm
Matching Engine	48	55.6 ± 2.8	57.1 ± 2.1
Order	60	42.6 ± 1.7	46.2 ± 6.9
Iceberg	78	54.6 ± 4	56 ± 0
Queue	73	49.2 ± 7	56 ± 12.3
PO Observer	38	48.6 ± 8	63.8 ± 15.7
Credit Observer	75	57 ± 26.2	72.2 ± 22.9

As the mutation framework cannot calculate the mutation score where there is a test failure, we had to exclude the test cases that produced the two discovered faults from the test cases to calculate the mutation score. Table 2 shows the mutation scores, i.e., the percentage of the mutants that were killed by the test suites.

As table 2 shows, in many of the classes, more mutants were killed in test generated by both our methods compared to the development team's tests. Like in the case of branch coverage, in Order, Iceberg and Queue classes the mutation score is lower in our tests, because those classes contain logic that has not been specified yet. Also, there is a big variance in Credit and PO classes. This is because, as mentioned, some test cases had to be excluded from the test suites so that we could calculate the mutation score. These classes were the target of those tests since the bugs were related to logic in these classes. As a result, some test suites where heavily affected by the exclusion of the test cases, causing a big variance in the mutation score.

The results of both evaluation methods show that the tests generated by our method are at least as effective as the tests that were generated by domain experts and developers. This is achieved by a much smaller test suite. The effort to develop the unit and BDD tests was much more than the effort put to develop the specification and generate the required adapters to run the tests (approximately 60 man days).

6 Related Work

Formal Methods for the Analysis of Auction Theory. There is a limited number of works in the formal analysis of trading algorithms in financial market. Sarswat et al. in [16] formalized various notations for auction theory which are required for the analysis of trades in financial markets. These notations were implemented in Coq and authors defined properties like fairness, uniformity, maximality, and individual rationality using the defined notations. Lange et al. in [14] showed how theorem proving tools can be used for the analysis of the consequences of different options in auction designs. In comparison with these works, the focus of this paper is in the implementation of auction theories not in their specifications.

Formal Methods in Automatic Test Case Generation for Enterprise Applications. Asaadi et al. applied model-based testing techniques to an Electronic Funds Transfer (EFT) switch [4]. They used ISO 8583 specification to provide a formalization of the transaction flows in terms of a labeled transition system. They used input space partitioning to generate test data and the formalization for model-based testing based on input-output conformance testing. In comparison with this work, we use feedback for the better generation of test cases; however, in [4] test cases are randomly generated. In addition, they used Timed Automata [3,5] to specify ISO 8583 which cannot be used for our case, as it is data-intensive application not a control-intensive application. In contrast, Liu et al. in [15] proposed Vibration-Method for automatic generation of test cases and test oracle from model-based formal specifications, which is tailored for testing information systems in which rich data types are used. This method provides automatic test case generation based on functional scenarios, test case generation criteria, and a mechanism for deriving test oracle. The main difference between the work of [15] and this paper is in the specification language. The functional nature of the specification language of this paper makes it appropriate for specifying the order matching algorithms, which are explained procedurally in the informal specifications, compared to a pre- and post-condition specification style of [15]. In addition, by automatically translating the specifications into Haskell programs, we make use of various tools such as coverage tools. There are also some works on automatic test case generation for software components using formal method, e.g. [13,8]. These works focus on the detailed description of the behavior of components and cannot be generalized for testing an enterprise application.

7 Conclusion

In this paper, we reported our experience in applying specification-based testing to an industrial enterprise software system. The system is specified in a functional language which has a formal foundation and is not too difficult to work with for an average software engineer. We developed simple specification patterns such as the decorator which modularizes the specification and make it easier to develop and understand. Our experience showed that the size of the specification is much smaller than the functional tests developed by the development team, and could be developed by a considerably less effort. The test case generation is guided by the specification coverage with the aim of taking all corner cases into account. The overall results indicated that we can attain the same coverage and mutation score with much less effort.

In addition to the work needed to make the specification fully cover the domain logic, there is more room for improvement in several aspects. As an example, more sophisticated fitness functions may be used during test case generation, such as the ones incorporating branch distance measures [9]. Also, more patterns and abstraction mechanisms may be developed to further simplify the specification and make it more understandable.

References

1. Adzic, G.: Specification by Example: How Successful Teams Deliver the Right Software. Manning Publications (2011)
2. Ali, S., Briand, L.C., Hemmati, H., Panesar-Walawege, R.K.: A systematic review of the application and empirical investigation of search-based test case generation. IEEE Transactions on Software Engineering **36**(6), 742–762 (2009)
3. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science **126**(2), 183–235 (1994)
4. Asaadi, H.R., Khosravi, R., Mousavi, M.R., Noroozi, N.: Towards model-based testing of electronic funds transfer systems. In: Fundamentals of Software Engineering - 4th IPM International Conference, FSEN. LNCS, vol. 7141, pp. 253–267. Springer (2011)
5. Bengtsson, J., Larsen, K.G., Larsson, F., Pettersson, P., Yi, W.: UPPAAL - a tool suite for automatic verification of real-time systems. In: Proc. of the DIMACS/SYCON Workshop on Verification and Control of Hybrid Systems. LNCS, vol. 1066, pp. 232–243. Springer (1995)
6. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series, Springer (2004)
7. Dyson, P., Longshaw, A.: Architecting enterprise solutions: patterns for high-capability Internet-based systems. John Wiley & Sons (2004)
8. Fang, Y., Zhu, H., Zeyda, F., Fei, Y.: Modeling and analysis of the disruptor framework in CSP. In: IEEE 8th Annual Computing and Communication Workshop and Conference, CCWC. pp. 803–809. IEEE (2018)
9. Fraser, G., Arcuri, A.: Whole test suite generation. IEEE Transactions on Software Engineering **39**(2), 276–291 (2012)
10. Gamma, E., Helm, R., Johnson, R., Vlissides, J.M.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, 1 edn. (1994)
11. Harman, M., McMinn, P., De Souza, J.T., Yoo, S.: Search based software engineering: Techniques, taxonomy, tutorial. In: Empirical software engineering and verification, pp. 1–59. Springer (2010)
12. Hierons, R., Bogdanov, K., Bowen, J., Cleaveland, R., Derrick, J., Dick, J., Gheorghe, M., Harman, M., Kapoor, K., Krause, P., Lüttgen, G., Simons, A., Vilkomir, S., Woodward, M., Zedan, H.: Using formal specifications to support testing. ACM Comput. Surv. **41**(2) (2009)
13. Kong, L., Zhu, H., Zhou, B.: Automated testing EJB components based on algebraic specifications. In: 31st Annual International Computer Software and Applications Conference, COMPSAC. vol. 2, pp. 717–722. IEEE (2007)
14. Lange, C., Caminati, M.B., Kerber, M., Mossakowski, T., Rowat, C., Wenzel, M., Windsteiger, W.: A qualitative comparison of the suitability of four theorem provers for basic auction theory. In: Intelligent Computer Mathematics - MKM, Calculemus, DML, and Systems and Projects, Held as Part of CICM 2013. LNCS, vol. 7961, pp. 200–215. Springer (2013)
15. Liu, S., Nakajima, S.: Automatic test case and test oracle generation based on functional scenarios in formal specifications for conformance testing. IEEE Transactions on Software Engineering pp. 1–1 (2020). https://doi.org/10.1109/TSE.2020.2999884
16. Sarswat, S., Singh, A.: Formally verified trades in financial markets. arXiv preprint arXiv:2007.10805 (2020)