FinExpert: Domain-Specific Test Generation for FinTech Systems

Tiancheng Jin*
East China Normal University, China
51184506019@stu.ecnu.edu.cn

Qingshun Wang*
East China Normal University, China
wqseleven@gmail.com

Lihua Xu†
New York University Shanghai, China
lihua.xu@nyu.edu

Chunmei Pan
CFETS Information Technology Co. Ltd., China
panchunmei_zh@chinamoney.com

Liang Dou†
East China Normal University, China
ldou@cs.ecnu.edu.cn

Haifeng Qian
East China Normal University, China
hfqian@cs.ecnu.edu.cn

Liang He
East China Normal University, China
lhe@cs.ecnu.edu.cn

Tao Xie
University of Illinois at Urbana-Champaign, USA
taoxie@illinois.edu

ABSTRACT

To assure high quality of software systems, the comprehensiveness of the created test suite and efficiency of the adopted testing process are highly crucial, especially in the FinTech industry, due to a FinTech system's complicated system logic, mission-critical nature, and large test suite. However, the state of the testing practice in the FinTech industry still heavily relies on manual efforts. Our recent research efforts contributed our previous approach as the first attempt to automate the testing process in *China Foreign Exchange Trade System (*CFETS*) Information Technology Co. Ltd.*, a subsidiary of *China's Central Bank* that provides China's foreign exchange transactions, and revealed that automating test generation for such complex trading platform could help alleviate some of these manual efforts. In this paper, we investigate further the dilemmas faced in testing the *CFETS* trading platform, identify the importance of domain knowledge in its testing process, and propose a new approach of domain-specific test generation to further improve the effectiveness and efficiency of our previous approach in industrial settings. We also present findings of our empirical studies of conducting domain-specific testing on subsystems of the *CFETS* trading platform.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

*Both authors contributed equally to this work.
†Both authors are corresponding authors.

ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia

ACM ISBN 978-1-4503-5572-8/19/08…$15.00
https://doi.org/10.1145/3338906.3340441

KEYWORDS

FinTech; automated test generation; domain knowledge

ACM Reference Format:
Tiancheng Jin, Qingshun Wang, Lihua Xu, Chunmei Pan, Liang Dou, Haifeng Qian, Liang He, and Tao Xie. 2019. FinExpert: Domain-Specific Test Generation for FinTech Systems. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3338906.3340441

1 INTRODUCTION

Financial Technology, or FinTech for short, is becoming increasingly widespread in financial institutions. According to Klynveld Peat Marwick Goerdeler, global FinTech funding rose to $111.8 billion in 2018, up 120 percent from $50.8 billion in 2017 [6]. Various software systems are specially designed and utilized nowadays to automate and accelerate financial services. Such systems are often mission critical: any unexpected failures or wrong-doing may lead to huge financial losses. However, their complicated underlying business logic and numerous domain-specific inputs pose great challenges for creating tests that can sufficiently exercise these systems, let alone automating the testing process.

To ensure high quality of their services, many financial institutions have employed groups of testers to manually test their FinTech systems with high manual efforts. One example of such financial institutions is our industrial collaborator, *China Foreign Exchange Trade System (*CFETS*) Information Technology Co. Ltd.*. As a subsidiary of *China's Central Bank*, *CFETS* provides services of currency swap and exchange rate swap for tens of thousands of active clients, and involves 150 trillion US dollars of transactions per year. For this complicated yet mission-critical trading platform, even a simple change in one of its subsystems can lead to months of manual testing of the whole system, not to mention the huge financial losses that it may suffer from due to even a small wrong-doing in the system.

To reduce the burden of testers, our recent efforts [14] have improved the comprehensiveness of the created test suite and the efficiency of the adopted testing process by delivering automated testing mechanisms to *CFETS* in two important ways. First, we conduct automatic test-data generation for subsystems in the trading platform, and demonstrate how our approach accomplishes substantial branch-coverage improvements over manual testing. Second, we propose an improved differential testing technique that addresses the problem of lacking testing oracle.

Our such recent efforts have been just the beginning of a journey started by the academia-industry partnership formed by the authors of this paper. The goal of this partnership is to bridge the gap between the FinTech industry and the research community in software testing, improve the state of the practice in the area of automated test generation for FinTech systems, and produce high industry impact alongside impact on the research community. Accomplishing this goal requires continuous dedicated efforts to gain further understanding on "what did not work and why", and develop practical solutions to tackle these identified challenges when transferring techniques into industrial practices.

To further investigate the current testing practices in the FinTech industry, we consult with the testing department of *CFETS* to learn that manual efforts especially those from domain experts (who have valuable domain knowledge) are still heavily relied upon for testing service-intensive subsystems because of their complex underlying logic. The testers of *CFETS* have to work side-by-side with the domain experts. The domain experts, who understand deeply the whole trading platform and financial logic, design test scenarios and data constraints; the testers then generate test data accordingly. In other words, the whole testing process is often inefficient, heavily relying on the domain experts and their maturity. Even worse, the business logic, especially most of the underlying financial logic, is implicit and un-documented, and is passed down as informal experiences. It is not unusual for the domain experts to spend months, even longer, in understanding the trading services and their complex logic. It is also very likely for the entire testing process to be stalled when current domain experts leave. Furthermore, the collaboration between the domain experts and their corresponding testers may not work well and can elongate the testing process, e.g., the testers understand the data constraints differently from the domain experts; the domain experts miss corner cases when designing the testing scenarios.

After confirming the importance of domain knowledge in testing FinTech systems, we closely investigate the situations where our previous approach [14] falls short, e.g., about 20% of the branches from certain subsystems of the trading platform have never been covered. With the help of our industrial partners, we identify two main sources of ineffectiveness: lacking domain knowledge of data-field dependencies and lacking domain knowledge of exceptional cases of input data to guide automatic test generation.

Lacking domain knowledge of data-field dependencies. Many not-covered branches in the certain subsystems of the trading platform are related to special business logic that can be triggered only when the input data meets certain special conditions that reflect implicit dependencies between different data fields. For instance, only when input data contains data fields representing a RMB-gold transaction and completion time as a specific time period, the corresponding code branches can be triggered. These implicit data dependencies have high probability of being missed by a brute-force generation technique such as the one in our previous approach [14].

Lacking domain knowledge of exceptional cases of input data. Our previous approach [14] creates exceptional data by specifically identifying the category of unacceptable input data, and generating input data falling into this category. However, further investigation reveals that only "meaningful exceptional data" is worth being generated. Meaningful exceptional data is exceptional data that is valid but not-expected input to the systems, such as a wrong institution code for a transaction party. Such valid but not-expected data is more likely to trigger failures. Unfortunately, defining such "meaningful exceptional data" is part of the domain knowledge, not available to our previous approach [14] during automatic test generation.

To tackle the aforementioned limitations, in this paper, we propose FinExpert, a new test generation approach based on a Domain-Specific Language (DSL), to specifically document and utilize FinTech domain knowledge to guide the testing process. According to our study, most of the domain knowledge in a FinTech system can be concretized as data **type**, **range**, and **format** for each input field, as well as **dependencies** among its different input fields. By introducing our DSL to describe such constraints as rules, our test generation approach categorizes the data range of the input and its different fields, and generates data accordingly. With much accurate categorization of the input data and dependencies among data fields, our approach is able to generate a comprehensive test suite. Even better, although domain experts are still needed for writing the constraint rules, the business logic and its underlying implicit financial rules, once documented, can now be passed down, and from these rules test cases can be automatically generated directly. In contrast to the current practice, where domain experts learn from their precedents and testers manually generate test inputs accordingly, FinExpert is able to not only save the manual efforts during the phase of test generation, but also preserve the underlying financial rules.

Additionally, **test oracles** can also be specified as domain knowledge. From our industrial partners' experiences, many FinTech subsystems expect simple predefined return codes as their execution result, denoting either a successful transaction or an error code that indicates unexpected input or exceptional behavior. For such subsystems, the data constraints for the expected result and the dependencies among input data and its corresponding result can be described in the same way as input data using our DSL, so that the expected result can be generated alongside with the input data and serve as the test oracle when evaluating the test result.

To evaluate whether introducing and documenting domain knowledge in the testing process can reduce burden on manual efforts and is able to effectively and comprehensively test a FinTech system, we apply FinExpert to multiple subsystems of the *CFETS* trading platform, including two major subsystems and one middleware. We compare the manual efforts in terms of time spent in testing the subsystems before and after utilizing FinExpert, and FinExpert's ability to reduce the size of the test suite yet preserve high code coverage of the system. The results from our comparison

show that FinExpert can save about 70% manual efforts, as well as achieving increased code coverage with less test data.

In summary, this paper makes the following main contribution:

- A case study in the FinTech industry for confirming the importance of domain knowledge in the testing process.
- A domain-specific testing approach to utilize domain knowledge for test generation.
- A Domain-Specific Language designed for concretizing FinTech domain knowledge as constraint rules.
- Empirical studies of conducting domain-specific test generation on subsystems of the *CFETS* trading platform.

In the rest of this paper, we present background information in Section II. We illustrate our motivating example in Section III, and present the overview information as well as the detailed approach in Sections IV and V, respectively. We show and discuss our evaluation results in Section VI. We present related work in Section VII. We discuss conclusion and future work in Section VIII.

2 BACKGROUND

2.1 China Foreign Exchange Trade System

*China Foreign Exchange Trade System (*CFETS*) Information Technology Co., Ltd.* is a wholly-owned subsidiary of the *China Foreign Exchange Trading Center*. Its parent company is directly under the head office of the *People's Bank of China*. As the specific organizer and operator of the Chinese inter-bank foreign exchange market, money market, bond market, and exchange rate and interest rate derivatives markets, the trading center provides an internationally advanced trading platform. In the whole year of 2018, the cumulative turnover of the market was 1,262.8 trillion, a year-on-year increase of 26.6%. By the end of 2018, there were 24,804 members in the inter-bank local currency market and 678 members in the inter-bank foreign exchange market. The *CFETS* company itself is committed to providing technology development, operation and maintenance, and information services for the main trading platform of the trading center.

2.2 Facts

To obtain a more comprehensive collection of test cases, our previous research efforts have contributed Facts [14], an automated black-box testing approach for FinTech systems. FinTech systems usually take high-dimensional inputs that contain a group of fields and each of which belongs to one of various data types or user-defined complex data structures. These characteristics bring high difficulties for test generation. Furthermore, due to the complex and diverse behavior of different systems, it is hard to find a universal test oracle for testing FinTech systems. To tackle these problems, Facts includes three main techniques as listed below.

First, Facts collects system logs from real transactions and retrieves passing messages as input seeds, to construct high-dimensional test messages; in the meantime, Facts also identifies the input category of each data field for each corresponding test message to cover all possible field categories.

Second, according to the formats of data fields in the system logs, Facts is able to partition the input domain of the system under test, and select test cases from each class of the partition. For example, for a field of data type "Date", all the valid and invalid values can be classified into different categories, respectively. Each category can be further subdivided accordingly.

Third, to derive the test oracles, Facts tests the system under test with two previous versions in parallel: the currently-deployed version and the last legacy version in the version repository, assigning different priorities with the two system versions, and constructs the test oracles based on their outputs.

3 MOTIVATING EXAMPLE

To further understand the bottleneck of the testing process in FinTech industry sectors, such as *CFETS*, we conduct a case study on the *Independent Software Vendors Protocol Translation (*ISVPT*) subsystem*, a middleware used by *CFETS* to convert different data formats during data exchange between different subsystems. We choose this particular subsystem for our case study because it resembles a typical FinTech system, with numerous system-defined data types and underlying financial logic. As illustrated in our previous efforts [14], FinTech systems expect high-dimensional input data, with each field/dimension a system-defined data type, and hence a randomly generated test suite without knowing these data types would not be able to include acceptable input data.

Our consultation with testers in *CFETS* reveals that domain experts are heavily relied on during the testing process. Hence, in this case study, we intend to specifically understand (1) how important the domain knowledge is in testing FinTech systems; and (2) what domain knowledge helps with automated testing of FinTech systems. With help from our industrial partners, we are able to acquire three different data sets of input data: one representing the current testing practices (manually written by domain experts and testers), one randomly generated (with only type information), and one generated by our previous approach [14] (test generation based on data constraints). We further describe each data set as follows.

Manually written. The first data set resembles the current state of practice in *CFETS*. Manually writing the input data is joint work done by domain experts and testers, and the data set is very small and contains only 4 inputs. Since it is the actual test data being used, this data set serves as a comparison baseline, labeled as "Set 1".

Randomly generated. The second data set is randomly generated with additional information on input data types, such as the data type of each input field. The resulting data set is generated by randomly generating a value for each input field with its corresponding type, e.g., a random number for an input field of type integer. This data set contains 1,000 data, labeled as "Set 2".

Previous approach. We would like to directly apply our previous approach [14] on the subsystem to generate test data. Unfortunately, the subsystem is newly developed and yet-to-be deployed, and thus there are no system logs available. Since the size of the manually written test cases is too small, we resort to a "compromised" version of our previous approach. In particular, our previous approach utilizes system logs to retrieve the accurate system-defined data type and range of each input field, such as the right format of date and time for each transaction. In contrast, the "compromised" version of our previous approach requires manually specified system-defined data type and range of each input field. For

Table 1: Code coverage achieved by different data sets

	Set 1	Set 2	Set 3
Code coverage	44.8%	24.3%	32.6%
Number of cases	4	1000	100

example, the three fields shown in Table 2 are required to contain an integer that is a timestamp within year 2018. Data in this data set can meet the input requirement, whereas data in "Set 2" are not guaranteed to satisfy the input requirement. This data set includes 100 data, labeled as "Set 3".

We then execute the subsystem with each data set and collect its corresponding code coverage. The result is shown in Table 1. It is not surprising to see that although the manually written data set is the smallest, its achieved code coverage is almost twice that of the second data set. The last data set is much smaller than the second data set but still outperforms the second one.

After consulting with testers in *CFETS* and conducting in-depth analysis of the data sets, we confirm that the biggest difference between these data sets is on how much domain knowledge (specifically in this case, the data type and range for each input field) is reflected in the test generation. The first data set can be considered to be generated under the guidance of comprehensive domain knowledge, while the second data set is generated using a pure brute-force approach without any guidance. Therefore, the first data set satisfies all constraints and dependency relationships required by the subsystem, because the first data set is designed by domain experts who clearly understand the domain knowledge. But the second data set is filled with random values, and it is almost impossible for the data to coincidentally satisfy all the constraints and relationships. Thus the data cannot pass the validation to trigger further behaviors thereafter. The last data set achieves better results with smaller size because it incorporates part of the domain knowledge: the specific data format and its range during the generation. But information about constraint relationships between different fields, such as dependency and transitive dependency, is missing; therefore, it is still outperformed by the manually written data set. For example, there are three fields in the input data of the ISVPT subsystem named "DateDash", "DateTime", and "DateTimeMilliSecond". Each of the three fields contains a Unix timestamp for representing the same timestamp but with unit accuracy as day, second, and millisecond, respectively. In other words, all three timestamps should point to a time in the same day, and the last two timestamps should point to the same second.

The results from this case study, as well as confirmation from our industrial partners, demonstrate that domain knowledge is crucial to generate a comprehensive test suite; and such domain knowledge in FinTech systems mainly includes the input data type and format for each input field, and the constraints and dependencies among the input fields.

4 OVERVIEW

4.1 Problem Statement

With deep understanding that domain knowledge is crucial in effectively testing FinTech systems, we identify two issues with the current human-centered testing process.

Deep learning curve. As described in Section 2.1, *CFETS* develops and provides a trading platform that requires to handle a large number of market members and huge trading volume. The requirements on the capacity, performance, and reliability are extremely high, resulting in a highly complex trading system. The entire system consists of hundreds of subsystems, and in order to respond to market efficiently, *CFETS* has to continuously update its system. Consequently it takes tremendous efforts to fully understand such system, not only functionality or business logic, but also enormous implicit financial logic. Domain experts need to be proficient in testing, and have at least 2 years of testing experience in relevant FinTech fields to derive effective test cases. Unfortunately, the current loss rate of domain experts is around 20%, and it is increasingly difficult to recruit such experienced personnel.

Quality of test data. Domain experts define test scenarios for each subsystem. Each test scenario describes a specific use case, as shown in the left side of Figure 2. Both concrete data and detailed business situations are to be included in the scenario set so that testers can prepare test data accordingly. Testers need to communicate often with their domain experts to understand the test scenarios and corresponding data constraints, in order to generate a meaningful data set. The communication process is often error-prone and greatly increases the manual efforts. Even worse, it is very common for domain experts to miss some corner cases in their test scenarios, especially the combinations of different business scenarios, due to complex financial rules.

Therefore, it is necessary to improve the current testing practice to tackle the aforementioned problems. More specifically, we investigate techniques to transform the current human-centered testing process to a knowledge-centered process, where the domain knowledge can be well structured and utilized throughout the entire testing process.

4.2 Extension of Facts

Our previous efforts [14] have proposed Facts to automatically test FinTech systems. Facts is able to improve testing effectiveness compared to the original manual tests. Motivated by the case study described in Section 3, we extend Facts to propose FinExpert by specifying the domain knowledge to guide test generation. In our work, domain knowledge specifically refers to the data **type**, **range**, and **format** for each input field, and **dependencies** among different fields of the input data, as well as the **expected return codes** as test oracles. Besides reducing manual efforts, FinExpert is able to optimize Facts in the following dimensions.

Eliminate the need of test seeds. To address the challenges of high-dimensional inputs, Facts leverages the input data collected from system logs as seeds and mutates them to generate valid data. But for a newly developed system, logs are unavailable, so Facts is not applicable. As a more applicable solution with the domain knowledge being directly specified, FinExpert collects data format and constraints of each input field and their dependencies from the specified domain knowledge, so FinExpert is able to generate test data directly without requiring any test seed.

More comprehensive yet smaller test suite. A test suite generated by Facts is always very large. However, many of the test cases in the test suite are redundant or unnecessary, and have no

Table 2: Examples of data field value in different data sets

Field	Set1	Set 2	Set 3
DateDash	1543161600000 (2018/11/26 00:00:00)	38111886081466 (3177/09/20 03:01:21)	1523377875809 (2018/4/11 00:31:15)
DateTime	1543246332000 (2018/11/26 23:32:12)	19573242030453 (2590/04/02 11:40:30)	1520882677818 (2018/03/13 03:24:37)
DateTimeMilliSecond	1543246332011 (2018/11/26 23:32:12)	44963693695 (1971/6/5 17:54:53)	1544437792249 (2018/12/10 18:29:52)

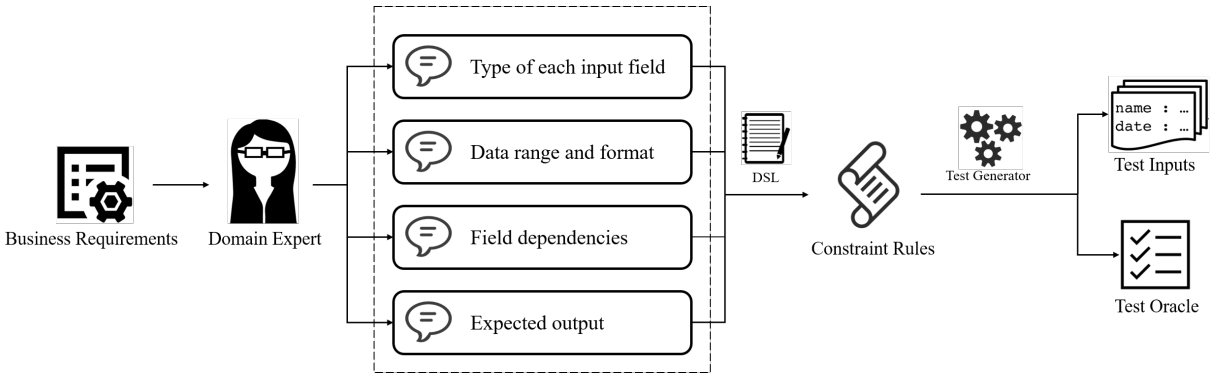


Figure 1: Domain-specific Test Generation Process

contribution to the code coverage. The reason of the redundancy is that many logs represent the same scenario with slightly different data, but Facts cannot easily identify these same-scenario logs and conducts the generation step on each of them. Even worse, our previous approach mutates each data field in isolation to generate the test data, which may break some implicit constraint among data fields. With data constraints and their dependencies explicitly specified and utilized to generate the test data, we expect to have a more comprehensive yet smaller data set.

Meaningful exceptional data. Our previous efforts [14] have shown that exceptional inputs should receive more attention in testing to assure the robustness of the FinTech system under test. However, not all invalid data are worth testing. One of the interesting "experiences passed down by domain experts" in the FinTech industry is that meaningful exceptional data is likely to trigger failures. Meaningful exceptional data refers to inputs that the FinTech system under test is likely to accept and process, but lead to exceptional behaviors, such as an input with data fields satisfying its format. An example of meaningful exceptional data is an input with a data field as a string of code (representing a client institution) that satisfies the data format but belongs to a wrong institution. But this kind of errant data cannot be generated by simply generating values that do not match the corresponding input-format rules. Therefore, we design FinExpert to provide additional mechanisms for domain experts to specifically describe such errant scenarios, to generate meaningful exceptional inputs.

Easier test-result evaluation. Facts uses an enhanced differential testing to evaluate the test results. It requires to concurrently execute the same test inputs over different subsystem versions and compare their results. Despite useful, it typically elongates the testing process. We observe that there exist quite a few subsystems whose expected return codes are simple codes, e.g., ones denoting transaction completion or error codes.

5 OUR APPROACH

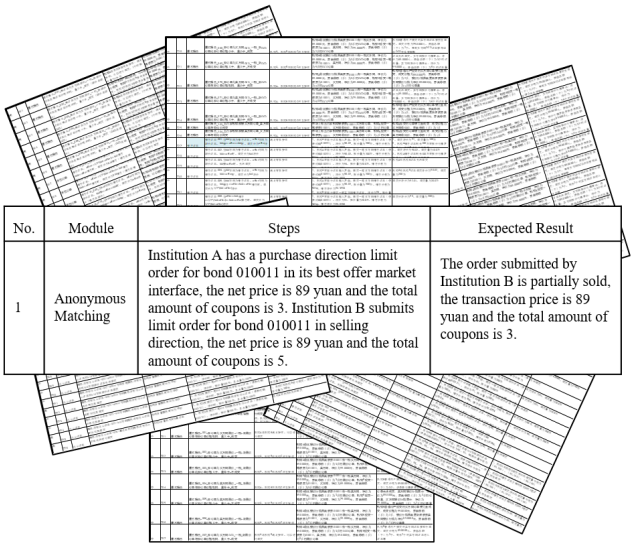
Motivated by our field observation and the case study shown in Section 3, we propose FinExpert to introduce domain knowledge into the process of generating test data automatically. Our goal is to generate test cases that are effective and efficient, in order to produce a comprehensive yet small collection of test cases. Being comprehensive yet small indicates that the generated test cases should cover as much code as possible, while the size of the resulting test cases should be as small as possible [12, 15].

The overall testing process using FinExpert can be approximately divided into three steps as shown in Figure 1:

(1) Domain experts document the domain knowledge, mainly the data formats and constraints of each input field and their dependencies via the Domain Editor.
(2) Domain parser collects the documented information and parses it into constraint rules of each input field.
(3) Test Generator accepts the constraint rules and generates input data, as well as expected outputs, accordingly.

5.1 Domain-Specific Language (DSL)

According to our observation, from the perspective of efficient testing, domain knowledge can be concretized as data constraints of the system inputs, such as inputs "DateDash", "DateTime" and "DateTimeMilliSecond", as well as their dependencies, as described



No.	Module	Steps	Expected Result
1	Anonymous Matching	Institution A has a purchase direction limit order for bond 010011 in its best offer market interface, the net price is 89 yuan and the total amount of coupons is 3. Institution B submits limit order for bond 010011 in selling direction, the net price is 89 yuan and the total amount of coupons is 5.	The order submitted by Institution B is partially sold, the transaction price is 89 yuan and the total amount of coupons is 3.

```
input is List
input:0 is Dictionary
input:0:userID format "bsid(\d){3}"
input:0:bsType in [0,1]
input:0:payWay in ["01","02","03"]
input:0:productID is Integer
input:0:ordPrz is Integer range (0,10000)
input:0:TransactTime is String daterange (2012-03-05 16:26:23,2019-2-1
12:26:23)
input:1 is Dictionary
input:1:userID is String format "bsid(\d){3}"
input:1:bsType is Integer in [ 1-<%input:0:bsType%> ]
input:1:payWay is String in ["01","02","03"]
input:1:productID is Integer in [ <%input:0:productID%> ]
input:1:ordPrz is Integer switch (input:1:bsType==1) { True:
range(0,<%input:0:ordPrz%>),False: range(<%input:0:ordPrz%>,10000)}
input:1:TransactTime is String in ["<%input:0:TransactTime%>"]
ret is Dictionary
ret:retCode is String in ["200"]

-ErrorStart-
ret:retCode is String in ["201"]
input:1:ordPrz is Integer switch (input:1:bsType==0) { True:
range(0,<%input:0:ordPrz%>), False: range(<%input:0:ordPrz%>,10000)}
-ErrorEnd-

-ErrorStart-
ret:retCode is String in ["202"]
input:0:ordPrz is Integer switch (input:0:bsType==0) { True:
range(10000,20000), False: range(0,<%input:1:ordPrz%>)}
input:1:ordPrz is Integer switch (input:1:bsType==0) { True:
range(10000,20000), False: range(0,<%input:0:ordPrz%>)}
-ErrorEnd-
```

Figure 2: Comparison between test scenarios with domain constraints

in Section 3. We believe that by formally defining these data constraints, such domain knowledge can be used to generate test data automatically and in turn improve the quality of the generated test cases.

Hence, we design a domain-specific language (DSL), as described in Section 5.2, to specify such domain knowledge. Domain experts determine and define the data constraints with our specific format, so that FinExpert is able to parse these constraints as rules and generate test cases accordingly. We also notice that subsystems from the same business domain tend to share certain common constraints, so an extra benefit of such approach is that formalized and documented rules can be easily reused or adapted for testing other (sub)systems. As a result, the domain experts do not need to restart from scratch for testing every subsystem from the same domain.

In summary, as shown in Figure 2, by using the DSL, the massive piles of manually-written test scenarios can be abstracted into one simple file that contains multiple lines of constraint rules, and at the same time, the resulting test suite is comprehensive yet small.

5.2 DSL Design

Based on the common patterns that we observe in our case study as well as consulting with our industrial partners in *CFETS*, we design a DSL specifically to represent the domain knowledge: the data types, constraints, and dependencies of each data field; such domain knowledge is used to guide the test generation. An example of using the DSL is shown in the right part of Figure 2. The DSL has the ability to express common domain requirements of FinTech properly and concisely.

5.2.1 Variable Definition. Rules defined by our DSL are a set of variable definitions and descriptions. Each data field is defined as a variable using the following form:

[variable name] **is** [type] [keyword] [description]

where the "variable name" is an unique identifier for each field, and the word followed by "is" declares the data type of the field, which can be chosen from "String", "Integer", and "Float" to represent a basic data type, or "Dictionary" and "List" to represent a composite data structure. ":" is a special character to split outer and inner data fields' name or index of a list. For example, in Figure 2, "input" is defined as a list, and "input:0 is Dictionary" describes the data type of the first element of list "input", which is a dictionary. Then, "input:0:userID format "bsid(\d){3}"" is used to describe the data field named "userID" in that dictionary. The purpose of "keyword" and "description" is explained in the following section.

5.2.2 Keyword and Description. There are five predefined keywords in our DSL, and below are explanations of each keyword:

in. "in" is always followed by a list, and the elements in the list are the range of the valid values for this variable. For example, "input:0:payWay in ["01","02","03"]" denotes that the range of variable "payWay" in the first dictionary of the list "input" is "01"', "02", and "03".

format. "format" is always followed by a regular expression, and can be used in combination with only a variable of type "String". The regular expression is used to specify the format of

Algorithm 1: Domain-specific Test Generation

```
Input: rule_file, the file that contains constraint rules of
       the input data.
Input: required_num, the required number of test data.
Output: data_set, a set of test data.
original_rules, all_errant_conditions = readRules(rule_file);
data_set = ∅;
generated_num = 0;
while generated_num < required_num do
    data_set.addData(generateData(original_rules));
    generated_num = generated_num + 1;
end
for errant_condition ∈ all_errant_conditions do
    generated_num = 0;
    while generated_num < required_num do
        data_set.addData(generateData(combineRules(original
          _rules, errant_condition)));
        generated_num = generated_num + 1;
    end
end
return data_set;
```

this variable. For example, "userID format "bsid(\d){3}"" indicates that the value of the variable "userID" should begin with "bsid" followed by three digits.

daterange. Dates are often used in FinTech systems, so we design keyword "daterange" to generate strings for representing dates within the specified range. "daterange" is always followed by a tuple of two dates, and the generated dates should be later than the first date and earlier than the second date. This keyword can be used in combination with only a variable of type "String".

range. "range" can be used in combination with only a variable of type "Integer" or "Float". "range" is always followed by a tuple of two values, and represents the range of randomly generated values for the corresponding variable. The two values in the tuple represent the minimum value and the maximum value of this variable, respectively.

switch. "switch" is always followed by an expression and a dictionary. The test generator calculates the value of the expression, and uses the resulting value as a key to find the corresponding value in the dictionary, and then assigns that value to the corresponding variable.

The most important ability of the DSL is that it can express the dependency between fields. Each variable definition can refer to other fields' values by including a string of another field's name wrapped by "<%" and "%>". These strings will be replaced by the generated values of the corresponding fields during test generation, and the value generation for fields that refer to other fields will be delayed until all the references are replaced by a concrete value.

5.2.3 Errant Condition. To test the system robustness, it is common to generate exceptional inputs, i.e., inputs that are not expected by system. We focus on "meaningful exceptional data",

Algorithm 2: Generate input data from constraint rule set

```
Input: rules, the specified constraint rule set.
Output: data, the generated input data that satisfies rules.
data = constructDataStructure(rules);
unassigned_variables = rules.getAllVaraibles();
assigned_variables = ∅;
while unassigned_variables ≠ ∅ do
    for variable ∈ unassigned_variables do
        rule = getRuleForVariable(rules);
        for reference ∈ rule.getReferences() do
            refVariable = reference.getReferencedVariable();
            if refVariable ∈ assigned_variables then
                replaceReference(rule, reference,
                  refVariable.getValue());
            end
        end
    end
    abort_flag = true;
    for variable ∈ unassigned_variables do
        rule = getRuleForVariable(rules);
        if rule.getReferences() == ∅ then
            variable = rule.getVariable();
            value = generateRandomValueForRule(rule);
            assignValue(variable, value);
            unassigned_variables.removeVariable(variable);
            assigned_variables.addVariable(variable);
            abort_flag = false;
        end
    end
    if abort_flag == true then
        abort;
    end
end
fillDataWithValue(data, assigned_variables);
return data;
```

which is exceptional data that is valid in data type but not expected by the system under test, such as a wrong institution code for a transaction party. Our previous work [14] reveals that such data is more likely to trigger failures, hence worth being generated. We design an additional mechanism to specifically support this need.

After defining valid data types and constraints of each field, the users can choose to define "Errant Conditions". Each of these errant conditions starts with "-ErrorStart-" and ends with "-ErrorEnd-", with one or more lines of rules in between to describe the exceptional input constraints. These rules use the same syntax as described in Section 5.2. With the same format, the users are able to construct an errant condition by simply modifying the previous constraints.

The test-generation engine processes and generates valid data and exceptional data separately. When processing an errant condition, the engine first reads all rules for valid data in the same file, and then replaces the corresponding variable value with the

Algorithm 3: Combine the original constraint rules and exceptional data specification

```
Input: original_rules, the original rule set.
Input: errant_condition
Output: result_rules, the combination of original_rules and
        errant_condition.
result_rules = original_rules;
for rule ∈ errant_condition do
    variable = targetVariableOfRule(rule);
    original_rule = getRuleForVariable(result_rules,
     variable);
    if original_rule ≠ null then
        result_rules.removeRule(original_rule);
    end
    result_rules.addRule(rule);
end
return result_rules;
```

errant condition. For instance, as shown in Figure 2, both the last line in the first part of the specification (ret:retCode is string in ["200"]), and the first line in the errant condition (ret:retCode is string in ["201"]) define constraints for variable "ret:retCode"; thus, the latter rule is used to generate corresponding exceptional data. As such, both valid data "ret:retCode="200"" and exceptional data "ret:retCode="201"" are generated.

5.3 Data Generation

Once the domain knowledge, i.e., the input data constraints, is documented and parsed, FinExpert can generate test data accordingly. The data generation algorithm is shown in Algorithm 1. Definitions of functions *generateData* and *combineRules* are shown in Algorithms 2 and 3, respectively. To simplify the description, we assume to require the same number of valid input data and exceptional data. At the beginning of the data generation process, all variables are not assigned. In each iteration, the generation engine checks all variables that have not been assigned a value. For each of such variables, if there is no reference in the corresponding constraint rule, the engine produces a random input that satisfies the rule, and then assigns that value to the variable. At the end of the iteration, the generation engine checks the rest of the unassigned variables, each of which should contain at least one reference to another variable. If any of the referenced variables has been assigned a value, the reference is replaced by that value. The generation engine continues iterating until every variable has an assigned value, or aborts if none of the remaining variables can be assigned. The generation procedure is applied on the original rule set for valid data and each errant condition.

6 EVALUATION

Our goal is to generate a comprehensive yet small test suite by specifying domain knowledge and utilizing the specification to guide the test data generation. In our evaluation, we aim to answer the following two research questions:

- **RQ1. Effective Testing Process**: Compared to the current testing process, is our approach able to save substantial manual efforts and result with a more effective testing process?
- **RQ2. Efficient Test Suite**: Can our approach lead to a more comprehensive yet smaller test suite, compared to our previous approach [14]?

RQ1: Effective Testing Process. The *CFETS* trading platform includes more than 100 subsystems, each of which is a complicated system by itself. It could take anywhere from one month to even a year for just testing a subsystem itself, almost impractical to collect and measure the testing process of the whole trading platform. Hence, we evaluate FinExpert with the Trading Matching (TM) subsystem, one of the latest-tested subsystems. The TM subsystem is a core component of the trading platform and is responsible for matching suitable orders between sellers and buyers to seal a transaction. Specifically, the testing group along with its corresponding domain experts spent about one and half months in testing the subsystem, with detailed breakdown shown in Table 3. For a fair comparison, we employ a different group of testers and domain experts but with a similar background as the previous group. More specifically, the two groups have similar backgrounds in the FinTech industry and same years of experiences in testing. We find that **69.5% of the manual efforts of current practice can be saved** by using FinExpert, especially the phase of Test-Data Generation that heavily relies on manual efforts, changing from 256 man-hours down to 0 man-hour. Additionally, since the domain knowledge is documented as specification, some of the data constraints can be referenced and reused, saving two-third of the man power in the phase of Test Suite Management where testers need to update the test suite according to test results. It is also worth to mention that the current practice requires 40 domain experts' man-hours and 32 testers' man-hours for designing the test scenarios, and FinExpert is able to cut down to 24 domain experts' man-hours with help from only 8 testers' man-hours. Instead of concerning detailed data values and exploring their correlations, domain experts now need to concentrate on only the data constraints. Furthermore, thanks to the structured domain-specific language provided by FinExpert, the size of the resulting specification is much smaller than the original test scenario set defined in natural language. For the phase of Test Execution and Evaluation, about half of the man-power are saved because some of the expected return codes are able to be documented in the specification, and the remaining half of the man-power are mainly for evaluating the test outputs.

RQ2: Efficient Test Suite. We also investigate whether the resulting test suite is a comprehensive set with not too many redundancies. We first apply FinExpert on our motivating example, the ISVPT subsystem. The comparison results are shown in Table 5. Recall that three different approaches generate three data sets as described in Section 3. FinExpert is able to outperform the three data sets with a relatively small number of cases. Unfortunately, for security reasons, we have no access to the source code of the ISVPT subsystem, and thus we cannot collect information regarding branch coverage and cannot conduct a more in-depth analysis. But it is clear that FinExpert is able to **test the subsystem more comprehensively**, with a **much smaller test suite**.

Table 3: Time spent in different approaches

Time spent (man hour)	Current Practice			FinExpert		
	Domain Expert	Tester	Total	Domain Expert	Tester	Total
Requirements Analysis	40 (Test Scenarios Design)	32	72	24 (Domain Specification)	8	32
Test Data Generation	0	256	256	0	0	0
Test Execution & Evaluation	32	216	248	16	104	120
Test Results & Aggregation	0	32	32	0	32	32
Test Suite Management	0	48	48	0	16	16
Total	72	584	**656**	40	160	**200**

Table 4: Code and branch coverage achieved by FACTS and FinExpert on CSTP

	Code coverage		Branch coverage		Number of cases	
	FACTS	FinExpert	FACTS	FinExpert	FACTS	FinExpert
FxclDealLogParser	95.9%	99.7%	91.5%	97.0%	2322	100
FxDealLogParser	89.1%	99.1%	85.3%	97.4%	10855	1000
FirdvDealLogParser	91.9%	98.9%	82.0%	98.0%	4500	200

Table 5: Code coverage achieved by different data sets and FinExpert

	Set 1	Set 2	Set 3	FinExpert
Code coverage	44.8%	24.3%	32.6%	56.6%
Number of cases	4	1000	100	100

We then extend our previous evaluation of the CSTP subsystem [14] with applying both approaches. We have already shown that automated test generation can generate a more comprehensive test suite [14]. Here we compare FinExpert with our previous efforts to understand whether the resulting test suite is comprehensive yet small.

The results are shown in Table 4. We choose three important classes in CSTP used to handle three types of input data, respectively. We observe that for each class, FinExpert is able to **achieve higher branch coverage with a much smaller test suite**.

We further manually analyze the not-covered portion of both the CSTP and ISVPT subsystems, and summarize the findings as follows.

Unreachable code. We have found several conditional expressions that will never be true; as a result, the corresponding code blocks will never be executed. After we report these cases back to the developers, it is confirmed that these conditions are either mistakes made due to complex logic expressions or directly copy-pasted conditions that will never be satisfied in the current context.

Unused functions. There also exist functions and methods that are defined but never used. These cases include deprecated functions that are no longer used but are retained for backward-compatibility, as well as template code generated by IDE but not actually used, such as getters and setters of a Java class. There are also residual functions and methods used for temporarily testing the module, such as "main" functions in different Java classes.

Incomplete constraints. The comprehensiveness of the test suite depends on the completeness of the constraint rules for the input data. The coverage of the ISVPT subsystem is relatively low because the requirements document is insufficient to start with, and it is difficult to write a complete set of constraint rules without sufficient requirements.

7 RELATED WORK

Although it is crucial to comprehensively test FinTech systems to ensure their robustness and correctness, to the best of our knowledge, FACTS is so far the only tool for automated testing of a FinTech system. Other state-of-the-art test generation tools, such as Randoop [9], EvoSuite [4], KLEE [2], EXE [3], Pex [16], and Ballista [5] fail to generate valid input values for the system under test in our evaluation.

In this paper, we have proposed a specification-based test-generation approach. Specification-based testing is the technology to generate test cases based on specification documents.

Richardson et al. [11] proposed approaches to specification-based testing by extending a wide variety of implementation-based testing techniques to be applicable to formal specification languages. They presented an algorithm for automatically deriving efficient test oracles from Graphical Interval Logic (GIL), which is a graphical temporal logic easier for non-experts to understand than many formal languages [8].

Rutherford et al. [13] developed a test-generation tool for model-driven systems. Antonio et al. [1] developed a prototype test generator that is able to provide a test collection automatically with formal semantics. The generator works by parsing Object Constraint Language (OCL) specifications to extract constraints and then generating test cases that satisfy the constraints.

Nilsson et al. [7] proposed a model-based approach for generating test cases to test timeliness by a genetic algorithm. This approach is suitable for generating test cases for small real-time systems. Rayadurgam et al. [10] proposed an approach for generating test cases to satisfy structural coverage criteria. They abstract the software as a finite state model, and use a model checker to generate test cases.

However, none of the preceding related approaches can tackle the problems faced in testing FinTech systems, such as high-dimensional input data and various data types and formats, and thus cannot be applied to test FinTech systems, the focus of our work.

8 CONCLUSION AND FUTURE WORK

To address the problem of lacking domain knowledge faced in the current testing practice, we have proposed a new approach named FinExpert that enables to specify and utilize domain knowledge to guide automatic test generation. FinExpert improves the effectiveness and efficiency of our previous approach named Facts [14] in industrial settings. We have applied our approach on three important subsystems of the *CFETS* trading platform, and the results show that FinExpert is able to achieve higher code coverage than Facts with a much smaller number of test cases, and much fewer manual efforts compared with the current testing practice in *CFETS*.

We also identify three main directions for future work.

Reusable constraint rules. Subsystems from the same domain may share common business logic. Thus, we expect that some of the constraint rules that represent underlying financial logic may be able to be reused at other subsystems from the same domain. We plan to investigate further how to efficiently reuse these previously-written constraint rules, as well as mechanisms for domain experts or testers to easily reuse these rules.

Manually defined specification. Manual efforts are error prone. Rules defined by domain experts may contain errors, e.g., circular references with which no data can be generated. We plan to investigate mechanisms to check the conflicts between rules to prevent such errors. Furthermore, it is commonly understood that domain specification itself needs to be verified as well. We plan to investigate ways to verify the specification against the system requirements.

Complex oracles. Subsystems may expect different outputs. FinExpert allows domain experts to define simple expected outputs, such as pre-defined return codes, or can be easily derived from inputs. For subsystems whose expected outputs are beyond this simple description, we plan to investigate various other ways to define complex oracles.

ACKNOWLEDGMENTS

This work was supported in part by the National Natural Science Foundation of China (No. 61571191 and No. 61632012), the 'Shuguang Program' supported by Shanghai Education Development Foundation and Shanghai Municipal Education Commission (No.16SG21), the Science and Technology Commission of Shanghai Municipality Grant (No.18511103802 and No.18511106202) and the key teaching reform project for undergraduates in Shanghai Universities (Project name: Innovative Education in the Era of Artificial Intelligence and Big Data), and NSF under grants no. CNS-1513939, CNS-1564274, CCF-1816615, and a grant from Futurewei.

REFERENCES

[1] Percy Antonio, Pari Salas, and Bernhard K. Aichernig. 2006. Automatic test case generation for OCL: A mutation approach. In *Proceeding of International Conference on Quality Software*. 64–71.

[2] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of USENIX Conference on Operating Systems Design and Implementation*. 209–224.

[3] Cristian Cadar, Vijay Ganesh, Peter Pawlowski, David Dill, and Dawson Engler. 2006. EXE: A system for automatically generating inputs of death using symbolic execution. In *Proceedings of ACM Conference on Computer and Communications Security*. 322–335.

[4] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic test suite generation for object-oriented software. In *Proceedings of ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 416–419.

[5] Philip Koopman. 1998. Toward a scalable method for quantifying aspects of fault tolerance, software assurance, and computer security. In *Proceedings of the Conference on Computer Security, Dependability, and Assurance: From Needs to Solutions*. 103–131.

[6] Kent Miller. 2019. *Global fintech investment rockets to a record $111.8B in 2018, driven by mega deals: KPMG Pulse of Fintech.* Retrieved April 2, 2019 from https://home.kpmg/xx/en/home/media/press-releases/2019/02/global-fintech-investment-hits-record-in-2018.html

[7] Robert Nilsson, Jeff Offutt, and Jonas Mellin. 2006. Test case generation for mutation-based testing of timeliness. *Electronic Notes in Theoretical Computer Science* 164, 4 (2006), 97–114.

[8] T. Owen O'Malley, Debra J. Richardson, and Laura K. Dillon. 1996. Efficient specification-based oracles for critical systems. In *Proceedings of California Software Symposium*. 50–59.

[9] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-directed random test generation. In *Proceedings of International Conference on Software Engineering*. 75–84.

[10] Sanjai Rayadurgam and Mats Heimdahl. 2001. Coverage based test-case generation using model checkers. In *Proceedings of Annual IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*. 83–91.

[11] Debra Richardson, Owen O'Malley, and Cindy Tittle. 1989. Approaches to specification-based testing. In *Proceedings of ACM SIGSOFT Symposium on Software Testing, Analysis, and Verification*. 86–96.

[12] Gregg Rothermel and Mary Jean Harrold. 1993. A safe, efficient algorithm for regression test selection. In *Proceedings of International Conference on Software Maintenance*. 358–367.

[13] Matthew J Rutherford and Alexander L Wolf. 2003. A case for test-code generation in model-driven systems. In *Proceedings of International Conference on Generative Programming and Component Engineering*. 377–396.

[14] Qingshun Wang, Lintao Gu, Minhui Xue, Lihua Xu, Wenyu Niu, Liang Dou, Liang He, and Tao Xie. 2018. FACTS: Automated black-box testing of FinTech systems. In *Proceedings of ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 839–844.

[15] W Eric Wong, Joseph R Horgan, Saul London, and Aditya P Mathur. 1998. Effect of test set minimization on fault detection effectiveness. *Software: Practice and Experience* 28, 4 (1998), 347–369.

[16] Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. 2009. Fitness-guided path exploration in dynamic symbolic execution. In *Proceedings of IEEE/IFIP International Conference on Dependable Systems & Networks*. 359–368.