Towards Model-Based Testing of Electronic Funds Transfer Systems

Hamid Reza Asaadi[1,2], Ramtin Khosravi[1,2],
MohammadReza Mousavi[3], Neda Noroozi[2,3]

[1] Department of ECE, University of Tehran, Tehran, Iran
[2] Sofware Quality Lab., Fanap Co., Tehran, Iran
[3] Department of CS, TU/Eindhoven, Eindhoven, The Netherlands

May 25, 2010

Abstract

We report on our first experience with applying model-based testing techniques to an operational Electronic Funds Transfer (EFT) switch. The goal is to test the conformance of the EFT switch to the standard flows described by the ISO 8583 standard. To this end, we first make a formalization of the transaction flows specified in the ISO 8583 standard in terms of a Labeled Transition System (LTS). This formalization paves the way for model-based testing based on the formal notion of Input-Output Conformance (IOCO) testing. We adopt and augment IOCO testing for our particular application domain. We develop a prototype implementation and apply our proposed techniques in practice. We discuss the encouraging obtained results and the observed shortcomings of the present approach. We outline a roadmap to remedy the shortcomings and enhance the test results.

1 Introduction

Electronic Funds Transfer (EFT) systems provide the infrastructure for online financial transactions such as money transfer between bank accounts, electronic payments, balance enquiries, and bill payments. A central part of an EFT system is the *EFT Switch* (also known as *Payment Switch*, or simply *Switch*), which provides a communication mechanism among different components of an EFT system such as Automated Teller Machine (ATM) and Point-of-Sale (POS) terminals, e-Payment applications, and core banking systems.

The EFT system components communicate in the form of transactions consisting of several messages passed through the switch. For example, during a simple *purchase transaction* originated by a POS terminal, the switch forwards the purchase request to the core banking system (to charge the card holder's account) and forwards the response back to the POS terminal. In the real setting however, possible failures in the components and asynchrony in the communication media may give rise to more complicated transaction flows. For example, if a POS terminal sends a purchase request and it does not receive the response from the switch in time, it will time-out and send a reversal message to the switch, requesting to cancel the previous transaction. It is also possible that when the time-out occurs, the purchase response is on the way back to the POS terminal. In this case, the POS terminal receives a purchase response after it sends a reversal request (which of course must be responded too, by the switch). This way, each transaction may comprise a complex combination of different possible interaction scenarios among the components of the EFT system.

In the presence of such complicated transaction flows, a thorough testing of EFT switches is essential, as presence of errors may lead to inconsistencies among different accounts (particularly among accounts at different banks). This calls for a reconciliation process, possibly requiring manual checks which are very costly for the banks.

The correct behavior of a typical EFT system is specified in the ISO 8583 standard [2] at a high level of abstraction. Since the nature of the system is concurrent and distributed, generating test cases manually with a high coverage is practically impossible, as the number of (combinations of) transaction flows is very large. To solve this, we use *model-based testing* [4, 11] as a systematic method to automatically generate test cases from the specification.

Our testing method is mainly based on a formalization of the ISO 8583 standard in terms of Labeled Transition Systems (LTSs). Our formal specification captures the behavior of an ISO-compliant EFT switch as well as its environment, i.e., the terminals and the core banking system. We have also performed model-checking on our formal model to make sure that our formalization of the ISO 8583 standard meets the intuitive requirements set forth by the standard as well as by the switch designers. This formalization paves the way to exploit a formal conformance testing method called IOCO (for Input Output Conformance) testing [17, 18] to automatically generate test-cases and perform online conformance testing. We combine IOCO testing with functional testing techniques, à la category-partition method, to capture the data-related aspects of switch functionality. Moreover, we interface the test-case generator, with our own test-case analysis and execution tool to evaluate, store, and prioritize test cases; the test-cases are executed and their outcomes are also stored by the same tool. Our test selection technique combines the black-box nature of IOCO (focusing on model-coverage criteria) with white-box coverage metrics in order to choose an effective test-suite. We developed a prototype tool implementing the above mentioned functionality. Using our tool, we can generate a prioritized test-suite for off-line and regression testing, without any need to explore the formal model any more. Furthermore, during the execution of test cases, our tool also validates various business rules which could not be captured in the formal model.

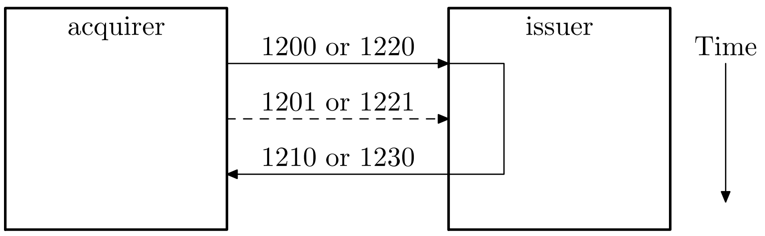
We applied our prototype implementation to an operational switch, developed by Fanap Co., interacting with POS terminals and a core banking system as its environment. We have covered a number of major transaction types and related business rules and have detected some defects in the switch, which are reported in the remainder of this paper. The defects have been reported to development team and have been fixed subsequently. The initial results obtained from our prototype, presented in this paper, were very encouraging. Hence, Fanap decided to embark on the development of a proprietary test-case generation tool which automatically combines the behavioral and functional models outlined in this paper.

The rest of this paper is organized as follows. Section 2 provides a background on the switch specification as described in the ISO 8583 standard. Section 3 covers our testing approach in addition to a quick overview of the IOCO theory. The way we model the system in terms of Input Output Transition Systems is described in Section 4. Various aspects of our testing method including test case execution, and generation and prioritization of off-line test suites, as well as checking business rules are presented in Section 5. The test results and code coverage are given in Section 6. Discussion of the merits and demerits of the current approach are discussed in Section 7. Section 8 presents a brief overview of related work. Finally, we conclude the paper and present some directions for future work in Section 9.

2 EFT Switch Functionality

Typical functionality of an EFT Switch include performing a purchase, balance enquiry, withdrawal, bill payment, refund, and money transfer. All these functions are composed of a few transaction flows introduced below. Apart from financial functions, there are also features for switch administration, monitoring and auditing that are out of the scope of this study.

As the components of an EFT system are usually provided by different vendors, the ISO 8583 standard [2] is defined to determine the type and format of the messages exchanged among the components of an EFT system. The standard also defines message and transaction flows at a high level of abstraction. For example, Fig. 1 shows the flow of a financial transaction as depicted in the standard [2]. According to the standard, the acquirer is defined as "the financial institution (or its agent) which acquires from the card acceptor the data relating to the transaction and



1200/1201	financial request/financial request repeat
1210	financial request response
1220/1221	financial advice/financial advice repeat
1230	financial advice response

Figure 1: Message flow for financial transactions [2]

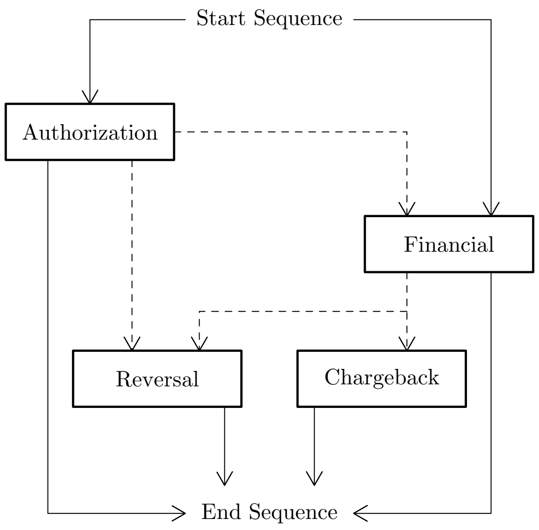


Figure 2: Allowed sequence of business transactions – solid arrows indicate the typical order, while dashed arrows indicate the possible (exceptional) order of transactions [2, Section 5.3.1].

initiates that data into an interchange system.” The card issuer is “the financial institution (or its agent) which issues the financial transaction card to the card holder.” According to the described flow, the acquirer sends a request to the card issuer, followed by zero or more request repeat messages, until it receives a response from the issuer. The data format of the messages (e.g., 1200 - financial request) has been defined elsewhere in the standard [2, Chapter 4]. Note that each typical functionality of an EFT switch, e.g., a purchase or a balance enquiry, is composed of a number of transaction flows, such as the one depicted in Fig. 1. Apart from the flow depicted in Fig. 1, there are eleven more transaction flows specified in the standard. We refer to [2, Chapter 5] for a detailed presentation of all transaction flows.

In addition to a generic flow for each transaction type, the standard also specifies two other flows. One that specifies possible sequences of transactions relating to a single instance of business at a point of service, and another that defines the reconciliation process between the acquirer and the card issuer. Fig. 2 shows the former, which can be seen as a high level sequence diagram referencing the individual transaction flows described before. Each box in the figure is a transaction with its flow specified in the standard, e.g., the financial transaction described before.

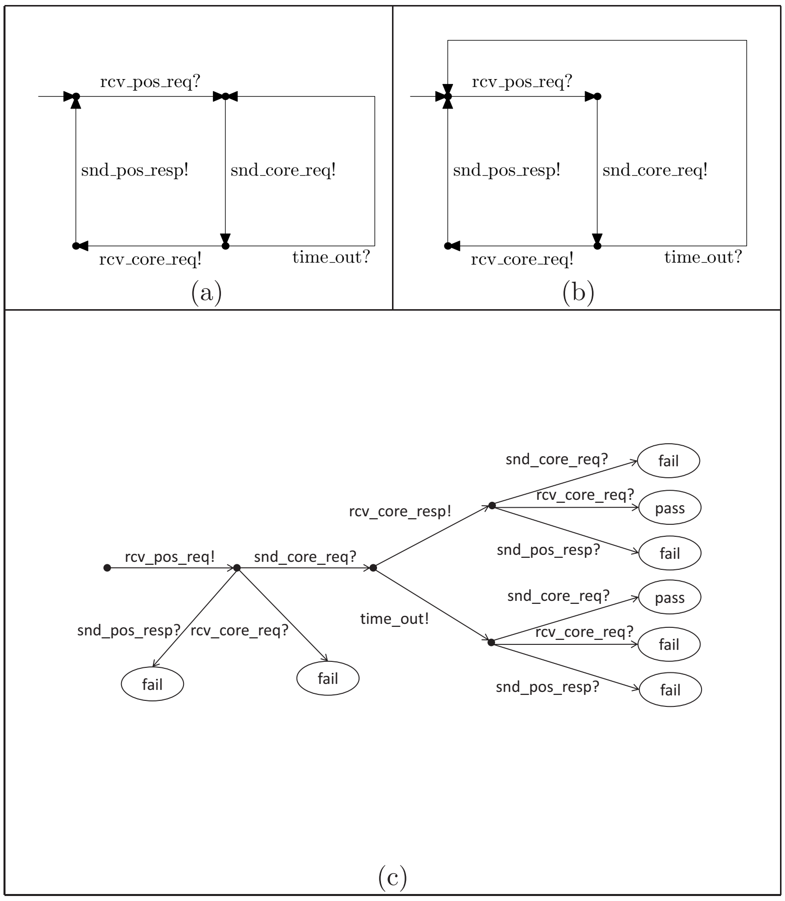


Figure 3: IOCO testing of (b) an implementation against (a) a specification results in (c) a tree presenting test cases based on transaction flows.

3 Testing Approach

3.1 IOCO Testing

IOCO testing [17, 18] is a formal approach to *model-based black-box* testing of *functional* requirements. The approach relies on a formal model of system behavior, a specification typically called s, which captures the observable input and output interactions of the system with its environment in terms of a Labeled Transition System (LTS). Based on the specification, IOCO testing generates test-cases in order to establish whether the implementation under test, typically denoted by i, *conforms* to its specification, written as i conf s. The basic concepts of IOCO testing is illustrated next using the specification LTS depicted in Fig. 3. In an LTS specification of (an extremely simplistic view of) a transaction's life-cycle in an ideal switch is given. The IOCO testing is aimed at checking whether a particular implementation, e.g., the one depicted in Fig. 3.(b), conforms to its specification depicted in Fig. 3.(a). (Note that the LTS of the implementation is not available to the tester, and the LTS is only used here to illustrate possible patterns of interaction with the system.) To this end, the IOCO testing technique uses the specification LTS and generates test-cases, e.g., those depicted in Fig. 3.(c), to test whether the (black-box) implementation conforms to the specification. In this figure, input and output actions are affixed with a question and an exclamation mark, respectively. In Fig. 3.(a), each path of the depicted tree presents a pattern of interaction (i.e., providing input- and observing output messages) eventually leading to a pass or a fail verdict. In particular, executing the test-case corresponding to the path rcv_pos_req! .

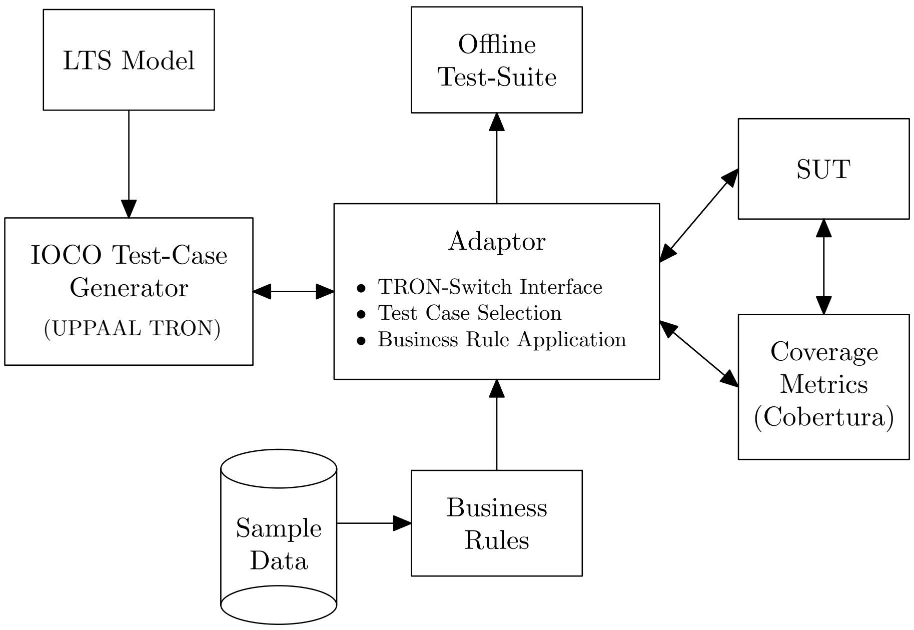


Figure 4: An overview of the test infrastructure

snd_core_req? . time_out! . rcv_pos_req! . snr_cor_req? reveals a bug in the implementation. (Note that inputs in the model become outputs of the test-case and outputs of the model become inputs of the test-case.)

3.2 The Testing Infrastructure

An overview of our test infrastructure is given in Fig. 4. We implemented this infrastructure in three phases, explained below.

In the first phase, we made an LTS model of the EFT switch system and a POS terminal as its environment. Details of this explanation are described in Section 4. Then, we run the generated test cases on a switch system connected to an operational core banking system. For this experiment, we used the timed-automata language of Uppaal [3] as our modeling language and Uppaal Tron [10] as our test-case generation tool. (Uppaal Tron implements a variant of IOCO, called RTIOCO; see [16] for a formal comparison of the notions.) For the commercial use, we plan to implement the test-case generation algorithm in our in-house tool and integrate it with our test infrastructure described below. In addition to that, we developed an adaptor to translate and augment abstract interactions of the model to concrete network messages sent to the switch, on one side, and strip down network messages from EFT to model interactions, on the other side. After running each test-campaign and receiving a verdict from the IOCO testing, we made a manual inspection of the log files. In the initial stages of testing, this resulted in one of the following conclusions:

1. a false negative: the IOCO testing has reported a failure while the system behavior is as expected; in our experience, such false negatives mostly stem from an incomplete treatment of asynchronous messages in our specification. By making the specification complete, we gradually got fewer and fewer false negatives.

2. a false positive: the IOCO testing has passed the success verdict while the actual behavior of the system is not as expected (e.g., the current state of the core banking system does not show the expected result of the transaction). This has been mainly due to failures in other

components of the EFT system, which fall outside our test-plan. We partially solved this problem in our second phase of modeling (see below).

3. a true negative: the IOCO testing has found a bug. For example, we have found a bug which was known by the development team but was deemed non-reproducible before. Again, the main cause of difficult reproduction of such bugs is the asynchronous nature of the system which allows for a huge space of interleaving among different phases of concurrent transactions. However, with a precise specification of the interleaving at hand, we could easily reproduce the bug and in debugging, it was traced back to a null pointer dereferencing.

In the second phase, we built a model of ideal POS and core banking systems and further implemented the model of the ideal banking system as a separate parallel component. This step is motivated by two facts: firstly, the core banking system is a separate component and usually operates in a different environment. Core may have its own bugs that should not affect switch testing process. Hence, using the models for our environment components, we could test a stand alone EFT switch. This helped us focus the process on our test plan, eliminate false positives and isolate the bugs in the debugging procedure. Secondly, we faced severe performance problems when we tried to generate massively parallel executions of transaction flows. After trying several options, the main cause is traced back to the huge state-space of the model resulting from the combination of the core banking system model on one hand and the EFT switch and POS models on the other hand. Hence, we separated the model of the ideal core banking system and implemented it as a separate component, running in parallel and acting as the environment for the test-case generator (more details to be found in Section 4).

Finally, in the third phase, we built our tools for storing test-cases and their outcomes, prioritizing them and executing off-line test-suites and placed it around the test infrastructure. For the test prioritization and selection, we implemented our heuristics and combined them with the code coverage metrics from Cobertura [1]. This allows us to re-use the information resulting from an online test campaign in future tests and also use the generated test-suite for regression testing.

4 Modeling the EFT Switch

4.1 A Cook's Tour of Uppaal Timed Automata

Our LTS formalization of the ISO 8583 standard is specified in terms of the the input language of Uppaal in order to benefit from several modeling, simulation, verification and test-case generation tools available in its tool-set. A model in Uppaal is in the form of a network of timed automata. A timed automaton is a finite-state machine (FSM), i.e., a set of *locations* which are connected via *edges*, extended with (constraints on and assignments to) clock variables [3]. An edge in an Uppaal timed-automata can be annotated by four types of labels: *selections*, *guards*, *synchronizations* and *updates*.

When taking a transition specified by an edge, an automaton may send or receive a *signal* in the synchronization part. Synchronization in Uppaal can be either a handshaking or a broadcast synchronization. Common to our previous examples, a send signal in Uppaal is annotated by an exclamation mark and its receive counterpart is annotated by a question mark.

In order to specify concepts such as guards, parameterized synchronization and updates, one can define *variables* of given finite types in Uppaal. A variable may be *global* (visible to all automata), *local* (visible only locally) or *bound* (visible only during a transition and assigned a non-deterministic value from a specified range). Uppaal also provides variable and signal arrays. Signal arrays can be used in order to pass data with signals. One may further define user-defined functions to manipulate and calculate data values in a C-like syntax.

When a transition corresponding to an edge is taken, the update label will be executed and the corresponding variables are thus updated. It is possible to assign an arbitrary value from an integer interval or a scalar set to a bound variable. This is achieved by specifying a select label for the variable.

4.2 Organizing Models

The behavior of an EFT switch and its environment is specified in terms of a number of transaction flows. Combining all of these flows into a single model (a timed-automaton) would compromise readability and maintainability; it is also very difficult, if not impossible, to check whether the specified automaton is a correct formalization of the flow specified by the ISO standard. Hence, we break the specification into several timed automaton, each modeling the behavior of EFT system components in a specific transaction flow (see Fig. 5).

To show how we model a transaction flow, consider a simplified model of the switch in Reversal transaction (Fig. 5.(b)). The starting state is s_1, where we send a rev_ready signal indicating we are ready to start a Reversal. Then, we wait for receiving a signal from a POS terminal to start the transaction. The signal is accepted from the channel rev_req[j]. The parameter j, determined by the environment, indicates the value of the current transaction ID and is assigned to the variable curTrx for future use. Specified as a requirement, the response of a reversal request should be immediately sent back to the POS terminal. Then, the reversal request is sent to the core banking system for processing. So, from state s_3, we make a transition to s_4, sending reversal responses to POS (rev_pos_rs). It is possible that the response is lost on the way back to the POS. So, an unlabeled transition to s_4 is also possible. The choice between the two cases is non-deterministic. Making a transition from s_4 to s_5, the switch sends a reversal request to the core banking system (rev_core_rq). Note that the two last signals are sent with the same transaction ID (since the switch does not change the transaction ID). However, it is possible to receive a reversal response from the core banking system (rev_core_rs) with an arbitrary transaction ID. So a separate variable is used as the channel parameter (receiveID). In case the received transaction ID is different form the expected value (curTrx), we go back to s_5 and wait for the signal with expected ID. Otherwise, we complete the transaction by sending a rev_done signal. The FSM in Fig. 5 (c) shows the behavior of POS in the same transaction flow. The POS instance sends a reversal request via the channel rev_req to the switch instance and receives the reversal response via the channel rev_pos_rs.

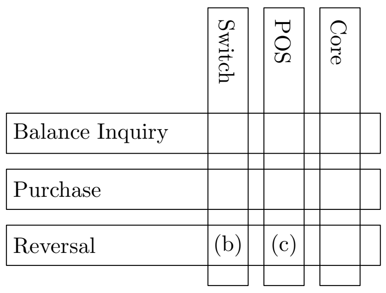
It is possible to have multiple instances of the same transaction flow executing concurrently. So, we need to have multiple instances of the corresponding FSMs in our model. This is possible in Uppaal, since we can declare multiple instances of the same FSM "template". In fact, the number of declared instances of an FSM determines the maximum number of concurrent instances of the corresponding transaction type. To generate various combination of transaction flows, we use a coordinator automaton. The coordinator non-deterministically selects the next flow to start and sends it the start signal and repeats this continuously as long as a parallel instance is ready to receive the start signal. For example, when the switch is ready to accept another Reversal request, it sends a rev_ready signal to the coordinator. Then, the coordinator sends a rev_start to the POS FSM to start the Reversal (Fig. 5).

During development of the model, human mistakes may introduce errors in the model. To discover such errors, we take a model-checking approach to verify the model against correctness properties before the testing process. We first formalized a few intuitive correctness based on the ISO standard and the intuition of the designer in the temporal-logic-based verification language provided by Uppaal Tron (for some properties, we had to augment the model with observer automata in order to compensate for the limited expressiveness of the logic). For example, the following formula is used to verify that every transaction started must eventually be finished.

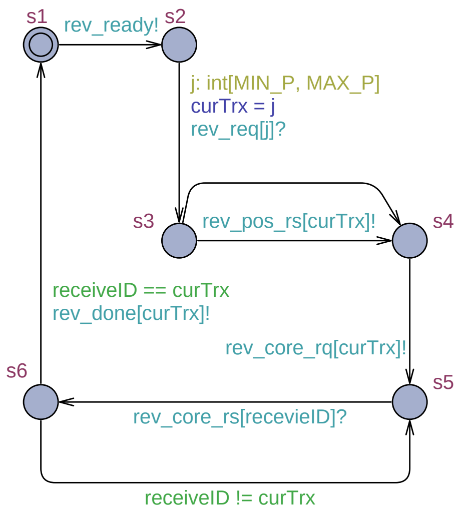
```
A[] forall (i : int[0, MAX_TX])
  TransFlow[i].start->TransFlow[i].finish
```

Subsequently, we use Uppaal verifier to model check the formalized correctness properties.

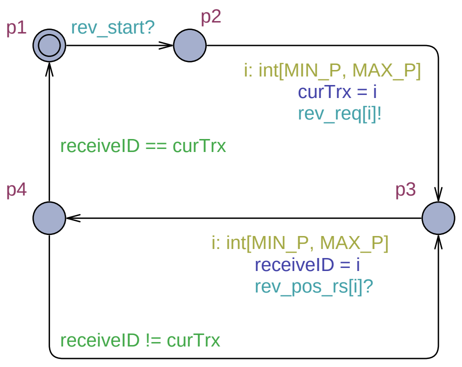
Due to the combinatorial explosion of the state space, the performance of the Uppaal Tron test-case generator was extremely low, when it tried to generate test-cases for the whole EFT system. To alleviate the state-space explosion problem, we implemented the abstract model of the core-banking system as a separate Java program and ran it in parallel with Uppaal Tron and its adaptor. With this simple improvement, we were able to increase the performance of the test-case



(a)



(b)



(c)

Figure 5: Each FSM in the model specifies the behavior of a component in a transaction flow (a). Simplified models of the behavior of Switch (b) and POS (c) in Reversal transaction flow.

generation by a factor of 10. This way, we could generate test-cases for hundreds of concurrent transaction flows for each instance of Uppaal Tron.

5 Testing the EFT Switch

5.1 Interfacing Switch and TRON

Uppaal Tron continuously interacts with the system under test while exploring the LTS model. In other words, on-the-fly test-case generation is combined with online testing, so that the next step in the test-case generation can be determined by the response from the system under test [13]. Hence, to interface Uppaal Tron with the system under test, an *adaptor* has to be implemented, which in its simplest form, communicates the messages between Uppaal Tron and the system under test (possibly after converting them to the right format for each side). We implemented such an adaptor which translates the rather plain signals of Uppaal Tron to (from) the elaborate format of financial messages specified by the ISO 8583 standard. In order to perform the translation to the ISO 8583 messages several details (concerning financial data of a transaction) have to be added to the message, which are selected from representative data stored in our sample database (more explanation about this to follow). Besides the format conversion and the addition/removal of financial data, we developed several other components in the adaptor which store and prioritize the test-cases in order to re-use them in regression testing. This way, a prioritized test suite is obtained, which can be run efficiently, without the overhead of exploring the formal model. Finally, there are some types of business rules that are hard to capture in Uppaal Tron models and hence, are applied and verified by a separate component in the adaptor. Next, we explain the functionality of our extended adaptor in some more details.

As mentioned before, Tron performs online testing by interacting with the switch during test-case generation and execution. SinceTron is a general purpose tool, there must be an adaptor to realize the communication protocol between Tron and the implementation under test. To this end, Tron provides a standard interface that should be implemented by the adaptor. The adaptor is used to translate Tron signals and data to the implementation host language (Java in our case).

The conversion of messages from Tron to switch requires augmenting them with several data-fields specifying the details of a certain transaction flow. The simplest example of such fields are the transaction type and the unique transaction identifier. The identifier ID has to be included as part of each message sent to or received from the switch, to differentiate between instances of the transaction type. Due to limited support of data types in Uppaal, the format of transaction identifiers is different in the model and the switch implementation. A function of the adaptor is to translate the ID values when passing messages. Moreover, there are many data items which are abstracted in the transaction flow specification (both in the standard and in our Uppaal model and thus, are to be provided by the adaptor.

Apart from the issue of generating data items, also some practical difficulties arise when connecting Tron to the adaptor. For example, Tron doesn't support arrays of boundary signals which is a useful feature provided by Uppaal. Solving this requires adding extra states to the model (to store parameters in local variables and synchronize them with the implementation under test). This increases the chance of human errors in modeling. It also makes the model rather large and increases the size of the state space, which in turn negatively influences the possibility of model checking them. To handle this, we first made a compact Uppaal model with parameterized synchronization (i.e., using signal arrays, which is used for model checking purposes) and once proven correct, we derive from that a second model (by replacing parameters with local variables and adding intermediate states and transitions for storing and synchronizing them), which is used during test execution by Tron.

Tron continuously generates the test-cases online using our Uppaal model. It communicates with our developed adaptor in order to send the output signals to the switch and receives the (abstracted) signal from the switch via the adaptor. Based on the received signal it gives a pass or a fail verdict. This process can generate several combinations of transaction flows with a considerable level of concurrency. However, to generate off-line test-suites, we need to specify extra parameters (such as the type and the length of each test-case) to break this continuous flow into discrete pieces and compare them. We have annotated the models in the later developments such that while interacting with TRON, the adaptor can partition the test-campaign into individual test-cases using the provided parameters. While executing each test-case, the adaptor checks gathers coverage metrics from the switch and uses this information together with the heuristics provided by a test engineer to select and prioritize test-cases. The prioritized test suite can later be used off-line, as well as for regression testing.

In addition to valid message flows, the ISO standard also specifies a number of business rules. For example, "the transaction ID of a reversal message shall be the same as the transaction ID of the original financial transaction", or "a reversal shall not be reversed". Some business rules are difficult (or sometimes impossible) to capture in UPPAAL models. To handle the cases in the example just mentioned, we keep in the adaptor, a list of transaction IDs sent from TRON and keep track of the last operation performed on each ID. This way, if we receive a reversal message on a transaction ID that is already reversed, we consider this case as an error.

5.2 Classifying and Covering Data Domains

Common to many reactive systems in the financial domain, the EFT switch exhibits complex reactive behavior while also having a data-dependent nature. An effective test method must address and integrate both of these facets. Theoretically, the test cases generated by IOCO cover all behavioral scenarios of interaction between the switch and its environment. However, since the model is based on labeled transition systems, it is hard to express data constraints on complex message structure and contents. So, we must set the fields of the messages generated by TRON to different combination of values. This results in multiple sequences of messages made from the single sequence of messages generated by TRON.

To manage the complexity of the data domain, we have used the *classification tree method* [9] (as an extension of the original category-partition method) to organize the test-case generation process. According to the method, we should select an aspect relevant to the test and partition the input domain into disjoint subsets called *classes*. The resulting classes will be subsequently classified according to some other aspect recursively, resulting in a tree of classifications and classes. This way, we specify representative elements for the content of date elements present in the structure of financial messages.

Moreover, to evaluate the quality of our test-case, we divide the ongoing pattern of interaction into discrete pieces; in the remainder of this paper, we define each of these pieces (with some re-use of terminology) as a *test-case*. Hence, a test-case is a combination of transaction flows (possibly of different types) with specified values for the data items in the messages passed during each transaction. For example, a test case may comprise a purchase transaction succeeded by a reversal. To specify discrete test-cases, in addition to the content of the financial messages, we should also specify the type of transaction flow and the size of transactions (repetition of messages and flows). We re-use the same concept of classification-tree to classify and specify representative values for these aspects, as well.

In our prototype implementation, we used the domain and the implementation knowledge of the EFT switch to classify the following set of data domains:

- Transaction flow types,
- PIN validity,
- Transaction amount, and
- Test-case size.

For each aspect, we select a suitable set of discriminating values by using the domain knowledge. For *Transaction type* we consider five different classes: Purchase only (P), Balance Inquiry only (B),

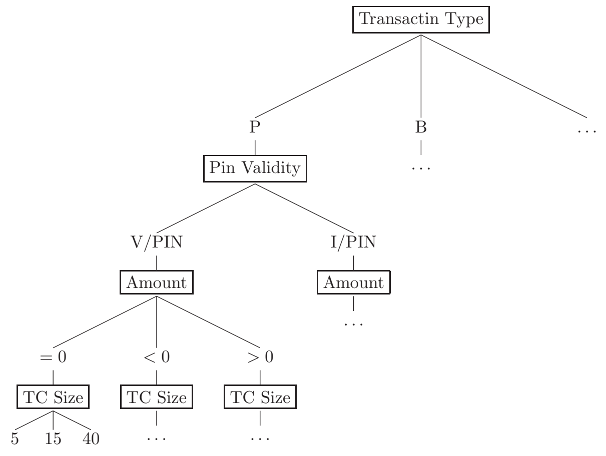


Figure 6: A part of test-case classification tree

Purchase and Balance Inquiry (PB), Purchase with Reversal (PR), and Purchase with Reversal and Balance Inquiry (PRB). The *PIN validity* classification shows whether the transaction is authorized to be executed according to the PIN number input parameter. The domain of *Transaction amount* is the set of positive integers. The negative and zero cases are also included to test invalid cases. Finally, the *Test-Case size* parameter is the number of transactions in the test case that determines the size of each transaction.A part of the resulting classification tree is shown in Fig. 6. Note that the classification tree is not supposed to be a balanced tree and hence, some parameters may not apply to all cases. For example, a Balance Inquiry transaction does not have a transaction amount as an input parameter.

The transaction type parameter is applied to the coordinator FSM (see Sec. 4) which affects the sequence of transactions generated by Tron. The other parameter values are set by the adaptor. The data selection tree is currently hard-coded in the adaptor, but we plan to make this generic and include it in the test specification model (see Section 7).

6 Test Results

Apart from online testing, which has been very helpful in revealing defects in the product, defining suitable test-cases enabled us to measure test coverage for each test-case and prioritize the test-cases according to our test plan (in this case: full statement coverage of functional components, i.e., components involved in the realization of functionality in the main transaction flows). This prioritized set is used for off-line and regression testing, particularly when running the whole test-infrastructure is not feasible and the testers have to choose some of test-cases to get the most coverage. In this work, our test plan is to cover different flows as much as we can, instead of trying to cover all features of the switch.

We have selected the test cases using based on our category-partitioning analysis. Due to some obstacles in the implementation, in this work we have just used positive values for the *Amount* parameter. Though, other parameters are tested as described in the resulted classification tree (Fig. 6). We have measured the statement coverage using Cobertura [1]. To reduce measurement errors, each test case has been repeated four times (with the same configuration) and the average coverage is reported in percents in Table 1.

Early analysis showed that there is a considerable amount of common code between the purchase and balance enquiry implementation because of inherent common logic. This hypothesis can also be proven by measuring the relative coverage of adding a test-case of former type to a test-case of the latter type (or vice versa); namely, the addition of each type of test-case to the other does not significantly increase the statement coverage measure. Hence, combining these two

	Trx No.		
	5	15	40
B	0.614	0.622	0.620
B(Invalid PIN)	0.487	0.487	0.487
P	0.589	0.589	0.594
P(Invalid PIN)	0.487	0.487	0.487
PB	0.596	0.596	0.592
PB(Invalid PIN)	0.487	0.487	0.487
PR	0.630	0.671	0.671
PRB(Invalid PIN)	0.529	0.529	0.529
PRB	0.712	0.710	0.712
PRB(Invalid PIN)	0.529	0.529	0.529

Table 1: Statement coverage results in percent

tests (i.e., the PB row in Table 1) did not result in any considerable improvement in coverage.

Further analysis shows that a significant amount of code for processing a transaction is devoted to common tasks such as authorization and packet routing. This justifies why there is not much difference between the coverage results of the cases.

Note that the increase in the size of transactions beyond 15 messages did not increase the coverage considerably, since apparently this does not lead to any new behavior in the EFT switch and the same logic is executed repeatedly. However, in our experience, having a large number of parallel transaction instances does increase the chance of catching errors caused by concurrency issues or (thread-pool) overflow problems.

Another point is that test-cases with exceptions have lower coverage among other combinations, yet they are deemed very important by domain experts. This is true because the switch drops unauthorized messages in early stages, so a big part of the code will never run. This is justified by developers' insight that the code for handling exceptional cases has little overlap with the code for the normal transaction flows. Hence, despite their individual low coverage, these test-cases should be appeared with high priority in the final priority list.

7 Discussion

Our system under test is inherently a mixture of reactive and functional behavior: it implements a high-level protocol for exchanging messages for a financial transaction, while its detailed implementation is very much dependent on the functional and data-related aspects. This mixture, if not structured properly, makes the generated models overly cluttered and complicated and unfortunately, most of the existing IOCO-based tools, including Uppaal Tron and TorX [19] do not provide proper facilities for orthogonalizing, structuring and relating reactive and functional behavior. Hence, we plan to make a high-level specification language (inspired by prior effort in UML Testing Profile [14], TTCN3 [22]) as a front-end for our proprietary IOCO-based engine in order to solve the following issues:

1. Specification of abstract data types and their partitions: a specification language is needed to specify the data types used in the functional domain, different partitioning and the representative elements of partitions.

2. Full support of data parameters in the behavioral model: the support for data parameter in Uppaal Tron is limited; it is not possible to define the representative data values of each data type attached to messages of the behavioral model. Being able to attach different data types and their different partitionings is an essential ingredient for improving our test results.

3. Support for asynchronous message passing: Thus far, we have experimented with different additions to our model in order to cater for the asynchronous nature of communication in our domain. We first tried adding input/output queues was one option which immediately led to drastic performance drawbacks. Then, we have experimented with abstracting from the asynchronous delays in our protocols, which does lead to better performance. However, such an abstraction results in fictitious sequences of messages that are not expected by the SUT. To overcome this, we had to add several guards to guarantee that the model will only be triggered with appropriate signals. This last modification has led to a complicated specification. An inherent support for asynchronous message passing may be considered as an option, along the lines of the initial proposal in [21].

4. Specifying a more dynamic notion of test goal and model coverage: Uppaal Tron does not allow for specifying a notion of test goal. Apart from traditional notions of test goal, e.g., hitting certain states in the model, we need to specify test goals that refer to the coverage of the functional model. For example, it is essential to cover all (combinations of) representative elements of a certain partitioning of data types, a la the equivalence-class testing method.

Despite the above-mentioned shortcomings, Uppaal Tron can still be considered for prototyping a test-bed for similar systems, however, our experience shows that the following issues need to be considered:

1. Performance issues: Due to the very complex and mixed nature of the system, we soon reached the boundaries of possibilities with Uppaal Tron. To overcome this problem we had to distribute our test-case generation among a number of parallel instances of Uppaal Tron. A challenge imposed by this solution is how to pass the received messages to the right instance of Uppaal Tron. This problem is intensified by the lack of appropriate support of data-type-handling. To solve the latter problem, we annotated the messages in the underlying model of each Uppaal Tron instance with a unique identifier which can be recognized and distinguished by our adapter.

2. Data-related behavior: Uppaal natively supports data types and variables in the definition of its machines. Despite its limited flexibility (e.g., in defining customized data types), the specification language can still be used to implement basic data-dependent behaviors. The problem is, the Uppaal engine generates a state-space which is suitable for model-checking purposes (i.e., the whole state-space). Uppaal Tron uses this state-space to infer applicable test-cases, while a non-exhaustive state-space exploration algorithm could be sufficient to generate test-cases. Some of the above-mentioned performance issues, are also rooted in this problem. Additionally, not all Uppaal data structures are also supported in Tron. For instance, passing data arrays from the the test engine to the SUT (or more precisely the adapter) or vice versa is not possible. It turns out that the performance deteriorates drastically when the specification makes use of Uppaal variables, in comparison to hardcoded values in signal names (i.e., completely unfolding the model). Due to this, we decided to implement a *specification generator*, i.e., a script which creates multiple copies of the system behavior with all data fields embedded in signal names. These complex signals must be decoded by the adapter to get access to the actual values. A similar operation should be done with the SUT outgoing signals (i.e., the adapter should encode the data values appropriately in the signal name and pass it to the tester). Although we succeeded to reach an acceptable performance using this method, we soon reached the limit of defining automata in Uppaal.

We have so far experimented with few types of transactions. We plan to include other types of transaction (such as special POS services) and other EFT devices (e.g., Automatic Teller Machines – ATMs) in our future test infrastructure.

Our test-case prioritization policy is now based on absolute statement coverage of test-cases. This can be extended in two ways: *first*, other coverage measure. particularly coverage metrics on the model should be taken into account and *second*, more complex and mature prioritization techniques can be exploited (e.g., incremental analysis of test-case coverage and assigning weights to the covered scenarios or components [6]).

Our approach to check the validity of performed transactions inside our test service layer may extend in the future to incorporate checking more business rules. However, in order to keep our adaptor still manageable we would like to add another layer of abstraction for specifying models of such business rules and an independent component which can perform the necessary checks based on the rules.

8 Related Work

Gast [12] implements an FSM-based conformance testing algorithm close to IOCO. The FSM model in Gast is specified in the functional programming language Clean. One can define abstract data types and use the generic function definition in Clean to use them in generating test-cases. In [20], Gast is used to test Java Card applet implementing an electronic purse application. The applicability of their test-technique is then demonstrated by manually injecting a number of bugs (creating mutations) and applying the automated test technique to find them. The work reported in [20] is essentially based on the same principles as our work (modulo some technical, e.g., the differences in the definition of conformance relation). We improve upon the trajectory proposed in [20] by integrating domain knowledge and code coverage metric in prioritizing test-cases.

The model-based testing environment of Microsoft called *Spec Explorer* to design and run automatic tests [23]. Their modeling language combines scenario-based modeling with state-based modeling [7, 8]. This prevents complicated conversion from the developed code (which are scenario-based) to an FSM model (which is state-based) by test designers. This can make the learning curve for model-based testing less steep. For our application domain, however, a more elaborate model of both behavior and data domain seems indefensible and hence, we believe that it pays off to spend an extra effort to build a separate model for testing purposes. The ISO 8583 standard as a reference model facilitates making this model and keeps it relatively orthogonal to the changes in implementation.

Our prioritization method is based on the work of Elbaum et al. [6, 5] in which they have analyzed and compared different test-case prioritizing techniques which helps test designers to select appropriate techniques according to their needs. We used category-partitioning in order to organize our test-case generation process. The technique was originally introduced by Ostrand and Balcer [15]. In particular, this method is more effective when enormous variety of test-cases can be generated but only some of them have real testing value.

9 Conclusions and Future Work

In this work, we developed a formal model of a high-risk financial system, called an Electronic Fund Transfer (EFT) switch, in terms if Labeled Transition Systems (LTSs). The formal model is then exploited to apply model based testing techniques in order to test such a system automatically and systematically. We used an existing test-case generator, called Uppaal Tron, and extend it with several components, to augment the test-cases with financial data and to store, evaluate and prioritize the generated test-cases. Also, to enhance the performance and to prevent state-space explosion in our testing infrastructure, we implemented the formal model of some components in the environment as a separate Java component running in parallel with our test infrastructure.

Hitherto, we have only covered few transaction types (e.g., purchase, reversal and balance enquiry) and only used POS terminals to send messages to the EFT switch. Despite this limited scope of our current implementation, the test results both in terms of coverage and detected bugs are encouraging. However, we need to overcome the limitations in the present approach in order to replace the current manual testing techniques with the model-based approach presented in this paper. Hence, we would like to extend the approach along the lines presented in Section 7 and build an in-house tool to support it.

Acknowledgments

The authors would like to thank Marius Mikucionis for his helpful hints on the Uppaal Tron engine. Tim Willemse provided useful comments on an earlier draft of this paper. This work is partially sponsored by Fanap IT corporation.

References

[1] Cobertura project. Available from `http://cobertura.sourceforge.net/`.

[2] ISO 8583 standard for financial transaction card originated messages - interchange message specifications – part 1: Messages, data elements and code values, 2003.

[3] Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, John Håkansson, Paul Pettersson, Wang Yi, and Martijn Hendriks. UPPAAL 4.0. In *Proceedings of QEST'06*, pages 125–126. IEEE CS, 2006.

[4] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors. *Model-Based Testing of Reactive Systems, Advanced Lectures*, volume 3472 of *LNCS*. Springer, 2005.

[5] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Prioritizing test cases for regression testing. In *In Proceedings of the International Symposium on Software Testing and Analysis*, pages 102–112. ACM Press, 2000.

[6] Sebastian Elbaum, Gregg Rothermel, Satya Kanduri, Satya K, and Alexey G. Malishevsky. Selecting a cost-effective test case prioritization technique. *Software Quality Journal*, 12:2004, 2004.

[7] Wolfgang Grieskamp. Multi-paradigmatic model-based testing. In *Formal Approaches to Software Testing and Runtime Verification*, volume 4262 of *LNCS*, pages 1–19. Springer, 2006.

[8] Wolfgang Grieskamp, Nikolai Tillmann, and Margus Veanes. Instrumenting scenarios in a model-driven development environment. *Information & Software Technology*, 46(15):1027–1036, 2004.

[9] Matthias Grochtmann and Klaus Grimm. Classification trees for partition testing. *Softw. Test., Verif. Reliab.*, 3(2):63–82, 1993.

[10] Anders Hessel, Kim Guldstrand Larsen, Marius Mikucionis, Brian Nielsen, Paul Pettersson, and Arne Skou. Testing real-time systems using UPPAAL. In *Proceedings of Formal Methods and Testing*, volume 4949 of *LNCS*, pages 77–117. Springer, 2008.

[11] Robert M. Hierons, Kirill Bogdanov, Jonathan P. Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, Gerald Lüttgen, Anthony J. H. Simons, Sergiy A. Vilkomir, Martin R. Woodward, and Hussein Zedan. Using formal specifications to support testing. *ACM Computing Surveys*, 41(2), 2009.

[12] Pieter W. M. Koopman and Rinus Plasmeijer. Testing reactive systems with GAST. In *Post-Proceedings of TFP'2003*, pages 111–129, Intellect, 2004.

[13] Marius Mikucionis, Brian Nielsen, and Kim G. Larsen. Real-time system testing on-the-fly. In *Proceedings of NWPT'2003*, pages 36–38, 2003.

[14] Object Management Group. UML Testing Profile Version 1.0. 2005.

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

[16] Julien Schmaltz, and Jan Tretmans. On Conformance Testing for Timed Systems. In *Proceedings of FORMATS'2008*, vol. of 5215 of *LNCS*, pages 250-264, Springer, 2008.

[17] Jan Tretmans. A formal approach to conformance testing. In *Proceedings of the IFIP International Workshop on Protocol Test systems VI*, pages 257–276, North-Holland, 1994.

[18] Jan Tretmans. Model based testing with labelled transition systems. In *Proceedings of Formal Methods and Testing*, volume 4949 of *LNCS*, pages 1–38. Springer, 2008.

[19] Jan Tretmans, and Ed Brinksma. TorX: Automated Model-Based Tesing. In *Proceedings of the European Conference on Model-Driven Software Engineering*, 2003.

[20] Arjen van Weelden, Martijn Oostdijk, Lars Frantzen, Pieter Koopman, and Jan Tretmans. On-the-fly formal testing of a smart card applet. In *Security and Privacy in the Age of Ubiquitous Computing*, volume 181 of *IFIP International Federation for Information Processing*, pages 565–576. Springer, 2005.

[21] Martin Weiglhofer, and Franz Wotawa. Asynchronous Input-Output Conformance Testing. In *Proceedings of COMPSAC'2009*, vol. 1, pages 154–159, IEEE CS, 2009.

[22] Colin Willcock, Thomas Deiß, Stephan Tobies, Stefan Keil, Federico Engler, and Stephan Schulz. An Introduction to TTCN-3. Wiley, 2005

.

[23] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. Model-based testing of object-oriented reactive systems with spec explorer. In *Formal Methods and Testing*, vol. 4949 of *LNCS*, pages 39–76. Springer, 2008.