Front. Comput. Sci. China 2010, 4(3): 354–364
DOI 10.1007/s11704-010-0387-6

RESEARCH ARTICLE

Data-driven testing methodology for RFID systems

An LU (✉)[1], Wenbin FANG[1], Chang XU[1,2], Shing-Chi CHEUNG[1], Yu LIU[3]

1 Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Hong Kong, China
2 Department of Computer Science and Technology, Nanjing University, Nanjing 210093, China
3 Institute of Automation, Chinese Academy of Sciences, Beijing 100190, China

Abstract A radio-frequency identification (RFID) system including hardware and software may be updated from time to time after first time deployment. To ensure the reliability of the system, extensive tests are required. However, enumerating all test cases is infeasible, especially when the tests involve time-consuming hardware operations. To solve this problem, we propose a testing methodology for RFID systems which does not enumerate all test cases but rather those which are representative. A clustering method is adopted in selecting representative test cases. Although a small number of selected test cases are run, we can still obtain a relatively high bug detection rate compared with running the enumerated test cases. Our extensive experiments show the efficiency and effectiveness of our testing methodology.

Keywords radio-frequency identification (RFID), testing methodology, test cases, bug detection

1 Introduction

Radio-frequency identification (RFID) is the use of radio waves to identify and track a certain object, which can be an item in a supermarket or a warehouse, a person, an animal, and so on. A typical RFID application involves tags which are attached to objects and readers which receive signals from tags to identify objects. Compared with traditional barcode systems, RFID allows tags to be read outside the line of sight of the reader, and the distance between tag and reader can be several meters. Furthermore, the integrated circuit in an RFID tag can store and process much more information than a barcode can.

RFID is now recognized as an extremely promising technology for businesses, and is already changing business processes and operations in different areas for the purpose of efficient identification, tracking and management, such as supply chain management, inventory control, electronic passports, luggage management, traffic access control and toll collection, anti-counterfeiting, and so on. For example, US superstores including Wal-Mart and Target, all have mandates requiring their vendors to ship pallets and cases of materials with RFID tags attached, and these requirements are already injecting extra momentum to RFID adoption. Firms are expected to adopt RFID technology, not only to satisfy customer requirements, but also to increase productivity, by tracking goods in the supply chain, managing business assets such as cars or computers, and many other potential business applications yet to come once the technology is mature.

A typical RFID system contains both hardware and software. However, the exploration process of deploying such a system can be overly tedious, and the results of the deployment are often too random to be reliably generalized. RFID deployments typically suffer from problems such as:

1) Receiving too little energy from tags
2) Being too vulnerable to environmental factors
3) Cross coupling of neighboring tags (i.e., the coupling of a signal from one tag to another)
4) Requiring too much tuning

Thus, when deploying an RFID system, we need to test it to ensure its reliability and quality. The testing process

Received April 30, 2010; accepted June 9, 2010

E-mail: anlu@cse.ust.hk, wenbin@cse.ust.hk, changxu@cse.ust.hk, scc@cse.ust.hk, liuy@rfidlab.cn

involves a series of activities to evaluate and determine whether the system meets its required results under controlled conditions. For example, under a certain hardware and software configuration, a user of a system in an interface performs some operations and obtains particular results. We need to ensure that results meet our requirements. The testing techniques mainly involve executing a program in order to find bugs. Such testing can be very time-consuming or even impossible, since we may need to enumerate all possible test cases. For example, in the αGate (an RFID measuring, testing, and calibration instrument [1]) application of benchmarking tests, we need to design and generate suitable test sequences for antenna positions and orientations. However, much work is required to determine the suitable positions and orientations for each antenna. Assume that the motion of each antenna has 3 degrees of motion freedom, if each degree of freedom is subject to 10 different choices, the total number of combinations for 3 antennas is 10^9. It is very time-consuming to enumerate all the test cases on different hardware configurations. For example, it takes about 15 seconds for an antenna to move and for data to be collected for each test case in the αGate application. Suppose the total combination of the test cases is 10^9, then it will take more than 475 years to complete them all, this is obviously infeasible. Also, such test cases cannot be conducted in parallel due to hardware restrictions. For example, the 3 degrees of motion freedom of each antenna cannot be separated, and the data collection process of the 3 antennas cannot be simultaneous. Therefore, we cannot use concurrent programming to reduce the testing time. In addition, a typical RFID system commonly adopts multiple readers or antennas. This fact imposes the requirement of huge test combinations. That is to say, testing RFID systems is extremely time-consuming; this is not limited to the αGate application. Furthermore, when there is a modification in software or hardware, we need to repeat all test cases to see if such modification may trigger new bugs. Therefore, it is infeasible to conduct testing on an RFID system in such costly way.

To solve the above mentioned problem, we propose a new testing methodology for RFID systems. This methodology aims at largely reducing test time but still keeping a relatively high bug detection rate. When testing an RFID system, the most time-consuming part is related to hardware operations. For example, in the αGate application, it takes several seconds for the motor of an antenna to move from one place to another, or to swing from one angle to another. And there are too many combinations of different test cases. Therefore, our solution focuses on two time-saving points. One is to use mock objects instead of real hardware components in the test. Mock objects are simulated objects that mimic the behavior of real objects in controlled ways. By applying simulated objects, real hardware operations are avoided in the test which saves time. We argue that the mock objects we use to simulate real hardware behaviors are effective and efficient. It is also important to choose representative test cases instead of all enumerated test cases. Representative test cases are a relatively small number of test cases chosen from the enumerated test cases which can detect a majority of bugs in the system compared with the enumerated test cases. We use a clustering method to achieve such purpose. In practice, the well-known *k*-means clustering algorithm is used. To evaluate our solution's effectiveness and efficiency, we adopt mutation testing in our experiments. Our experimental results show that although only a few representative test cases are adopted in the test, we still achieve relative high bug detection rate.

To summarize, for any given deployment environment, a typical RFID system needs to be calibrated and/or benchmarked for performance tuning. This must be repeated when any software or hardware is updated. Such calibration or benchmarking needs to try numerous configurations that include changing the position and orientation of each antenna. We consider trying each configuration as a *test case*. We take the αGate application as a case study to explain our solution. We argue that some features of αGate are representative even for other RFID systems. There are many common features shared among different practical RFID systems. For instance, multiple readers and antennas are used to increase the tag detection range, the positions or orientations of antennas are subject to change for performance tuning, the number of test cases is vast, adjusting the positions or orientations for an antenna takes a nontrivial time due to mechanical movement, and so on. Thus, our solution can also be applied to other RFID systems.

The main contributions of this paper are as follows.

1) We use mock objects to replace hardware components in the test in order to save on hardware operation time.

2) We propose a new testing methodology for RFID systems. Our methodology applies a clustering method to

enumerated test cases, and select representative test cases in experiments, which greatly reduces total test time. Although the number of representative test cases is limited, we still maintain a relatively high bug detection rate.

Preliminary information about clustering methods and mutation testing is presented in Section 2. Section 3 presents the testing methodology for RFID systems, mainly focusing on the αGate portal application. In Section 4, we discuss the experiments. In Section 5, we present related work. Finally, in Section 6, we offer our concluding remarks.

2 Preliminary

In this section, we review some background information about clustering methods and mutation testing which are used in our methodology.

2.1 Clustering methods

Clustering is the assignment of a set of observations into subsets (called clusters) in order to make observations in the same cluster similar to each other in some sense. Cluster analysis is a common technique for statistical data analysis, which can help to reveal the characteristics of any structure or patterns present in many fields, such as bioinformatics, medicine, market research and so on [2]. There are several types of clustering method, including hierarchical clustering, and partitional clustering. Hierarchical clustering aims to build a hierarchy of clusters by either merging smaller clusters into larger ones, or by splitting larger clusters, this is subdivided into two types: agglomerative methods and divisive methods. The agglomerative method is a "bottom-up" approach which proceeds by series of fusions of the n observations into clusters. Each observation starts in its own cluster of one, and pairs of clusters are subsequently merged to build the hierarchy. The divisive method is a "top-down" approach which separates n observations successively into finer clusters. All observations start in a single cluster, and are split recursively to build the hierarchy. Partitional clustering directly decomposes the set of observations into a set of disjoint clusters. A commonly used partitional clustering method is k-means clustering [3].

The k-means clustering method [3] is one of the simplest unsupervised learning algorithms that solve the well known clustering problem. It is a method of cluster analysis which aims to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean. As it is a heuristic algorithm, there is no guarantee that it will converge to the global optimum, and the result may depend on the initial clusters.

Given a set of observations , $x_1, x_2, \ldots, x_n$, where each observation is a d-dimensional real vector, the k-means clustering aims to partition the n observations into k sets, $k < n$, $S = S_1, S_2, \ldots, S_k$ so as to minimize the sum-of-squares criterion:

$$J = \sum_{j=1}^{k} \sum_{x_i \in S_j} \|x_i - \mu_j\|^2, \tag{1}$$

where μ_j is the geometric centroid of the data points in S_j (the mean of points in S_j).

The algorithm consists of a re-estimation procedure [4]. The n observations are first assigned to the k sets randomly or by some heuristic. The algorithm proceeds by alternating between the following two steps:

1) Update: Calculate the centroid (the new means) of the observations in each set.

2) Assignment: Assign each observation to the cluster whose centroid is closest to that observation (that is, with the closest mean).

These two steps are repeated until a stopping criterion is met: the assignments no longer change.

2.2 Mutation testing

System testing of software or hardware is testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements. System testing falls within the scope of black box testing, and as such, should require no knowledge of the inner design of the code or logic [5]. Test techniques normally include the process of executing a program with the purpose of finding software bugs.

Mutation testing is a method of software testing. It modifies program source code or byte code in small ways [6], which is called mutation, in order to help a user to develop effective tests or locate weaknesses in the test data used for the program. More specifically, mutations can be mutation operators which mimic typical programming errors (e.g. using incorrect variables or operators), or the creation of valuable tests (e.g. changing some variables to zero). During mutation testing, faults are introduced into a program by creating many different versions of the program, each of which contains one fault. Test data are

used to execute these faulty programs in order to cause each faulty program to fail.

Mutation testing was first proposed in the 1970s [7–9]. A lot of research work has been proposed to develop mutation testing into a practical testing approach. A recent detailed survey on the development of mutation testing can be found in [10], which indicates that the techniques and tools for mutation testing are reaching a state of maturity and applicability.

2.3 Mock objects

Mock objects can simulate the behavior of real complex objects in controlled ways in object-oriented programming. They are especially useful when a real object is impractical or impossible to incorporate into a software test. A mock object is used to test the behavior of some other real object. Instead of calling the real objects, the tested object calls a mock object that merely asserts that the correct methods were called, with the expected parameters, in the correct order.

3 Testing methodology for RFID systems

3.1 αGate portal

RFID is a complex and still evolving technology. The performance of RFID tags and devices can vary significantly across different brands and models. A successful RFID deployment depends not only on the product specification and standard, but also on other variables such as reader collision, tag collision, and natural environment factors. As such, it is a nontrivial task to locate precisely the cause of an unsatisfactory RFID deployment; example problems could include: problems with materials, tag reading distance, tag orientation, antenna geometry, read rate, detuned frequencies, multipath effects, reader emitting power etc. Currently, one may explore these factors using a trial and error approach. To address those environmental, material and devices constraints, αGate dynamic RFID Portal (αGate Portal) allows for truly automated device calibration, data capturing, dispatch and process control, securing data reliability and integrity by eliminating all manual human setup and intervention, streamlining those repetitive processes to identify the perfect-fit setup for those constraints.

αGate Portal is an automatic data intake, calibration and analytical portal by which the optimal positions of the merchandise, antennas and tags will be anchored. Combining the spinning metal platform, the relative RFID antennas will be automatically placed into an optimal position and orientation. Then we can capture, synthesize and analyze the data through the antennas and therefore benchmark the performance of the RFID devices and tags against a preset or variable environment.

αGate Portal is specifically designed to generate precise findings for:

1) Optimal physical layout: When designing an RFID deployment, users consider both business processes and physical layout. Where is the most logical place to locate readers for ease of asset movement and other considerations?

2) Data accuracy: One of the bigger challenges of RFID is that tags behave differently and this creates trouble in transferring data to a reader under different proximity, orientation and other environmental conditions.

3) Reliability and robustness: Ensure that the network the RFID devices operated on is highly reliable and robust.

4) Data integrity and synthesis: Challenges stem not only from capturing large volumes of data at high speed accurately, but also from turning those data captured into valuable information to facilitate decision making and integrate into business process applications.

The system overview of αGate is given in Fig. 1. α Gate Portal consists of the following five modules:

1) Multiple antenna framework (MAF). It is a physical framework consisting of portable and adjustable mounting framework (PAMF) and spinning metal platform (SMP). PAMF is based on a personalized design to facilitate relocation and reconfiguration for different environmental needs. SMP is coupled with PAMF. αGate Portal provides all-round environment configuration, including X, Y and Z axes, tag-antenna proximity and orientation setup. It is flexible and capable of supporting up to 6 RFID antennas.

2) Robotronics control (RC). It includes antenna position controller (APC) and antenna position holders (APH), moveable object platform (MOP), Smart RoboConfigurator, and Intelligent auto-park and reposition schema. Smart RoboConfigurator is used to setup the device parameters, i.e., the position, orientation, distance between the tags and antennas.

3) RITAS (RFID intelligent tags analysis system). It is a computer integrated system (CIS) with readable region analysis (RRA) software, data configuration suite, and data analytics suite. The data configuration suite measures

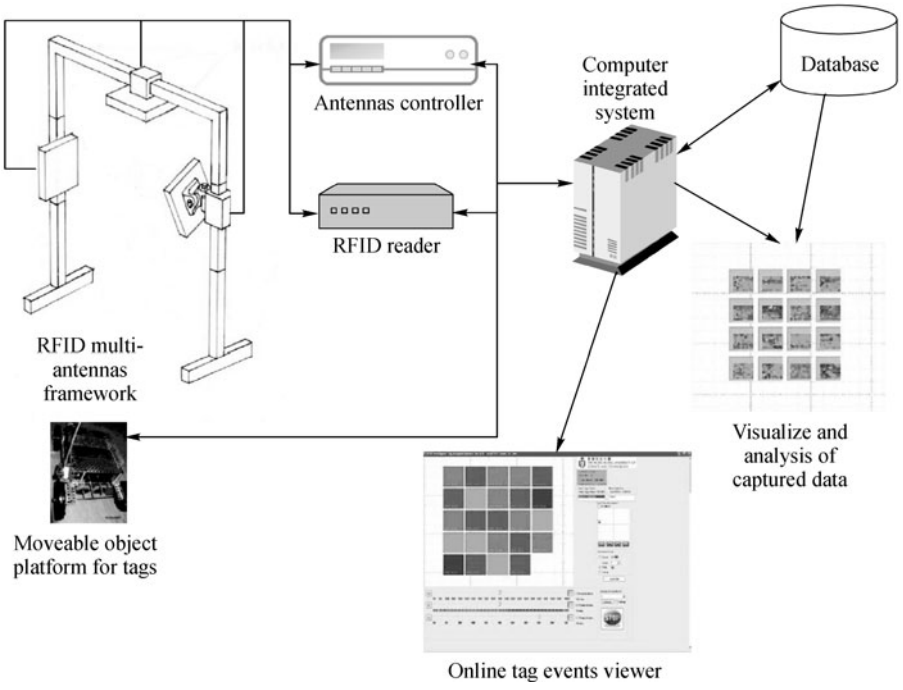

Fig. 1 System overview of αGate

configuration specifics such as environmental parameters, antenna proximity, positioning and orientation, and RFID reader parameters. It also calibrates the testing environment and parameters, such as RFID data collection process and tag reading, and modular testing schema setup. The data analytics suite implements RFID database capture and processing, it also implements RFID performance benchmarking. RFID database capturing and process includes the following functions.

- Capturing data and classification
- Data filtering, sorting and grouping
- RFID database import
- Disparate database reconciliation and integration
- Data filtering and exploration
- RFID database export / storage

RFID performance benchmarking determines the performance criteria of readers and tags, position and orientation.

4) Visual analyzer. It visualizes findings into large volume data in three-dimensional space with specific situation/application driven parameters. It also generates performance reports and provides recommendations for test case development.

5) Data management module. It provides a data backup agent.

RITAS is a data collection tool specially designed for RFID research to capture, monitor and analyze RFID tag information in a scientific and reliable manner. It combines both robotic and electronic components integrated with the powerful software tool to achieve a reusable and repeatable data capturing and analyzing processes.

The system provides a computerized mechanism for controlling the movement of the RFID antenna under a test tag platform. The user can predefine the testing location for antennas and tag platform via a user friendly graphical interface. The captured tag information from various test locations can then be visualized and analyzed by a graphical viewer in both tabular and map format. A query feature enables user to filter or consolidate the captured data from the system database. The configuration and settings for the data capturing process can also be saved and revised. The software architecture of RITAS is given in Fig. 2, where arrows show the interactions between the different components of RITAS.

3.2 Testing methodology

We identify two major challenges for rapid testing of the RFID system. First, it takes as long as 15 seconds for antenna's physical movement and data collection per test case. Second, there would be millions or even billions of test cases for αGate, due to the combinatorial nature of

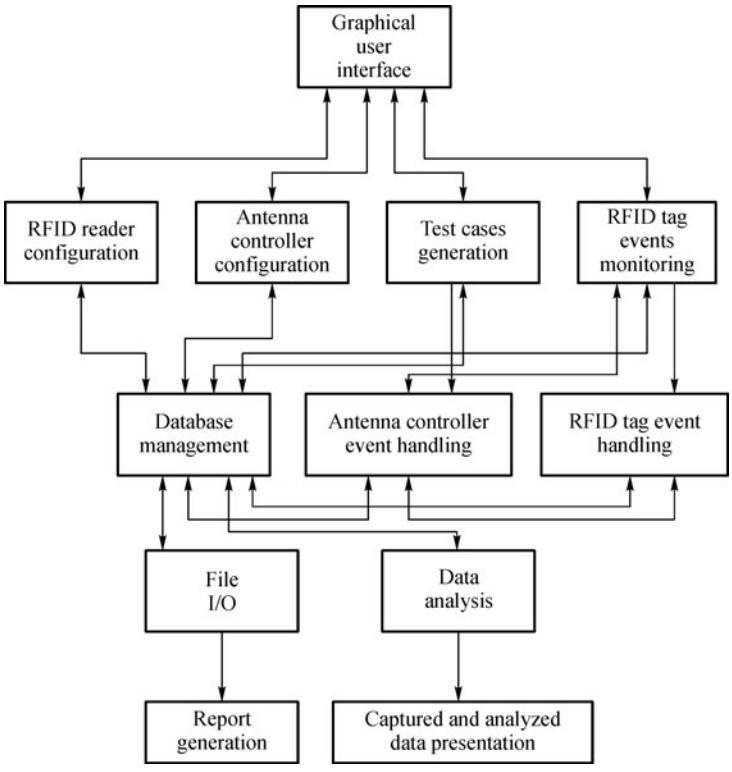

Fig. 2 Software architecture of RITAS

antenna positions. We address the first challenge by using mock object to simulate the hardware behavior, which removes the physical movement of antenna. For the second challenge, we apply a prevalent data analysis technique, clustering, to reduce the number of test cases.

3.2.1 Mock objects

The antenna controllers and RFID readers are the only two hardware devices that directly interact with RITAS. The test cases in the databases contain the information of both the antenna position and the RFID tag. Therefore, we can feed test cases to RITAS through mock objects to simulate the tag reading process of RFID readers. Since the test cases already contain the information of antenna positions, it is unnecessary to physically move the antennas. Finally, for each test case, we reduce the running time of interaction with hardware devices from 15 seconds to an average of 10 seconds.

3.2.2 Clustering

A clustering algorithm is usually an iterative operation that needs to scan the data set multiple times, so it is also a time consuming task to perform clustering on large data set, which could perhaps be even worse than running all the test cases without clustering.

To solve the above problem, we filter the test cases as the first step. The filtering process is in essence to apply a predicate to all test cases, and to obtain a small number of test cases for further clustering. Since the αGate application stores test cases in a database, we can take advantage of modern database management system's superior query performance to perform an SQL query, and obtain a few test cases in a short time. The tricky part is how to make a suitable query that reduces the number of test cases to a reasonably small amount, and at the same time, the test cases remaining can still fail as many functions as possible. That is to say, we expect those functions can definitely be failed by the test cases before filtering. We have to examine the software specification to determine a suitable query to perform. For example, a function under test only uses a subset A of all attributes in the test case, and an attribute in the test case may have a few possible values. Through such information derived from the software specification, we can remove duplicate test cases by applying the GROUP-BY operator on A. It is also possible that we select a few test cases with the most possible values of an attribute according to certain distribution of this attribute.

After filtering, we perform the clustering algorithm on the remaining test cases. In order to maintain the simplicity of our method, we adopt the *k*-means clustering algorithm. Finally, we randomly pick a representative test case from each cluster produced by the clustering algorithm. If there are k clusters, then the number of representative test cases would also be k.

There are two parameters that affect the clustering quality. The first parameter is the number of attributes in the test case. The second parameter is the number of clusters, which should be set before performing *k*-means. We can approximate both parameters from the software specification. Given M functions under test, the i-th ($i < M$) function that uses the test case with a set of attributes A_i, has N_i divergent execution paths on these attributes. We can derive the number of attributes to be $|\cup A_i|$, that is, the number of the union of all attributes in all functions under test. We approximate the number of clusters to be $\prod N_i$, so that the representative test case can cover as many divergent path as all functions. Note that noisy attributes that cannot be used in functions under test would degrade the clustering quality, and we do not need noisy attributes that will not be used in functions.

4 Evaluation

In this section, we evaluate the effectiveness of our testing methodology.

4.1 Experimental setup

Our experiments were performed on a PC equipped with an Intel dual-core CPU with 2 GB RAM, running Windows XP Pro SP3, 32-bit. We used Microsoft Visual Studio 2008 Pro, and performed data-driven unit testing using the Visual Studio Unit Testing Framework. All test cases were originally stored as tuples in a test case table (TCT) in a Microsoft SQL Server 2008 database. The TCT contained 660,000 test cases, and each test case consisted of 14 attributes. Each test case recorded the antenna position, and the information of a tag that was read in. Multiple test cases would have the same antenna position, or the same tag. Table 1 illustrates three sample test cases.

In our evaluation, we tested two core methods in RITAS. The first method was to read RFID tags from the RFID hardware system, and write the tag information into a table (not TCT) in the database. We denote this method as Method *A*. We used the Microsoft Moles technique to write mock objects for reading the tags stored in TCT, and simulate the tag reading process. According to the RITAS specification, only the attribute TPID is used in Method *A* in the test cases. The second method was to read tag information from the TCT in the database, and plot an RSSI graph. We denote this method as Method *B*. There was only an attribute RSSI involved in Method *B* according to the RITAS specification. Our methodology is also applicable to other RITAS methods that interact with an RFID hardware system.

Please note that, in our experimental context, we have a detailed development specification at hand and developers available. Therefore, we are able to know the design details of RITAS, which helps us to select suitable methods to be tested.

4.2 Mutation testing and baseline

A mutant is the result of applying one mutation operator to the method under test. We used 5 types of mutation operators, and made 2 mutants for each type. If the test cases are able to fail the mutant (e.g., causing an exception), then the mutant is said to be killed. Table 2 summarizes the mutants for Method *A* and Method *B*. In our settings, executing a mutant for Method *A* with a test case would take 16 seconds on average, while only 14 seconds on average for Method *B*. There were 660000 test cases for each mutant. Obviously, it is infeasible to run all test cases for a mutant, which would require some 120 days! Although testing would stop once a test case kills the mutant, such test cases may randomly appear in any

Table 1 Sample test cases in TCT

JobID	TagID	AntID	Freq	RSSI	Time	TimeStamp	Seq	TPID	*X*	*Y*	Angle *X*	Angle *Y*	TxPwr
Job1	01...0C	Antenna3	903.75	−51.00	12...54	2008-9-15 2:27:32	1	0000	0	200	40	0	30.00
Job1	02...BA	Antenna3	903.75	−45.00	12...54	2008-9-15 2:27:32	1	0000	0	200	40	0	30.00
Job1	01...0C	Antenna1	903.75	−60.00	12...55	2008-9-15 2:27:32	1	0000	200	0	30	0	30.00

position in TCT, so that the average testing duration is still as long as tens of days.

To make the baseline of killing mutants, we randomly picked test cases from TCT to run Method *A* and Method *B*, by setting the DataAccessMethod attribute to be Random in Visual Studio Unit Testing Framework. The framework automatically picked random test cases from TCT, and executed the method under test. For each method under test, we set a timeout to be 4 hours, so as to stop testing. We show in Table 2 whether the mutants can be killed by randomly-picked representative test cases.

Compared with the baseline testing, we expected that using our methodology, we can pick many fewer representative test cases with the least manual interference, and can kill just as many mutants as the baseline. We only focus on those mutants that can be compiled.

4.3 Filtering

In our testing environment, we examined the semantics of attributes for the test case. We derived from the specification that, as the antenna position was fixed, the RSSIs of all tags read would contain the most possible values, while TPID is kept the same. Therefore, we performed the SQL equality selection on the antenna position (the *X* or *Y* attribute in Table 1). However, it is still not enough to use only equality selection on the antenna's position. In terms of testing, we should also include the NULL case for boundary testing.

Thanks to the database system's query performance, we performed multiple equality queries on different antenna positions. Each query finished in seconds. We ran the SQL query over all test cases, and got the number of distinct RSSI values. Then, we ran the same SQL query with an equality selection on (X = [arbitrary number]), and got the number of distinct RSSI values. We found that these two numbers are very close. No matter what the antenna position was, the RSSIs included the most possible values.

Therefore, we chose two arbitrary positions $X = 400$ and $Y = 400$. Table 3 summarizes the number of test cases left after filtering. Although the number of test cases can be reduced to as few as 13 thousand, it would still take more than 2 days to run all these test cases. Thus, the clustering is still a necessary step.

4.4 Clustering

We used Cluster [11], a software containing *k*-means and hierarchical-based clustering algorithms, to perform clustering. The predicate on the filtering was "$X = 400$ OR X is NULL," as shown in Table 3.

In our experiment, we investigated how the number of

Table 2 Mutants for Method *A* and Method *B*

Mutants	Description	Method *A*	Method *B*
Mutant 1	Statement deletion	Killed	Killed
Mutant 2	Statement deletion	Killed	Killed
Mutant 3	Replace each boolean subexpression with true and false	Killed	Unkilled
Mutant 4	Replace each boolean subexpression with true and false	Unkilled	Killed
Mutant 5	Replace each arithmetic operation with another one, e.g., + with *, − and /	Cannot be compiled	Killed
Mutant 6	Replace each arithmetic operation with another one, e.g., + with *, − and /	Cannot be compiled	Killed
Mutant 7	Replace each boolean relation with another one, e.g., > with >= , == and <=	Killed	Killed
Mutant 8	Replace each boolean relation with another one, e.g. > with >= , == and <=	Cannot be compiled	Unkilled
Mutant 9	Replace each variable with another variable declared in the same scope (variable types should be the same)	Killed	Killed
Mutant 10	Replace each variable with another variable declared in the same scope (variable types should be the same)	Killed	Killed

Table 3 Test cases reduction by filtering

Filter	# of test cases left	Completeness
$X = 400$	13959	No
$X = 400$ OR X is NULL	13972	Yes
$Y = 400$	34074	No
$Y = 400$ OR X is NULL	34087	Yes

attributes and the number of clusters affect the clustering quality. According to the specification of RITAS, Method *A* and Method *B* used 2 different attributes in total, and there were 3 divergent paths in Method *A*, and 2 in Method *B*. Thus, we approximated the number of attributes to be around 2, and the number of clusters to be 6 (Section 3.2.2).

Figure 3 plots the clusters produced by *k*-means, as the number of attributes varied from 2 to 14, and the number of clusters varied from 3 to 12. We visualize clusters in 2D graphs, containing the index of test cases and the RSSI. After the clustering, we can randomly pick any test case from each cluster. For example, we picked 12 test cases in total, if the number of clusters is 12.

We examined the clustering result, and randomly picked a representative test case from every cluster. Then, we ran all mutants (shown in Table 2) of Method *A* and Method *B* by using the representative test cases. We found that:

- When there were 2 attributes, the representative test cases from 6-cluster (Fig. 3(b)), 9-cluster (Fig. 3(c)), and 12-cluster (Fig. 3(d)) can kill all mutants.
- When there were 4 attributes, the representative test cases from 9-cluster (Fig. 3(g)), and 12-cluster (Fig. 3(h)) can kill all mutants.
- When there were 6 attributes, the representative test cases from 12-cluster (Fig. 3(l)) can kill all mutants.
- When there were 14 attributes, none of the settings' representative cases can kill all mutants.

From the clustering results we see, as the number of attributes increases, clusters of test cases tend to be similar to each other. Therefore, the randomly-picked test cases from clusters tend to be similar too, so that those test cases cannot cover many divergent execution paths, which ends up failing to kill some mutants. On the other hand, as the number of clusters increases, clusters of test cases tend to

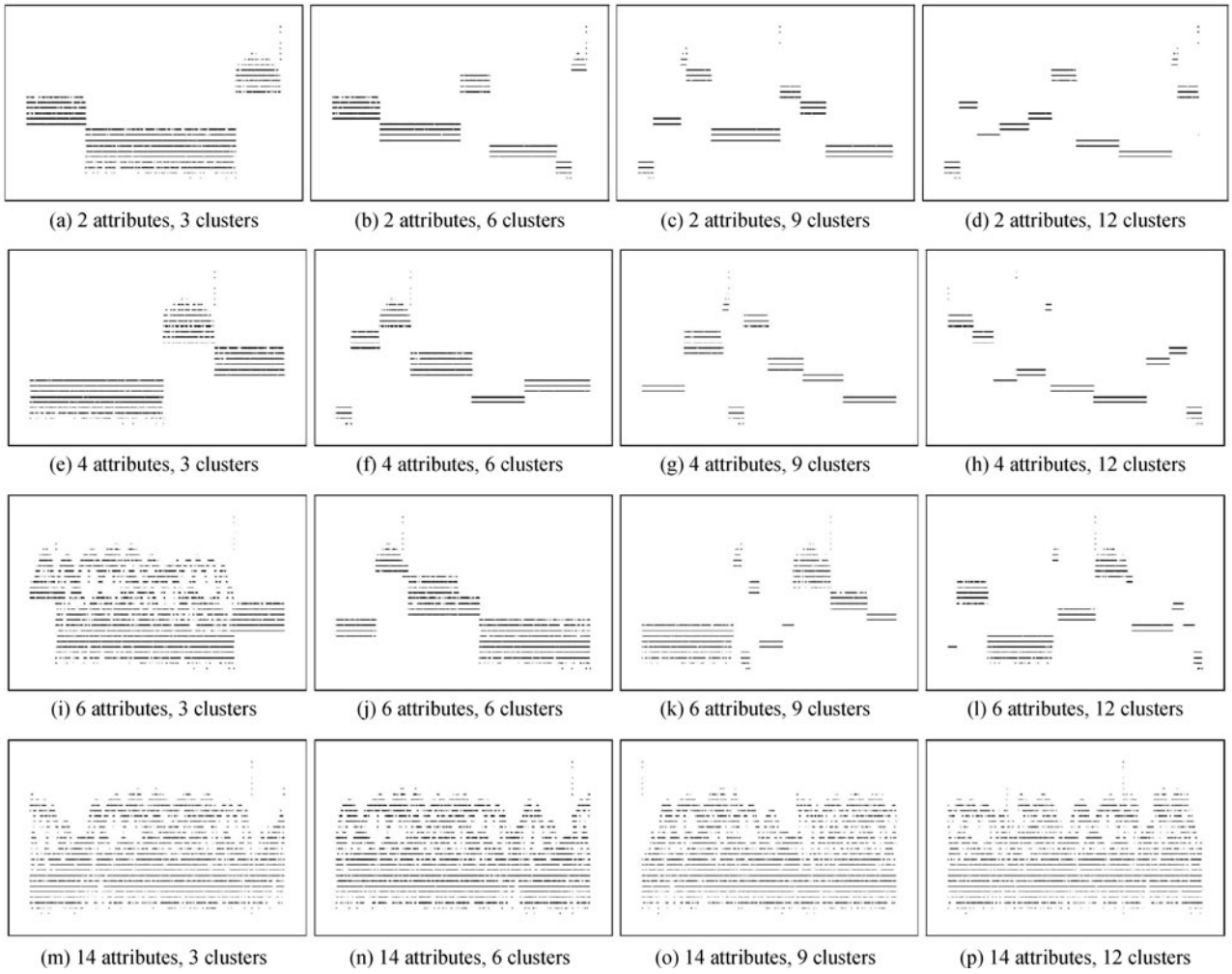

Fig. 3 Clusters of test cases. *X*-axis is the new index number of test cases; *Y*-axis is RSSI

be dissimilar to each other, so, the randomly picked test cases from clusters tend to be dissimilar too, which ends up killing as many mutants as possible. In our evaluation, the best case is to cluster test cases with 2 attributes into 6 clusters (Fig. 3(b)), which shows the effectiveness of our method. We only need to pick as few as 6 representative test cases from those clusters. In this way, to run a mutant, we significantly reduced the running time to less than 2 minutes, in comparison to 120 days to run all test cases exhaustively, which demonstrates the efficiency of our methodology.

5 Related work and discussion

Though RFID technology has developed rapidly in recent years, there exists little research work discussing a testing methodology for RFID systems which include both hardware and software. In Ref. [12], Youm et al. conduct a study on the method for testing an RFID system in a library scenario. By using integration testing and unit testing, guide lines for a system test procedure are proposed through the case study of development of RFID systems in a library. In [13], practical difficulties in implementing RFID based parcel tracking systems in the Department of Post of India are discussed. They propose strategies are to minimize the risk during implementation. Our work differs from these, as we focus on increasing the efficiency of the testing especially with vast data sets, whilst maintaining the high bug detection rate.

Our testing methodology is also applicable to RFID systems in general, for example, Internet of Things [14]. It is infeasible to collect RFID tag data whenever a test is performed. By storing tag data in database systems as part of test cases, we can reuse the data for multiple test procedures, and utilize well the highly optimized query performance of database systems to select a small amount of tags of interest. Moreover, though the RFID software system may evolve from time to time, the input data tends to be relatively stable. It is reasonable to reuse the stored tag data to test the RFID software system, for which, we simulate the tag reading process by mock objects. Additionally, well developed database privacy techniques [15] can alleviate the potential loss of personal privacy in RFID system, especially in the Internet of Things [14]. Finally, using clustering to pick representative test cases is also a one-time investment. The small number of representative test cases enables efficient testing on RFID software system, while retaining effectiveness.

6 Conclusions

In this paper, we have proposed a new testing methodology for RFID systems. Our solution largely reduces the total testing time by using mock objects instead of hardware components, and selecting representative test cases from the enumerated test cases. Though only a limited representative set of test cases is chosen, our solution can still have relatively high bug detection rate. It will be both interesting and challenging to deploy our testing methodology for Internet of things applications, and devise more efficient and effective selection methods for representative test cases as future work.

Acknowledgements This research was supported by the following grants: the Innovation and Technology Fund (ITF) (ITP/022/08LP), the Sino Software Research Institute (SSRI) (SSRI08/09.EG01), and the Research Grants Council (RGC) (RPC07/08.EG24).

References

1. RFID Benchmarking lab, http://www.rflab.org/
2. Bishop C M. Neural Networks for Pattern Recognition. Oxford University Press, 1996
3. MacQueen J B. Some methods for classification and analysis of multivariate observations. In: Proceedings of 5th Berkeley Symposium on Mathematical Statistics and Probability. Berkeley: University of California Press, 1967, 1: 281–297
4. Weisstein E W. K-Means Clustering Algorithm. From MathWorld-A Wolfram Web Resource, http://mathworld.wolfram.com/
5. Geraci A. IEEE Standard Computer Dictionary: Compilation of IEEE Standard Computer Glossaries. Piscataway: IEEE Press, 1991
6. Jefferson Offutt A. A practical system for mutation testing: Help for the common programmer. In: Proceedings of ITC. IEEE Computer Society, 1994, 824–830
7. Mathur A P. Mutation testing. Encyclopedia of Software Engineering, 1994, 707–713
8. DeMillo R A, Lipton R J, Sayward F G. Hints on test data selection: Help for the practicing programmer. Computer, 1978, 11(4): 34–41
9. Hamlet R G. Testing programs with the aid of a compiler. IEEE Transactions on Software Engineering, 1977, 3(4): 279–290
10. Jia Y, Harman M. An analysis and survey of the development of mutation testing. IEEE Transactions on Software Engineering, 2010 (in press)
11. De Hoon M J L, Imoto S, Nolan J, Miyano S. Open source clustering software. Bioinformatics, 2004, 20(9): 1453–1454
12. Youm S K, Kim J H, Cho S K. A study on the methodology for

testing of RFID system at library. In: Proceedings of MUE. IEEE Computer Society, 2007, 1076–1079

13. Inderjeet Singh Banwait K. Harihara Sudhan. Testing and integration of RFID systems in field: Experiences with department of post. In: Proceedings of ASCNT-2010. 2010, 221–227
14. Welbourne E, Battle L, Cole G, Gould K, Rector K, Raymer S, Balazinska M, Borriello G. Building the internet of things using RFID: The RFID ecosystem experience. IEEE Internet Computing, 2009, 13(3): 48–55
15. Olivier M S. Database privacy: balancing confidentiality, integrity and availability. SIGKDD Explorations Newsletter, 2002, 4(2): 20–27