Towards Automatic Test Database Generation

Carsten Binnig
SAP AG

Donald Kossmann
ETH Zürich

Eric Lo
The Hong Kong Polytechnic University

Abstract

Testing is one of the most expensive and time consuming activities in the software development cycle. In order to reduce the cost and the time to market, many approaches to automate certain testing tasks have been devised. Nevertheless, a great deal of testing is still carried out manually. This paper gives an overview of different testing scenarios and shows how database techniques (e.g., declarative specifications and logical data independence) can help to optimize the generation of test databases.

1 Introduction

Everybody loves writing new code; nobody likes to test it. Unfortunately, however, testing is a crucial phase of the software life cycle. It is not unusual that testing is responsible for 50 percent of the cost of a software project. Furthermore, testing can significantly impact the time to market because the bulk of testing must be carried out after the development of the code has been completed. Even with huge efforts in testing, a report of the NIST [16] estimated the cost for the economy of the Unites States of America caused by software errors in the year 2000 to range from $\$22.2$ to $\$59.5$ billion (or about 0.6 percent of the gross domestic product).

In the early days of software engineering, most of the testing was carried out manually. One of the big trends of modern software engineering is to automate testing activities as much as possible. Obviously, machines are cheaper, faster, and less error-prone than humans. The key idea of test automation is that tests become programs. Writing such test programs is not as much fun as writing new application code, but it is more fun than manually executing the tests [3]. Automating testing is particularly attractive for the maintenance of existing software products. With every new release of a product, the implementation of a change request, or a change in the configuration of a deployment, a series of similar tests need to be carried out in order to make sure that the core functionality of the system remains intact. In fact, most software vendors carry out nightly so-called regression tests in order to track changes in the behavior of their software products on a daily basis.

In a nutshell, test automation involves writing and maintaining code and it is just as difficult as writing and maintaining application code. In fact, as will be argued, writing and maintaining test code is more difficult because it has additional dependencies. One such dependency which is of particular interest to this work is the *test database* which needs to be built and maintained together with the test code as part of a test infrastructure.

In order to deal with the complexity of managing test code, the software engineering community has developed a number of methods and tools. The main hypothesis of this paper is that testing is (to a large extent) a *database problem* and that many testing activities can be addressed best using database technology. It is argued that test code should be declarative. In particular, the specification of a test database should be declarative, rather

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

than a bunch of, say, Perl scripts. Given such a declarative specification (actually, we propose the use of SQL for this purpose), testing can be optimized in various ways; e.g., higher coverage of test cases, less storage overhead, higher priority for the execution of critical test cases, etc. Another important argument in favor of a declarative specification of test code is the maintainability and evolution of the test code. Again, the belief is that logical data independence helps with the evolution of test code and test databases in similar ways as with the evolution of application or system code. Finally, testing actually does involve the management of large amounts of data (test runs and test databases).

Of course, all software quality problems could be solved with formal methods of program verification, if they would work. Unfortunately, they do not work yet for large-scale systems and breakthroughs in this field are not foreseeable in the near future. Fortunately, test automation can benefit nicely from the results of the formal methods community, as will be shown in Section 3.

The remainder of this paper is organized as follows. Section 2 defines the many different aspects of testing. Section 3 gives an overview on how database technology can be used in order to generate test databases. Section 4 describes some research problems that we believe can be addressed by the database community.

2 The Big Picture

There has been a great deal of work in the area of software quality assurance. Obviously, one reason is the commercial importance of the topic. Another reason is that testing involves many different activities. This section gives a brief overview.

First of all, testing requires a *test infrastructure*. Such a test infrastructure is composed of four parts:

1. An installation of the *system under test (SUT)*, possibly on different hardware and software platforms (e.g., operating systems, application servers). The SUT can be a whole application with its customizations (e.g., an SAP R/3 installation at a particular SAP customer), a specific component, or a whole sub-system. One specific sub-system that we are particularly interested in is the testing of a database management system as needs to be carried out by all DBMS vendors (e.g., IBM, Microsoft, MySQL, Oracle, Sybase).

2. A series of *test runs*. A test run is a program that involves calls to the SUT with different parameter settings [13]. Depending on the kind of test (see below), the test run can specify preconditions, postconditions, and expected results for each call to the SUT. Test runs are often implemented using a scripting language (e.g., Perl), the same programming language as the SUT (e.g., ABAP, Java, or VisualBasic), or some declarative format (e.g., Canoo's WebTest [1] and HTTrace [12]). In practice, it is not unusual to have tens of thousands of test runs.

3. *Test Database:* The behavior of the SUT strongly depends on its *state*, which is ideally captured in a database. In order to test a sales function of a CRM system, for instance, the database of the CRM system must contain customers. When testing a DBMS, it does not make much sense to issue SQL queries to an empty database instance (at least not always). In complex applications, the state of the SUT might be distributed across a set of databases, queues, files in the file system (e.g., configuration files), and even encapsulated into external Web Services which are outside of the control of the test infrastructure. Obviously, for testing purposes, it is advantageous if the state is centralized as much as possible into a single test database instance and if that test database instance is as small as possible. For certain kinds of tests (e.g., scalability tests), however, it is important to have large test database instances. In order to deal with external Web Services (e.g., testing an online store which involves credit card transactions), a common technique is to make use of mock objects [4].

4. *Test Manager:* The test manager executes test runs according to a certain schedule. Again, schedules can be specified manually or computed automatically [13]. During the execution of a test run, the test manager

records differences between the actual and expected results (possibly response times for performance tests) and compiles all differences into a test report. Furthermore, the test manager controls the state of the test database if test runs have side effects.

Establishing such a test infrastructure is a significant investment. In practice, test infrastructures are often built using a mix of proprietary code from vendors of components of the SUT, from home-grown developments of test engineers, and dedicated testing tools (e.g., from Mercury or as part of IBM's Rationale Rose suite). Often, a great deal of testing needs to be carried out manually, thereby making the test team the fifth component of the test infrastructure. To the best of our knowledge, there is no silver bullet solution on how to best build a test infrastructure; likewise, there is no silver bullet solution to evolve a test infrastructure when the SUT evolves. The situation becomes even worse when considering the other dimensions of testing such as the granularity of testing (component test vs. integration test vs. system test), kinds of test (functional specification vs. non-functional requirements such as scalability, security, and concurrency), and the time at which tests are carried out (before or after deployment).

Obviously, we have no coherent solution to address all these test scenarios. Nevertheless, we believe that declarative specifications help in most cases. As an example show case, the next section shows how declarative specifications can be used in order to automate one particular activity that is important for software quality assurance: the generation of test databases.

3 Generating Test Databases

This section presents several related techniques in order to generate test databases. Both the generation of test databases in order to test application systems such as ERP and CRM systems and the generation of test databases specifically for the testing of DBMS are studied. The novel feature of these techniques is that the generation of the test databases is *query* and/or *application-aware*. This way it is possible to generate *relevant* test databases that take characteristics of the SUT and test case into account. Traditionally, generic tools to generate test databases (e.g., IBM DB2 Test Database Generator [2], [9], [8], or [14]) generate a test database based on the schema only (and possibly some constants and scaling factors). As a result, many test databases in practice are either manually constructed (possibly using the result of a generic database generator as a starting point) or constructed using scripts that must be programmed by the developer of the SUT for that particular purpose.

Query-aware and/or *Application-aware* generation of test databases has two advantages. First, the generation of test databases is simplified; only high-level declarative specifications are needed in order to generate a test database: the programming of scripts or manual adjustments are typically not needed. Second, the evolution of the system is easy. When the SUT changes and additional test data is needed, only the high-level declarative description needs to be adjusted. As shown in Section 3.2, often it is only necessary to provide an additional (SQL) query in order to specify the missing part that needs to be generated for the evolved SUT.

3.1 Reverse Query Processing

Traditional query processing takes a database and a (SQL) query as input and returns the result of that query for that specific database. The key idea of reverse query processing (RQP, for short) is to turn that process around. The input of RQP is a query, a query result and a database schema (including integrity constraints); the result is one possible database which has the property that if the query is applied to that database, the specified query result is produced. Furthermore, the generated database meets all constraints specified in the database schema.

The most obvious application of RQP is the generation of test databases. In an OLAP application, for example, RQP can be used to compute test databases from the definition of a data cube and an example report. In OLTP applications, typically, several queries are needed in order to specify a meaningful test database (Section

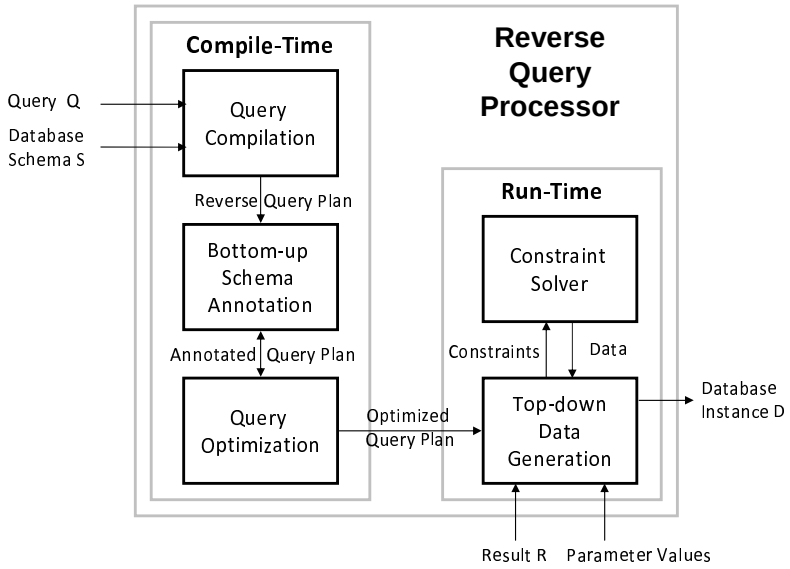


Figure 1: RQP Architecture

3.2). However, in general RQP has many more applications; e.g., security and the maintenance of materialized views.

In principle, there are many different database instances which can be generated for a given query and a result of that query. Depending on the usage of the test database, some of these instances might be better than others. For functional testing of a database application, RQP should generate a small database which satisfies the correctness criteria mentioned above, so that the running time for executing the tests is reduced. Thus, our prototype implementation tries to generate a minimal test database for a given query and a result of that query. However, other implementations are conceivable that satisfy different properties. The details of our implementation are described in [5].

Figure 1 shows the architecture of our implementation. In many ways, it resembles the architecture of a traditional (forward) query processor: A query is parsed, analyzed, optimized, and executed. Some of the key differences are that SQL queries are parsed into a *reverse relational algebra*, the optimizations are very different than in traditional query processing, and that the run time algorithms are quite different.

The reverse relational algebra can be seen as the reverse variant of the traditional relational algebra and its extensions for group-by and aggregation [10]. Consequently, executing reverse relational algebra operators at runtime involves generating data. For example, the reverse projection operator generates columns while the forward projection operator deletes columns. In order to generate data that satisfies the constraints of the query (e.g., a selection predicate) and the database schema, a decision procedure of a model checker is called by some reverse relational algebra operators. This is one example in which test automation benefits from results of the formal methods community.

In theory, reverse query processing is not decidable; that is, it is not always possible to determine whether a database exists that meets the schema and the RQP correctness condition. In practice, however, RQP is effective. For instance, RQP can be applied to all queries of the TPC-H benchmark and to all queries that we have encountered so far. For complex queries with aggregation, RQP is not trivial and involves quite complex computations. In our experiments with queries of the TPC-H benchmark, the bandwidth to generate test data on a Linux AMD Opteron 2.2 GHz Server with 4 GB of main memory varied from 600GB per hour in the best cases to around 100MB per hour in the worst cases.

3.2 Multi Reverse Query Processing

In contrast to OLAP applications which implement reports that read a huge amount of correlated data from the database, OLTP applications usually implement use cases that execute a sequence of actions wherein each action reads or updates only a small set of tuples in the database. As an example, think of an online library application. One potential use case of such an application is that a user wants to borrow a book. The sequence of actions which is implemented by that use case could be as follows:

1. The user enters the ISBN of the book (where the ISBN is unique for each book of the library).
2. The system shows the details of that book.
   - Exception 1: The book is borrowed by another user. The system denies the request.
   - Exception 2: The book belongs to the closed stack of the library. The system denies the request.
3. The user enters personal data (username, password) and confirms that she wants to borrow the book.
4. The system checks the user data and updates the database.
   - Exception 3: The user has entered an incorrect username or password. The system denies the request.
   - Exception 4: There are charges on the user account that exceed a certain limit. The system denies the request.

Functional testing the implementation of such a use case means that we have to check the conformance of the implementation with the specification of the functionality [4] (i.e., the use case). Consequently, we need to create a set of test cases to test the correctness of the different execution paths of a use case. In order to execute all the test cases of an OLTP application, one or more test databases need to be created. For example, in order to test the use case above, a test database needs to be created which comprises the different types of books (i.e., books which are already borrowed by another user or not, and books which belong to the closed stack and other books which do not) and different user accounts (i.e., user accounts with and without charges which exceed a certain limit).

In order to specify a test database for the test cases of an OLTP application, one SQL `SELECT` query and one expected result are usually not sufficient. The reason is that most test cases of an OLTP application *read* or *update* different tuples in the database that are not necessarily correlated. Therefore, in order to specify the relevant values of the tuples that are read or updated by a particular test case, we suggest that a tester uses SQL as a database specification language; i.e., the tester specifies the test database for one test case by *manually* creating a set of SQL `SELECT` queries and their expected results (called test database specification). A test database which returns these expected results for all the given SQL `SELECT` queries enables the execution of a particular test case of an OLTP application. Compared to RQP where the queries are derived from the definition of a data cube, in MRQP the queries for the test database specification are not extracted directly from the code of the OLTP application. Consequently, the queries in the test database specification are independent from the SQL statements implemented by the OLTP application (i.e., the `SELECT`, `INSERT`, `UPDATE`, and `DELETE` statements).

For example, in order to execute a test case for the use case discussed before where the user borrows a book successfully (i.e., no exception occurs), the test database needs to comprise a book with a particular ISBN which does not belong to the closed stack and is not borrowed by another user (i.e., the attribute $b\_closedstack$ must have the value $false$ and the value of the attribute b_uid must be *NULL*) as well as a user whose charges do not exceed a certain limit (e.g., $20). The desirable database state, can be specified by multiple queries and the corresponding expected query results (Figure 2a) and the database schema of the application (Figure 2b). By doing so, the tester can focus on the data that is relevant (e.g., the values for b_isbn and $b\_closedstack$ specified by Q_1 and R_1) and she does not have to take care of the irrelevant data (e.g., the values for b_title).

RQP is not capable to support multiple queries and the corresponding expected results as input. Thus, in [6] we studied the problem of Multi-RQP (or MRQP for short). Unlike RQP, MRQP gets a *set of SQL `SELECT`*

```
Q1: SELECT b_closedstack, b_uid
    FROM book
    WHERE b_isbn= '12345'
R1: {<false, NULL>}


Q2: SELECT u_password, u_charges
    FROM user
    WHERE u_name='test'
R2: {<test, 0.0>}
```

(a) Test database specification

```
CREATE TABLE user (
u_id INTEGER PRIMARY KEY,
u_name VARCHAR(20) UNIQUE,
u_password VARCHAR(20),
u_charges FLOAT NOT NULL
  CHECK(u_charges>=0));


CREATE TABLE book (
b_id INTEGER PRIMARY KEY,
b_title VARCHAR (20) NOT NULL,
b_closedstack BOOLEAN NOT NULL,
b_isbn VARCHAR(20) UNIQUE,
b_uid INTEGER FOREIGN KEY
 REFERENCES user(u_id));
```

(b) Database schema S

u_id	u_name	u_password	u_charges
1	test	test	0.0

user

b_id	b_title	b_closed stack	b_isbn	b_uid
1	TitleA	false	12345	NULL

book

(c) Test database D

Figure 2: MRQP Example for OLTP testing

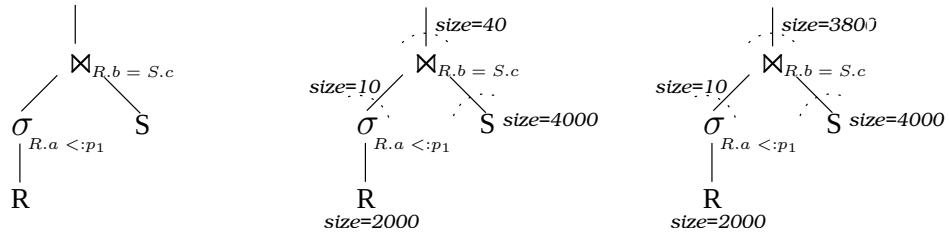
queries, the *corresponding expected query results* and a database schema as input and tries to generate one test database that returns the expected results for all the given queries. A test database which could be generated for the example above is shown in Figure 2c. In [6] we showed that MRQP is undecidable for arbitrary SQL SELECT queries. Consequently, we defined a database specification language called MSQL based on SQL for which the MRQP problem becomes decidable. Moreover, based on MSQL we suggested a solution for MRQP which utilizes the RQP engine discussed in Section 3.1.

3.3 Symbolic Query Processing

Symbolic query processing (SQP), which first appeared in [7], is a fusion of traditional query processing and a formal verification technique called symbolic execution [15]. In symbolic query processing, the data in a database is represented by symbols and a database query manipulates symbolic data rather than concrete data. One predominant application of SQP is to test components of DBMSs.

In the database industry, when a component or a technique is going to be integrated into a DBMS, it is necessary to validate the system correctness and evaluate the relative system improvements under a wide range of test cases and workloads. Consider that there is a new join algorithm available and a company which offers a commercial DBMS product wants to evaluate the performance of that algorithm in their DBMS product. For example, the company wants to know how much memory would be taken by such algorithm during the execution of a simple query like the one in Figure 3a. Usually, given the test query Q like the one in the figure, different *test cases* can be constructed by varying the results of the query (operators). In DBMS component testing, a test case T is a parametric query Q with a set of constraints (e.g., output cardinality) C annotated on the operators of the query. Figure 3b shows an example test case T_1 that is based on the given query Q in Figure 3a. Test case T_1 enforces that if the test query Q is executed on a test database D with two tables R and S (where R and S have 2000 and 4000 tuples respectively), then the intermediate selection $\sigma_{R.a<:p_1}$ (a is an attribute in table R and $:p_1$ is a parameter) and the final join result are expected to have exactly 10 and 40 tuples respectively. Test case T_1 is helpful to test how much memory the join algorithm would take when its two inputs have large size differences and the final result is small. As another example, test case T_2 in Figure 3c can test the memory consumption of the join algorithm when its two inputs have large size differences but the final result is big (3800 tuples).

Currently, testing the components of a DBMS is a manual process and thus very time consuming. It is a manual process because no tools are able to generate test databases that can fulfill the cardinality requirements of a test case. For example, in order to execute test case T_1 in Figure 3b, a tester first needs to use a normal database generator (e.g., IBM DB2 Test Database Generator, [9], [8], or [14]) to generate a test database D with



(a) The test query Q (b) Test case T_1 (c) Test case T_2

Figure 3: Examples for DBMS component testing

two tables R and S, and then *manually* adjust the content in R and S in order to ensure that the execution of Q can obtain the desired (intermediate) query results (e.g., 10 tuples should be returned by the selection $\sigma_{R.a<:p_1}$).

SQP can be used to build a database generator to automate this testing process. In fact, a test database generator called QAGen has been developed by us using SQP [7]. QAGen is a "Query-Aware" test database GENerator which generates a query-aware test database for a particular test case. It takes as input a database schema M and a test case T, and directly generates a database D and query parameter values P such that D satisfies M and $Q_P(D)$ satisfies C (where $Q_P(D)$ means the execution of query Q with parameter values P on database D, and, C are the constraints defined in T). For the example test case T_1 in Figure 3b, QAGen first instantiates the two tables R and S. In particular, table R consists of 2000 *symbolic tuples* (a symbolic tuple is a tuple containing symbols rather than concrete values; see [7] for details) and table S consists of 4000 symbolic tuples. Afterwards, the input query is evaluated by a *symbolic query engine* in QAGen. The symbolic query engine follows the paradigm of traditional query processing; i.e., each operator is implemented as an iterator, and the data flows from the base tables up to the root of the query tree [11]. In addition, the operators in the symbolic query engine manipulate input data (which are symbolic tuples) according to (1) the operator's semantics and (2) the test-case-defined constraints. On the one hand, (1) and (2) are transformed into a set of propositional constraints and the set of constraints is imposed on a subset of input tuples (and returned to the parent operator). On the other hand, the same set of constraints is directly imposed on a subset of tuples in the base tables. For the example in Figure 3b, the selection operator would impose the constraint $R.a <: p_1$ on ten of its input tuples (as well as $R.a \geq: p_1$ on all other input tuples) and return the ten tuples which pass the selection operator to the join operator. At the same time, the selection operator would impose the same constraint on the corresponding symbolic tuples in table R as well. At the end of symbolic query processing, the tuples in the base tables would capture all the requirements (constraints) defined in the input test case but without concrete data. Finally, QAGen uses a constraint solver to instantiate the constrained tuples in the base tables to obtain the final test database.

By using SQP, it could be shown that QAGen is able to generate test databases for a variety of complicated test cases. In our experiments with queries of the TPC-H benchmark, the bandwidth to generate test data on a Linux AMD Opteron 2.2 GHz Server with 4 GB of main memory varied from 230MB per hour in the best cases to 6MB per hour in the worst case [7].

4 Outlook

Test automation is an important technique in order to reduce the cost and time to market of software projects. Since there are many different test scenarios with different facets, a large number of alternative tools have been developed in order to support automated testing. Most of these tools are ad-hoc and support only one particular testing activity (e.g., the generation of test reports for a large number of test runs.)

This work made the hypothesis that test automation is a *database problem*. It was shown how test databases for OLAP and OLTP applications and DBMSs can be specified using SQL queries. The ultimate goal is to

support test engineers even further and to have more stable specifications of test activities.

The whole area of test automation is still in its infancy. There are still a number of open questions. We believe that in particular the following questions can be addressed using database techniques and plan to study these topics as part of future research:

- Optimize the generation of test databases; that is, generated databases with certain additional properties (e.g., the smallest possible test databases that meet the requirements or the fewest possible set of test databases).
- The evolution of test databases and test runs is a pressing issue for most software vendors who need to re-program a great deal of their test infrastructure with every major release.
- Testing distributed systems with virtualization is still a largely unexplored area; in such systems, the SUT is not known prior to deployment and changes dynamically.
- Testing the concurrency and scalability properties of a system is also largely unexplored.

References

[1] Canoo webtest. http://webtest.canoo.com.

[2] IBM DB2 Test Database Generator. http://www-306.ibm.com/software/data/db2imstools/db2tools/db2tdbg/.

[3] K. Beck and E. Gamma. Programmers love writing tests., 1998. http://members.pingnet.ch/gamma/junit.htm.

[4] R. V. Binder. *Testing object-oriented systems: models, patterns, and tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[5] C. Binnig, D. Kossmann, and E. Lo. Reverse query processing. In *Proc. of ICDE*, pages 506–515, 2007.

[6] C. Binnig, D. Kossmann, and E. Lo. Multi RQP Generating Test Databases for the Functional Testing of OLTP Applications. Technical report, ETH Zurich, 2008.

[7] C. Binnig, D. Kossmann, E. Lo, and M. T. Özsu. QAGen: generating query-aware test databases. In *Proc. of SIGMOD*, pages 341–352, 2007.

[8] N. Bruno and S. Chaudhuri. Flexible database generators. In *Proc. of VLDB*, pages 1097–1107, 2005.

[9] D. Chays, Y. Deng, P. G. Frankl, S. Dan, F. I. Vokolos, and E. J. Weyuker. An AGENDA for testing relational database applications. *Softw. Test., Verif. Reliab.*, 14(1):17–44, 2004.

[10] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book.* Prentice Hall PTR, 2001.

[11] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[12] F. Haftmann, D. Kossmann, and A. Kreutz. Efficient regression tests for database applications. In *Proc. of CIDR*, pages 95–106, 2005.

[13] F. Haftmann, D. Kossmann, and E. Lo. A framework for efficient regression tests on database applications. *The VLDB Journal (Best of VLDB 2005)*, 16:145–164, 2007.

[14] K. Houkjær, K. Torp, and R. Wind. Simple and realistic data generation. In *Proc. of VLDB*, pages 1243–1246, 2006.

[15] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[16] RTI. The economic impacts of inadequate infrastructure for software testing. May 2002. www.nist.gov/director/prog-ofc/report02-3.pdf.