Focused Iterative Testing: A Test Automation Case Study

Mechelle Gittens, Pramod Gupta, David Godwin, Hebert Pereyra, Jeff Riihimaki
IBM Corp.

Abstract

Timing-related defects are among the most difficult types of defects to catch while testing software. They are by definition difficult to reproduce and hence they are difficult to debug. Not all components of a software system have timing-related defects. For example, either a parser can analyze an input or it cannot. However, systems that have concurrent threads such as database systems are prone to timing-related defects. As a result, software developers must tailor testing to exploit vulnerabilities that occur because of threading. This paper presents the Focused Iterative Testing (FIT) approach, which uses a repetitive and iterative approach to find timing-related defects and target product areas with multi-threaded characteristics by executing system tests with a multi-user test suite. *Keywords: software testing, database management systems, multi-threaded applications*

1 Introduction

IBM® DB2® for Linux®, UNIX®, and Windows® (DB2 software) is a complex distributed, multi-process, and multi-threaded system. It consisting of several million lines of source code. Execution optimization is crucial for DB2 software, and overhead from instrumentation and monitoring must be minimized.

Atomicity, Consistency, Isolation, Durability (ACID) requirements must be maintained regardless of system failures that are due to unexpected events such as power outages. After an outage, when the operating system and database restarts, the database has to replay logs of the previous database activity, so that there are no partial transactions and so that other ACID requirements are met to keep the database in a consistent state. However, in a multi-threaded, multi-process system, small timing holes[1] often exist and elusive point-in-time defects can occur. The point-in-time defects are elusive because when such an unexpected event occurs, the logs must capture concurrent events and interleave them in the manner in which they occurred so that states are repeated as they occurred previously and together.

Within this context, the DB2 software quality assurance team varied the test approaches in several ways to trigger point-in-time (timing-related) problems. These methods attempted to simulate the unexpected external issues common to databases and included: (1) Varying the processor load by running an external program to consume most of the CPU cycles available to the database server; (2) Instrumenting code to selectively slow down execution with logging overhead; (3) Changing priorities of processes; and (4) Iteratively executing commands or programs with a background workload.

Copyright 2008 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.
Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

[1] A timing hole is an unexpected point in the state space of execution for the software, where multiple threads or processes interleave in such a way as to create an incorrect logic sequence that may cause the program to hang, crash or behave incorrectly.

The main contribution of the work presented here is therefore a testing methodology with automation for complex multi-threaded database software that iteratively executes commands or programs with a background workload, since that was the most successful approach. The first three methods found defects by burdening the system resources with the various kinds of overhead mentioned in methods 1, 2 and 3. These methods however, still executed events limited sequences of events against the database, and though the additional overhead deprived the events of resources, the large set of execution possibilities (the state space) could not be representatively sampled. The iterative approach, though straightforward, randomly samples every combination of the sequences of possible events, triggering timing-related defects (TRDs) that are possible in a user environment.

The paper continues as follows. Section 2 reviews work related to testing TRDs in multi-threaded database applications. Section 3 introduces the aspects of three facilities of DB2 software that make those functions - monitoring, fast communication manager and crash recovery - suitable for testing with this method. The paper then presents the methodology for focussed iterative testing in Section 4. This section includes a description of the tool support. Section 5 summarizes the results of using the approach. Section 6 provides a summary of conclusions and potential directions for future work.

2 Related Work

Testing for database systems is normally done under the conventional and general testing approaches. These include unit, functional and system testing[6]. All of these methods have been applied successfully to database applications and there have been a plethora of discussions and work in the more general applications. However, as databases grow larger and more distributed, the context of the timing hole has become an issue unadrressed by most of the more general methods. Point-in-time interleaving of states on the stack causes transient problems. In order to encounter these problems in the various combinations and permutations of states, some testing with a statistical focus must be performed.

Random testing has proven itself cost effective and more powerful than would be thought a priori[2]. In his text also called Random Testing[3], Hamlet discusses the perception of random testing as haphazard testing, done hastily and poorly. He consequently explains that the correct meaning of random testing is one where test cases are chosen with no relationship between them. This gives statistical independence to the test points that endows statistical significance to the testing results and prediction of the expected quality of the software.

The literature covering the random testing of database systems falls into the category of randomly generating inputs for the database tests, such as the evolutionary development of queries in recent work by Bati et al.[1], where the researchers create new queries by mutating and synthesizing queries, and determining whether those queries can be used to generate further queries. Although a useful approach yielding new defects, the approach omits the issue of testing for the elusive timing hole.

Outside of the database domain other approaches to testing that handle multi-threading have existed for several years. These methods use model-checking approaches to handle the interleaving of concurrent processes and the unexpected interactions. However, the issue here is state explosion, especially with the multiplicity of states possible in the database system with tens of millions of lines of code with several interacting components. The same point-in-time timing defects are still experienced because the full population of states is still underrepresented.

Sen [5] has very recently investigated, partial-order random testing approaches that choose thread schedules at random. This approach, though yielding more defects than a nonspecific random set of tests, and demonstrating that it is useful to detect the exceptions faster than the nonspecific set of tests, was only demonstrated for three small multi-threaded programs from the Java PathFinder[8] distribution. These do not compare to complex interleaving of a large DBMS such as DB2. We cannot however say, that extrapolation and repetition of the runs would not simulate a similar execution to DB2, but since this work is recent this question will have to be explored as future work. In addition, these methods such as the one by Sen[5] did not exist when we sought to

meet the TRD challenges.

In addition to the partial order methods, model-checking algorithms exist to limit the state space that must be searched to test a multi-threaded application[7]. Model checkers use the context switches that occur when a thread temporarily stops execution and a different thread starts, and systematically or iteratively suspends or binds execution of the thread at some random or arbitrary point to allow other more interesting threads to continue. One of the benefits is that the total number of executions in a program is polynomial in the number of steps taken by each thread and makes it theoretically feasible to scale systematic exploration to large programs without sacrificing the ability to go deep into the state space. This method shows potential but once again, the experimental space is preliminary with the tested programs ranging from 84 lines of code to just over 16,000 lines of code.

Having reviewed the existing work at the time and today, we found no methods to handle TRDs in a complex multi-threaded database application; however, we noted from the existing work that random testing was most suitable. We therefore created the approach presented here.

3 The Software under Study and the Components of Interest

FIT works because of the way in which functionalities such as crash recovery and monitoring are implemented in a multi-threaded system such as a DBMS. Here we explore the algorithms behind DB2 monitoring, fast communication manager, and crash recovery components and we will see why this iterative approach is particularly productive.

3.1 Monitoring

The database system monitor stores information it collects in entities called monitor elements. Each monitor element stores information regarding one specific aspect of the state of the database system. Monitor elements collect data for one or more logical data groups. A logical data group is a collection of monitor elements that gather database system monitoring information for a specific scope of database activity. Monitor elements are sorted in logical data groups based on the levels of information they provide. For this discussion, two levels are considered: database and application.

Monitor elements collect data for one or more logical data groups. These groups are collections of monitor elements that gather database system monitoring information for a specific scope of database activity. Monitor elements are sorted in logical data groups based on the levels of information they provide. The database and application levels are discussed here.

Snapshot monitor is one way that DB2 software makes element values available. Snapshots provide a point-in-time picture of the database state. During snapshot processing all relevant levels are read to complete the snapshot. As a result, for a database snapshot, the following events occur:

1. Read element values from the application-level structure. This will contain values for all terminated applications since the database activated.

2. Iterate through each application currently connected to the database. For each of these: (a) Read the element values from the application level structure. This structure contains values for all agents that have disassociated from this application. (b) Iterate through agents currently associated with the application. For each of these read the element values from the agent-level structure.

In the snapshot output, the rows_read element will contain the total from all these levels.

Event monitors are a second facility with which DB2 software makes element values available. Event monitors provide a real-time trigger-based monitoring capability. The event monitor infrastructure buffers records, using two internal buffers, before writing them to disk.

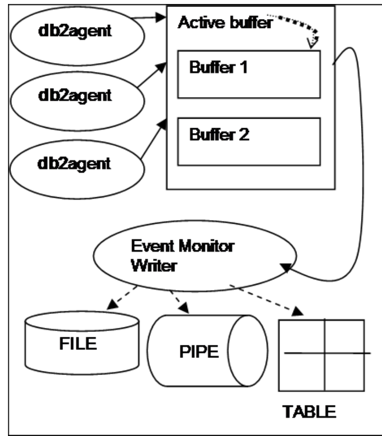

Figure 1: Generating event records for a STATEMENT event monitor

Figure 1 illustrates how event records are generated for a STATEMENT event monitor. Three applications are connected to the database, each having a single agent working on its behalf. As each application executes SQL statements, their agents generate new statement event monitor records and insert them into the active buffer, which is Buffer 1. When Buffer1 has filled up, a message is sent to the event monitor writer instructing it to process all records in Buffer 1. During this period, the “active” buffer is switched from Buffer 1 to Buffer 2 and the three applications will begin filling up Buffer 2. The Fast Communication Manager (FCM) event monitor writer, having received the message, processes Buffer 1 and inserts the data into files, if the event monitor is a FILE event monitor; or a named pipe, if the event monitor is a PIPE event monitor; or SQL tables, if the event monitor is a TABLE event monitor.

3.1.1 Opportunities for FIT

One of the challenges facing monitor processing comes from the transient nature of the memory it needs to read. Busy systems can find applications constantly starting up or terminating. This results in application memory being allocated and freed. Moreover, during the processing of a snapshot, agents may be in the process of either associating with applications or disassociating from applications. At the start of snapshot processing, an agent may be working on behalf of one application but by the end of the snapshot processing, it may be working on some other application.

To protect the ACID properties of a monitor operation and in particular the consistency and isolation of the monitor action, monitor processing must lock resources. Resources are locked before that resource can be accessed, and the lock protects the integrity of the monitor data and ensures that memory does not “disappear” while being read (resulting in crashes). Locking of a resource involves acquiring a “latch”, which is an internal mechanism for controlling concurrent events and the use of shared system resources. The protocol surrounding the locking of resources is strict, and resources must be locked and unlocked in a certain order and monitor processing must adhere to such protocols. If resource locking protocols are broken, the system can “hang”, so extra care must be taken to ensure that protocols are followed.

This requirement for ordering of events and the strict requirements of the protocols means that timing problems are more probable. It also results in a requirement to test these protocols by concurrently to increase the probability of performing locking and unlocking events out of sequence. The FIT method, which samples several event combinations and introduces resource constraints just as in a locking situation iterates through such probabilistic populations of events.

Multi-partition instances in the Data Partition Feature (DPF) environment present monitor processing with

other challenges. The activating or deactivating of event monitors requires the coordination of activity across all partitions. This involves sending messages to all partitions, waiting for all the partitions to perform the activation or deactivation and respond with success or failure, and coordinating the replies. In addition, global snapshot processing requires similar coordination. Messages must be sent to all partitions, snapshots executed locally on each partition with output sent in replies to the messages, and then replies merged into a single snapshot output stream. This processing must prove resilient to partitions activating and deactivating and dropped messages. Additionally, extra care must be taken to ensure that resources are not locked on one partition while messages are sent to other partitions. Failure to ensure the coordination of the deactivation and activation, as well as locking and messaging transmission, may result in severe software defects. In a threaded environment, the iterative approach in FIT creates a sample of situations where such interleaving and order can be disturbed.

3.2 Fast Communication Manager (FCM)

In order to achieve high performance and scalability, DB2 software provides two operating modes for parallel execution of activities. (Both rely on the availability of multiple CPUs for processing.) One of these configuration types is intra-query parallelism or SMP (Symmetric Multi-Processor configuration).

SMP works by generating SQL execution plans whereby portions of an SQL statement are divided into individual sections, which can be executed concurrently and independently by multiple processes/threads. The second type of configuration is Data Partitioning Feature (DPF). This configuration allows for the partitioning of data across multiple DB2 nodes. Each DB2 node is responsible for managing one data partition. The architecture where each DB2 node and its associated data partition are independent from other partitions is commonly referred to as “shared nothing”. It permits function shipping whereby SQL and non-SQL operations are directed to those nodes where the target data is held for local operation. Where multiple data partitions are involved, parallel processing occurs (with each DB2 node only working with its subset of data). SMP and DPF can also be combined within the same instance of DB2 software. Both these configurations require fast and efficient internal communication facilities.

A user may configure a DB2 instance with multiple nodes residing on the same host machine. Such configurations are described as Multiple Logical Nodes (MLNs). In such configurations, communication between DB2 nodes residing on the same host occurs through shared memory.

3.2.1 Fast Communication Manager Design

The fast communication manager component includes FCM resources, FCM receiver and sender conduits, connection management and node failure support. **FCM resources** are allocated from a separate shared memory segment that is allocated at start-up time by DB2. The two main FCM resources are buffers – which store communication data - and channels - which are the terminal points in communications. Each node on a DPF instance will have at least one **FCM receiver conduit** for incoming messages and one **FCM sender conduit** for outgoing messages. Connections are established on demand with **connection management**. The first indication of user activity on a node drives FCM to initiate communication with every other node configured in the instance. This ensures optimal performance and security of inter-node communication. **Node failure support** involves interrupting applications with dependencies on nodes that have lost their connection to the system and cannot be contacted. The FCM node failure-recovery facility allows applications to process node failures asynchronously from each other.

3.2.2 Opportunities for FIT

There are aspects of shared memory, failure recovery, connection management and conduit management that may create opportunities for challenges due to unusual circumstances such as frequent system interruptions,

frequent reconnections and unusual resource deprivation. An example of this is monitor running with FCM. With FCM, the single shared resource pool is created on each host on the MLNs to facilitate communication between logical nodes. In a multi-process environment, the interleaving of events may compete for memory. FCM, with the conduits establishing their own connections and resulting connection management, is designed to handle this interleaving adequately. In the case of node failure, and other unexpected events however, the probabilities of unexpected and sometimes incorrect interleaving may be increased. In this case, FIT can deprive resources and increase the samples of code failure events with memory management and connection management choices made by FCM. This will increase the probability of finding TRDs in FCM.

3.3 Crash Recovery

The third important feature of DB2 software that has been found suitable for testing with the FIT approach is the crash recovery feature[4].

Since units of work on a database can be interrupted unexpectedly, if an interruption occurs before all of the transactions in the unit of work are completed and committed, the database is left in an unusable state. Crash recovery moves the database back to a consistent and usable state by rolling back incomplete transactions and completing committed transactions still in memory.

Transaction failures result from conditions that cause the database or the database manager to end abnormally. Partially completed units of work that have not been flushed to disk at the time of failure, leave the database in an inconsistent state. Following a transaction failure, the database must be recovered. Conditions that may result in transaction failure include a power failure on the machine, causing the database manager and the database partitions on it to end abnormally; a hardware failure such as memory corruption, or disk, CPU, or network failure; or a serious operating system error that causes the DB2 application to end abnormally.

3.3.1 Opportunities for FIT

The conditions mentioned above that lead to transaction failure can create vulnerabilities and timing issues that should be found in testing. Order dependency is important because logs of the events that ran earlier must be replayed either to roll back partial transactions or complete uncommitted transactions in memory. The same issues arise because of parallelism and the need to replay logs in correct sequence. FIT is able to exploit the sequencing vulnerabilities by sampling from a large number of execution sequence possibilities.

In addition, transactions are logged while they occur, whether or not the transactions are committed. Transactions go from the log buffer to log files (transactional logging) before any data is written from the buffer pools to the database structures. Challenges can occur again in a multi-threaded environment because of the interleaving of events and the need to separate a given sequence when a problem occurs. This is a standard protocol, but problems can only be revealed with mass repetition of such logging of parallel processes. FIT tools facilitate execution of a large number of iterations of runtime and recovery scenarios and hence increase the probability of finding defects that occur during a particular sequence of events.

4 Methodology

The FIT approach hinges on repetition and resource deprivation, and as a result, automation is vital. The methodology presented is useful to those with large multi-process, multi-threaded software testing concerns, with vast combinations of possible executions and resource constraints are likely to trigger problems.

The approach is run as a number of controlled iterations on any machine and operating system combination. The iterations proceed with the following steps.

Step 1: Run random concurrent database test suites in the background to stress the supporting hardware, operating system and database, while varying the configuration parameters of the database, the size of the database,

the operating system, and the nature of the test suite being monitored (for example with a workload that tests monitoring functionality such as snapshot, crash recovery functions, or the fast communication manager). Since the configuration parameters control features crucial to monitoring, FCM and crash recovery (such as memory distribution (including sorting and locking), parallelism, I/O optimization (asynchronous page readers and writers), many aspects of logging (file size, buffer size), and recovery), it creates circumstances that are well-suited for uncovering potential software errors.
Step 2: Deliberately crash the database server by issuing a kill signal to the operating system.
Step 3: Restart the database server and the database
Step 4: Check for data integrity problems, database crashes (traps), and database hangs.
Step 5: If any problem is found, then exit and notify the tester via electronic mail alert. Else repeat from Step 1.

Supporting tooling was created to run several parallel processes and vary parameters. The tools execute the algorithm for the approach above and stress the monitor heavily by using the command line processor interface in one tool and the DB2 application-programming interface (API) in another tool to invoke the snapshot and the event monitor functions. The tool allows the user to control the number of iterations and is available for the UNIX and Windows platforms.

Another supporting tool runs the algorithm with the crash recovery procedure. The crash recovery tool runs the algorithm with several thousand crash recovery iterations by crashing the DB2 instance on all partitions and then restarting the database. This tool allows the tester to specify the number of crashes and is written for both the UNIX and Windows platforms.

5 Results

After applying the FIT method, defect detection improved significantly, and therefore increased tester productivity. Automation was key to the approach since the FIT tools were executed in scenarios with multiple databases with concurrent test suites running for extended periods (for example overnight).

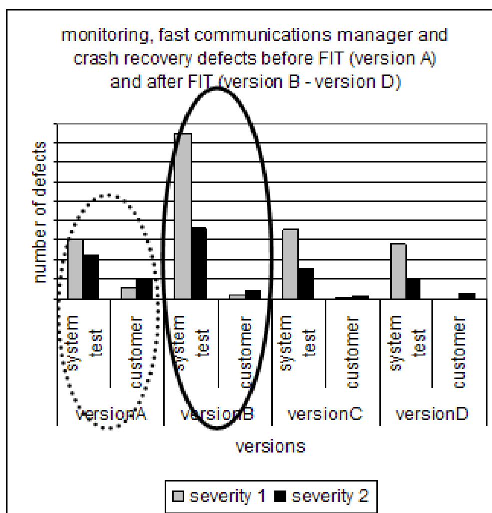

Figure 2: Defects for timing-dependent components found in system testing and by customers as an example of the defect discovery occurring between the introduction of FIT in version B and beyond

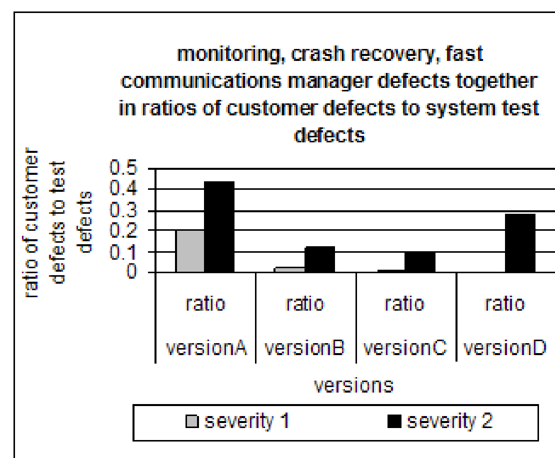

Figure 3: All defects for components most affected by timing issues as ratios between customer defects and test defects

Figure 2 shows the general trend for the three components in the DB2 product where FIT was used. The areas indicated by the solid circles (versus the dotted circles) show the increase in severity 1 and severity 2 defects found in system test versus by the customer after FIT in version B. Severity 1 defects cause the system to become unavailable, and severity 2 defects cause problems that hinder work but may be worked around. In addition, the ratio of customer defects versus system test defects decreases. That is, after FIT, testers find more defects and customers find fewer. Figure 3 shows the ratio between defects found in testing in the components before (version A) and after (version B and on) FIT; and the defects found by customers.

Since the tool repeats the same steps nondeterministically with each iteration, the tool increases the probability of hitting the same defect over time. Previously where TRDs occurred, the causes were difficult to record. This iterative method, using the same non-deterministic workload, increased the probability of hitting defects, therefore making it easier to reproduce the defects for debugging.

Additionally, the number of successful iterations before detecting a defect provided management with a quantitative and objective measure of the quality of the software. The number of iterations is independent of CPU speed. As a result, the number of successful iterations differs from the execution time for a workload, which depends on the CPU speed. This led to the formal requirement of a minimum number of successful iterations before the product could be shipped.

The approach was first applied to monitoring, and because of its success it was extended to crash recovery and fast communication manager. It is suitable to any areas of DB2 software where timing-related concerns may exist. These are areas involving communications between different nodes, data transfer between nodes and monitoring of nodes.

5.1 FIT Overhead

The mean number of iterations required to find a defect varies with time. The number is small at the beginning of the test cycle and grows towards the end. The value of the number depends on the component under test. The ideal value is infinity, where no defects are found and every iteration is successful. However, in reality, the principle of "good enough" reliability discussed in software reliability engineering is used, and test management decides on a particular target value for exiting the test cycle. The higher this target value, the higher the reliability of the component is deemed.

There are two types of overhead for this technique: (a) extra tooling effort in automating the FIT approach by making a FIT tool and (b) extra computational effort of running a FIT tool. For (a), the extra effort is mostly a one-time effort at the beginning of the test cycle. However, this tooling effort is small compared to the effort spent on the entire test cycle. For (b), the extra effort is negligible since most of the time of the FIT tool is spent running the test and only a small amount is spent preparing for each new iteration.

6 Conclusions

The FIT approach has many benefits. They included an increase in the number of timing-related defectTRDs found in system testing as opposed to such discoveries by customers and a new objective measurement for the quality of the DB2 timing-affected components that is independent of the platform on which the software runs.

One of the side effects of the quantitative measure of quality is the ability to determine the mean number of iterations to failure for components tested with the FIT approach. For crash recovery functionality, for example, this number has improved over tenfold since this method was employed and the measurement taken.

The first major lesson is that the automation created to support FIT is crucial because the large state space has meant that the ease in executing the algorithm has had many unexpected but welcome effects. For example, FIT has been able to identify stability regression defects in real time during the development cycle, that is, whilst testing for build-to-build regressions. More specifically, during continuous crash recovery testing, when testing

from one build to the next, there are sometimes regressions in both runtime and crash recovery testing. These are all rooted in recently integrated code that is intended to correct a defect or add new functionality. Because of the existing automation and the ease of implementing the method, these build-to-build regressions were easily discovered.

There was also a significant return on the initial investment to create the tools, since with the tools several workloads could be run simultaneously and easily, and left to sample the execution state space for crash recovery, monitoring, and fast communication manager. The tools would easily find new defects while running over many days. Instead of requiring the previous tester time to explore the state space, the tools are left fishing for defects on their own. This is inexpensive.

"Build it and they will come" – this quote does not apply to testing tools. Complicated tools, however useful, are left to gather dust. One of the other important points for FIT beyond automation was the ease of use of its automation. If setup of tooling is complicated and running the tool is complicated, then it will likely be used sparsely in testing. The FIT tools were carefully crafted to avoid such difficulty and have been intensely employed to validate the product. This has meant that many more defects have been found. The tooling has also been built so that one tester can easily set it running on many machines. Moreover, the tool alerts the tester when a defect is found; hence the tester does not have to monitor the test systems continuously

The underpinning factor has been the feasibility of this approach to test automation. This has resulted in returns that far exceed the investment. The future work with this approach will be in extending it to additional areas of the DB2 product.

References

[1] H. Bati and L. Giakoumakis and S. Herbert and A. Surna. A genetic approach for random testing of database systems. In *Proceedings of the 33rd International Conference on Very Large Data Bases*,pages 1243-1241,Vienna Austria, September 2007. VLDB Endowment

[2] J. W. Duran and S. C. Ntafos. An Evaluation of Random Testing. *IEEE Transactions on Software Engineering*,SE-1-(10):438-443. July 1984.

[3] R. Hamlet. *Random Testing*. Wiley. 1994.

[4] DB2 for Linux, UNIX, and Windows . http://publib.boulder.ibm.com/infocenter/db2luw/v9. IBM Press, Current February 2008.

[5] Sen K. Effective random testing of concurrent programs. In *Proceedings of the 22nd IEEE/ACM international Conference on Automated Software Engineering*, pages 323-332, Atlanta, Georgia, USA, November 2007. ACM New York

[6] E. Kit. *Software Testing in the Real World*. Addison-Wesley Professional. 1995

[7] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multi-threaded programs. In *Proceedings Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*, pages 446 - 455. San Diego, California, USA, June 2007. ACM, New York, NY.

[8] W. Visser and K. Havelund and G. Brat and S. Park. Model checking programs. In *Proceedings of the 15th International Conference on Automated Software Engineering*. IEEE Computer Science Press. September 2000.

Trademarks

IBM and DB2 are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both.
Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.
Windows is a trademark of Microsoft Corporation in the United States, other countries, or both.
UNIX is a registered trademark of The Open Group in the United States and other countries.
Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.
Other company, product, or service names may be trademarks or service marks of others.