Role of Automation in Computer-based Systems

Moinuddin Qadir
Rutgers, The State University of New Jersey
qadir@eden.rutgers.edu

Abstract

"Manual Vs Automatic" has been a controversial topic for ages in Social Sciences and recently (a couple of decades) in Computer Science. This paper argues that, where applicable, automation should be adopted as a prime strategy where human beings are likely to make more mistakes as compared to machines. Paper describes how most of the problems observed in computer-based systems are due to human mistakes at various stages of system development life cycle (analysis, design, implementation, testing, deployment and operations) and then describes the role of automation in reducing such problems, where applicable.

1. Introduction

There is no doubt that human intervention cannot be completely eliminated from computer-based systems because human intelligence is required at every level of any system development life cycle. Our purpose in this paper is to highlight the places where automation can help during various phases of system development in order to reduce problems in the finished product.

We feel that the term "Automation" is slightly misunderstood in that it's mostly taken as referring to "Automatic Configuration " where software scripts and programs replace operator actions to configure a system. In this paper, we'd like to emphasize on "Automation" as a broad term in many different forms to deviate from this miss-conception. Some of the techniques we present here may not look like "automation" in conventional sense but a closer examination will reveal otherwise. For example a really impressive piece of work [1: “Understanding and Dealing with Operator Mistakes”] proposes a framework in which new changes are introduced into the production system only after verifying them in a validation environment, setup parallel to the production environment, against live request load. That is nothing but "Automated Change Validation" to automatically validate operator actions against huge number of real requests, which otherwise would not have been possible by human testers.

It's clear that computer-based systems are bound to have problems (minor, major, critical, fatal) and failures no matter how carefully they are built. Previous studies have shown that major sources of system problems are "operations" and "software" [1, 2, 3, 5]. A closer examination reveals that "operations" related problems are mostly due to operator mistakes (configuration, diagnosis), and "software" related problems are mostly due to mistakes made by developers (in implementation and unit testing), designers (in design), testers (in feature testing and load testing) and analysts (in analysis and requirements specification). Thus most of the system problems can be attributed to human errors in carrying out well understood, well defined procedures during various phases of system development life cycle.

It can be observed that human beings are better at unconventional, non-mechanical and otherwise innovative tasks but once an idea has been well understood to the point where it can be translated into a list of mechanical steps, it can be performed more efficiently by machines. Human beings are bound to make mistakes while carrying out a long list of boring, iterative, mechanical steps over and over again, or for example, sitting for hours and comparing thousands of request/response pairs, or staring at multiple terminals displaying hundreds of performance statistics hopping to catch an abnormality, while machines are very good at such tasks. For example how efficiently can we calculate 2.7^5.4 x 4.5^3.1 by hand? And what are the chances of us making a mistake in this task as compared to a simple calculator? Moral of the story - "Never send a man for a machine's job".

1.1. Assumptions

For illustration purpose, throughout the paper, we are assuming a general "System Development Life Cycle" (referred to as SDLC from this point on) having analysis, design, implementation, testing, deployment and operations phases. In practice various specific SDLC methodologies may be used (for example Fountain, Spiral and Rapid Application Development [4]) but that really doesn't have any significant effect on our discussion here since in general all systems go through most of these phases in one order or the other [4], and thus problem sources and proposed automation techniques described here remain essentially the same.

Throughout the paper we are assuming separate roles for analyst, designer, developer, tester and operator. In practice one or more persons could be responsible for more than one such roles but that really doesn't have any impact on our discussion here.

The word "problem" in this paper is intended to include all (minor, major, critical, fatal) problems and failures in a computer-based system in the production environment.

1.2. Paper Organization

Rest of the paper has following organization: Section 2 surveys existing work on automation. Section 3 lists typical sources of problems in computer-based systems. Section 4 proposes some automation techniques to reduce the problem sources mentioned in section 3. Section 5 suggests future direction. Section 6 tries to respond to common objections against automation. Section 7 concludes the paper.

2. Existing Work

[1: “Understanding and Dealing with Operator Mistakes”] Describes a framework to validate operator actions in a proxy replica system before exposing the changes to the end users. Apparently this work seems to contradict the automation approach but a closer look reveals that indeed it is supporting automation approach in a different way by proposing to automate the validation task using real requests. Although this framework identifies a novel approach to validate operator actions in order to avoid field problems, it requires the development of application specific and often complex comparator functions to compare each combination of request/response pair, which seems like a lot of work for developers having its own problems. The validation environment itself requires complex setup, shunting the production environment and thus risking operator mistakes.

3. Problem Sources

In this section we discuss typical sources of problems in computer-based systems due to deficiencies at various stages of system development life cycle to link their occurrences with human intervention. Clearly it's impossible to enumerate all possible sources of problems here; therefore we’d rather identify the deficiencies at each stage that actually cause such problems in the field.

3.1. Analysis

Analysis is the foundation for any system, as the rests of the system building activities completely rely on its outcome - generally a "System Requirements Specification" document (referred to as SRS from this point on). In our experience, many field problems can be traced back to deficiencies at the analysis stage. Following are some typical deficiencies during system analysis phase that could cause problems in the production:

3.1.1. Improper Analysis

This happens mostly due to miss-communication between end-users and system analysts. It could also happen when end-users are not sure about the requirements and would rather rely on the analyst to come up with suggestions. In either case it's the analysts’ job to effectively communicate with end-users, research existing systems and industry standards, and come up with valid SRS to be passed on to system designers.

3.1.2. Inconsistent or Ambiguous SRS

The resulting SRS could be inconsistent or ambiguous to be interpreted by system designers. This generally happens due to requirements specification in a natural language (for example English), which could potentially be interpreted differently by different people.

3.1.3. Improper Change Propagation

Unfortunately SRS keeps changing throughout SDLC due to incremental analysis and increasing understanding of requirements with time. This makes it more like a moving target for system designers to keep up with. Even an apparently simple change in specification could result in major changes in design and implementation. Many problems arise in production due to improper change propagation among various stages of SLDC. This happens mostly due to miss-communication among system analysts, designers and developers. It is also difficult for the analysts to ensure the integrity and consistency of the SRS after a change in the requirements.

3.2. Design

Design translates SRS into schematic representation of system architecture and behavior from a technical standpoint, generally resulting in a design document that is passed over to developers for implementation. An incorrect design results in an implementation that deviates from actual requirements, and if not caught during testing, causes problems in the field. Following are some typical deficiencies in the design phase that could subsequently translate into field problems.

3.2.1. Incomplete or Obsolete Design

Designers may overlook some of the details in SRS or may not be able to keep up with frequently changing SRS resulting in an incomplete or obsolete design that does not really reflect the real requirements.

3.2.2. Inconsistent or Ambiguous Design

A design itself may have internal inconsistencies (potentially carried over from inconsistencies in the SRS). For example a person may be allowed to hold dual citizenship at the application level but there is only one “country” field in the database against each person.

3.2.3. Improper Approximations

Certain important modeling techniques (like Petri Nets, Stochastic and Markovian Modeling) are often intractable for human beings due to their complexity and combinatorial state-space explosion [7, 8]. In such cases designers use certain heuristics to cut the complexity and make the design manageable. Such approximations may not be valid in the actual system and could ultimately reflect as problems in the finished product.

3.3. Implementation

Following are some typical deficiencies in the implementation phase that could subsequently cause problems in the field.

3.3.1. Improper Resource Usage

Resource miss-use is a common issue among developers, no matter what programming language they use. Every language allows to allocate memory, open sockets, open files, start threads left and right without bounds. This could slip through peer code reviews and limited testing but definitely blows up in the production environment. Manually going through thousands (even more in some cases) of lines of code, and keeping track of each and every resource usage is exponentially hard if not impossible for human code reviewers.

3.4. Testing

Any real world scenarios slipped though the testing phase could cause arbitrary problems in the field that were never reported in the test environment. Following are some typical deficiencies in the testing phase that could subsequently cause problems in the field.

3.4.1. Incomplete, Inconsistent or Obsolete Test Plan

Test plans manually prepared from SRS are often subject to mistakes, overlooks, misinterpretations and inconsistencies. They can easily get obsolete because it’s difficult to manually keep up with frequent changes in the SRS. As a result the finished system is never tested for certain requirements.

3.4.2. Improper Feature Testing

Testers may not be able to test large number of combinatorial possibilities associated with certain features due to human limitations. For example a user interface takes 25 items of information from user and passes over to another component for processing. It is possible that certain invalid combinations of such information were not handled properly in the code. But such scenarios could slip through the test due to large number of possible combinations involved.

3.4.3. Improper Load Testing

In our experience, very few systems are load tested until they start having load related problems in the field. Load is one of the major sources of problems in large-scale computer-based systems. That usually happens due to over-looked deficiencies at implementation, design or analysis stages. It’s important to take load testing as an essential part of over all system testing, especially in load intensive applications.

3.5. Deployment

Following are some typical deficiencies in the deployment phase that could subsequently cause problems in the field.

3.5.1. Improper Documentation

One important reason for operator mistakes is “improper documentation” (User Manual, Installation Guide etc). User documentation is often prepared based on introspection and not from general user perspective [11]. They tend to cover only simple operating scenarios with straightforward configurations. Rests of the complications are left for operators to deal with.

3.5.2. Improper Deployment

Of course, development environment is different from that of testing and testing environment is different from that of production. These differences could range from topology to number of machines to IP addresses and so on. While feature and load testing could got well, it’s still possible to jeopardize the system in the field during deployment. This could get more complicated in the absence of a rollback plan to handle potential failures.

3.6. Operations

Previous studies have shown that a major source of problems in computer-based systems is: actions performed by operators [1, 2, 3, 5]. Operators are human beings and thus are subject to make mistakes, and unintentionally repeat them over and over again. Following is a list of typical deficiencies in the “operations” phase:

3.6.1. Improper Operator Training

No matter how experienced operators are, a new system going into production is most likely new to them as well. Training materials and classrooms don’t really interest operators (just like other human beings) due to general material coverage, which may be too naive for experts and too complex for novice operators [9].

3.6.2. Improper Configuration

Configuration errors constitute a large number of failure cases in computer-based systems. Configurations could go wrong, even in the presence of well-defined procedures and lists of steps. Operators at every level (novice, intermediate, expert) make mistakes in even simple tasks [1, 2, 3, 5,].

3.6.3. Improper Troubleshooting

Once a problem happens, usually operators are the first ones to know about it (after end-users). First they must have to identify the problem cause in order to fix it. Some visualization tools (like throughput, topology, user operations and CPU monitors) help to identify potential causes of the problem but they have to be manually scanned to discover and correlate abnormalities among components.

4. How Could Automation Help?

In this section we argue that automation does not only refer to "automatic configuration" but can be used effectively in every stage of system development life cycle to verify correctness, catch inconsistencies and monitor abnormalities in the system. We describe how could automation help in identifying and fixing the problem sources mentioned in section 3.

4.1. Machine Readable Specification

Recent work has shown that systems can be specified in terms of machine-readable (and possibly understandable) formats [12, 13]. Such a representation has following advantages:

1) It can automatically identify inconsistencies and ambiguities in the specification. This addresses "Inconsistent or Ambiguous Requirements Specification" [3.1.2].
2) It can automatically ensure consistency of the new document after changes, propagate such changes to the dependent documents (such as a test plan), notify related designers and keep track of change acknowledgements. This addresses "Improper Change Propagation" [3.1.3].
3) It can automatically be compiled into other useful (and machine-readable) documents (such as a Test Plan). This addresses "Incomplete, Inconsistent or Obsolete Test Plan " [3.4.1].

The main problem with this idea is: analysts are not necessarily familiar with complex machine-readable formats. A brilliant piece of work [12: Controlled Natural Language Can Replace First-Order Logic] attempts to address this problem with a subset of English language that can be used by analysts to specify system requirements. The proposed framework then internally converts such specifications into formal logic expressions in order to check for consistency and validity.

4.2. Machine Readable Design

Various CASE tools (for example Rational Rose, Auto CAD) are available to specify schematic designs for different components (hardware, software, database) of the system in machine-readable formats. Such a representation has following advantages:

1) It can automatically be checked for correctness and consistency. [12] Proposes such a framework for database design. This addresses "Inconsistent or Ambiguous Design" [3.2.2].
2) It can automatically generate robust code for well-known design patterns (such as singleton, queue, parser etc). Implementing such well understood design patterns from scratch would be like reinventing the wheel with the potential cost of testing and bug-fixing while a robust, well-tested code generator is readily available. Commonly available examples of such code generators are "Rational Rose" and “Borland Core::Architect”. This addresses "Improper Resource Usage" [3.3.1].

4.3. Automatic Code Verification

Commonly available resource profiling tools (like CodeComplete, Purify, JProbe, AppPerfect etc) have proven to detect resource leaks in even complex and large code bases. This addresses "Improper Resource Usage" [3.3.1].

4.4. Automatic Feature Testing

Such automation is not limited to generating a test plan from specification but can also help carry out pre-defined sets of tests over and over again. “Win Runner”, “AdventNet QEngine” and “Borland Core::Tester” are some examples of automatic testing tools where a test scenario can be recorded once and then replayed for any number of times, with any number of value combinations, in any order with other tests. If no such tools fit the requirements of the application in question then we recommend building customized tool(s) for this purpose. In our experience it's worth spending resources on such tools as they can be reused throughout the life of the application and for a bunch of other similar applications. Such automation reduces the chances of missing feature testing and thus reduces field problems. This addresses "Improper Feature Testing" [3.4.2].

4.5. Automatic Load Testing

“Load Runner” and “AdventNet QEngine” are some examples of commercially available tools that help in this regard. If no such tools fit the requirements of the application in question then we recommend building customized tool(s) for this purpose. In our experience it's worth spending resources on such tools as they can be reused throughout the life of the application (in every release for example) and for a bunch of other similar applications. In one industry application, we had to write special simulators after realizing that our testing environment was nowhere close to that of customer in terms of load. That actually helped solve a lot of problems in the system before rolling out the next release. This would not have been possible without creating a load test environment in house and the system would have be thrown out of customer premises for ever. This addresses "Improper Load Testing" [3.4.3].

4.6. Automatic Action Tracking

Keeping track of operator actions can have following advantages:

1) System could suggest things like "shutting off this component will reduce current throughput from x to y" rather than simply displaying “are you sure” type of messages.
2) Such logs could be used to analyze and fix problems caused by operator actions.
3) Such logs could be used to reduce operation over heads and futz – an active area of research [10].
4) Such logs can be used to model human actions in certain environments [9, 10].

4.7. Automatic Problem Discovery

Instead of having a bunch of operators sit in front of multiple terminals, monitoring various performance statistics (like throughput, CPU usage, memory usage, network etc), we can have tools to automatically do that for us with certain threshold bounds (or some other assertions) for every statistic on each component. Once a statistic goes out of the normal range an operator (or a recovery process) can be notified by the monitoring tool. Of course some complex scenarios could be detected as false positives or false negatives, which would have been processed more intelligently by human beings but on the other hand human beings would have missed some obvious cases too. This addresses “Improper Troubleshooting” [3.6.3].

[14: Writing Assertion Programs Using The A Language and Run-time] Proposes a framework for run-time, model-based verification using assertion agents in "A" language. Such external agents can be programmed to monitor multiple components of a system for various assertions based on the “correctness” definition of individual components in the system and their combinations.

4.8. Automatic Recovery

This is an active area of research. The idea is to automatically detect problems and take corrective actions to keep things going while leaving the cause of the problem to be investigated and fixed latter. This kind of recovery is common in large-scale systems dating back to Tandem Systems [2, 3] and recently in large-scale Internet services, such as Google [15]. This addresses “Improper Troubleshooting” [3.6.3].

“Symantec VERITAS” is a commercially available tool based on active replication and external component monitoring without changing the underlying application.

Another effort in this direction is [16: Software Rejuvenation] that proposes a cost effective strategy to periodically restart system components in order to avoid potential problems in the future.

4.9. Intelligent Operator Training

[9] Proposes “feedback based” automated training systems to provide intuitive learning for operators in personalized environment targeting to match individual operator’s areas of improvement. This is an active area of research. This addresses “Improper Operator Training” [3.6.1].

Note that above described automation techniques do not help in the "Improper Analysis" [3.1.1], "Incomplete or Obsolete Design" [3.2.1] and " Improper Documentation" [3.5.1] cases, keeping them exclusively as human jobs - requiring effective communication, intelligent research and innovative insight into existing standards, none of which are a machine's characteristics (at least for now). Moral of the story - "Never send a machine for a man's job".

5. Future Direction

Above described techniques are just an introduction to a whole lot of automation techniques already developed and being researched in this field. Much more work needs to be done in this direction in order to realize computer-based systems with availability comparable to (for example) telephone systems today. Almost all of the automation techniques described above is active area of present and future research.

6. Objections Against Automation

In this section we examine some major objections against automation along with our response.

1) Any automation would also have to be designed and implemented by human beings, thus having their own problems, then how could automation help reduce problems in the target system? In other words automation has to be fault free in order to make the target system fault free - is that possible?

Our Response: That's a very good objection but we have a very simple counter argument - automation will definitely have it's own faults but they can be fixed with time while operators, testers and developers can not be fixed, they will continue to be faulty, even with proper training, procedures and experience!!! [2, 3].

2) Such automation would make Operators totally unaware of the internals of the system and they won't be able to fix anything when automation itself fails. This is also referred to as "Automation Irony".

Our Response: How about not providing operators any scripts at all and train them on the internals of each and every component, or let them figure out the correct configuration by themselves, or better yet just let them develop the system and set it up in the field on their own (BTW that's the model many companies seem to follow now a days - keep moving people from development to support in order to minimize MTTR). In our opinion that's not the way system development should work. Systems should be layered in a way that one layer need not worry about the internals of any other layers. Operators and developers are like two different, independent layers in a system, carrying out their respective jobs to the best of their abilities.

3) Extensive automation will result in work force reduction.

Our Response: Sigh!!!!! Unfortunately that's true for almost all advancements. In fact (ironically) that's a simple gauge to measure how successful your innovation is. On the other hand, a closer look will reveal that in fact that's what makes societies shift from every day mechanical work to highly intellectual innovations. You have to move on.

7. Conclusion

Automation can significantly reduce problems in computer-based systems, especially in lengthy, repetitive, boring, combinatorial and non-innovative tasks where human beings are likely to make more mistakes as compared to machines. Automation not only carries out long lists of mechanical steps efficiently but can also help verify the correctness of system components before they go into field, monitor components' health at run time, identify potential problems before they actually occur and analyze already occurred problems to identify the root cause. In other words, minimizing human intervention by introducing automation (where applicable) should result in significant decrease in system problems.

8. References

[1] Kiran Nagaraja, Fabio Oliveira, Ricardo Bianchini, Richard P. Martin, Thu D. Nguyen: Understanding and Dealing with Operator Mistakes in Internet Services

[2] Jim Gray: Why Do Computers Stop and What Can Be Done About It?

[3] Jim Gray: A Census of Tandem System Availability Between 1985 and 1990

[4] Russell Kay: Quick Study, System Development Life Cycle, http://www.computerworld.com/developmenttopics/development/story/0,10801,71151,00.html

[5] David Openheimer, Archana Ganapathi, David A. Patterson: Why do Internet Services fail, and what can be done about it?

[6] Kiran Nagaraja, Xiaoyan Li, Ricardo Bianchini, Richard P. Martin, Thu D. Nguyen: Using Fault Injection and Modeling to Evaluate the Performability of Cluster-Based Services

[7] John F. Meyer, William H. Sanders: Specification and Construction of Performability Models

[8] Oliver C. Ibe, Richard C. Howe, Kishor S. Trivedi: Approximate Availability Analysis of VAX Cluster Systems

[9] Christine M. Mitchell: Task-Analytic Models of Human Operators: Designing Operator-Machine Interaction

[10] David A. Holland, William Josephson, Kostas Magoutis, Margo I. Seltzer, Christopher A. Stein: Research Issues in No-Futz Computing

[11] Michael J. Albers: The Key for Effective Documentation: Answer the User’s Real Question

[12] Norbert E. Fuchs, Uta Shwertel, Sunna Torge: Controlled Natural Language Can Replace First-Order Logic

[13] Nazareno Aguirre, Tom Maibaum: A Temporal Logic Approach to Component-Based System Specification and Reasoning

[14] Andrew Tjang: Writing Assertion Programs Using The A Language and Run-time

[15] Sanjay Ghemawat, Howard Gobioff and Shun-Tak Leung: The Google File System

[16] Nick Kolettis, N. Dudely Fulton: Software Rejuvenation: Analysis, Module and Applications