Test and verification approaches in conformance checking

KEVIN JAHNS

English Communication for Engineers

January 27, 2014

Introduction

Testing is obligatory for good software development, and not just because it eases bug finding: Since we use technology everyday (e.g. airplanes and traffic circulation) it is evident that software bugs can cost lifes. Furthermore testing is important for the economy: In 2002 a study stated that software errors cost the U.S. economy $59.5 billion US-dollars annually, whereat most of this cost could be avoided by more exhaustive testing [1]. A recent study which was initiated by the Cambridge University states that software bugs cost the overall economy $312 billion US-dollars because debugging is inefficient [2]. Many software companies still use trivial testing approaches like *Monkey Testing*. More enhanced testing approaches could safe costs and in addition make the software better. Moreover there are approaches

to *verify* that a software does not malfunction. However, the crux of testing and verifying is always expressing *conformance*.

Testing

Until now most software developers write test cases for software by hand or just execute the program by themselves. They write test in the form of executable code. Obviously this is not the state of the art, but it is very easy. The technical term for this is *Monkey Testing*.

Monkey Testing

There are many forms of monkey testing. The crucial point of monkey testing ist that there is no definition of tests to execute. Developer and possibly testing employee execute the software product till they find a bug.

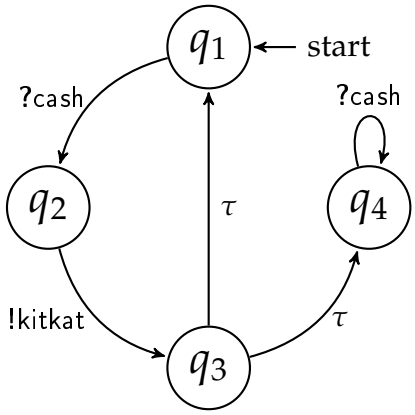
An interesting point is that Monkey Testing approach originates from an actual formal theorem: The *Infinite Monkey Theorem* [3]. Assumed we have infinitly many monkeys making random keystrokes on a typewriter machine. Then the Infinit Monkey Theorem states that almost certainly one of the monkeys writes the complete works of William Shakespeare. The expression “almost certainly” characterizes testing very good since we cannot be certain that testing finds all bugs in our software. This is due to the restriction of the size of a test suite (it cannot be of infinit size).

The downside of this approach is that there is no formal definition of conformance. If a program conforms relies on the intuition of the tester.

Model Based Testing

In Model Based Testing (MBT) we express conformance via an *Labeled Transition System* (LTS). Because transition systems have a theoretical background, information scientists are familiar with them. But also computer scientists need a lot of knowledge to express conformance in the form of an LTS. In figure 1 we express a very abstract specification of what a candy machine can do and cannot do in the form of an LTS.

Figure 1: Candy machine specification



1. A candy machine starts in state q_1
2. After the candy machine gets cash it must output a kitkat and go into state q_2
3. Then it may either go back into state q_1 or it goes to state q_4
4. In state q_4 it must not output a kitkat and only accept cash.

In MBT we can derive test cases automatically from the LTS. That is why MBT is one of the most advanced testing approaches nowadays. We can also exchange the software product since our test cases do not rely on it, and test a bad behaviour (e.g. “do not explode”) on several products.

Model Checking

In contrast to testing Model Checking (MC) *verifies* if a program conforms. This means that we are not only “almost certain” that a program conform - we are “absolutely certain” that a program conforms. Thus, if we test a very critical programm, we can be certain that it does not malfunction. Unfortunately this approach is not only very hard to implement, MC is sometimes impossible to compute. In theory the model checking of a program could take much more computation steps than there are atoms in the universe. Pioneering work was done by E. M. Clarke for what he received the Turing Award. A keywork of his work was defined checkable properties:

1. Statements like “Never do this” are called *persistent properties*
2. Statements like “After doing this you may do that never again” are called *liveness properties*

It may be unclear that this is exactly the behaviour that we want to check, but it can be shown that we can express every characterization of

a program and therefore making MC the ultimate tool for conformance checking. However MC does still need a quite long time to compute even simple programs.

Conclusion

In this essay we have shown the difference between testing and verifying. Conformance checking is an important topic for the industry and therefore we should think more about how to do it more efficient. On account of this we have shown that we can do much better than Monkey Testing.

References

1. Department ofg Commerce's National Institute of Standards and Technology (NIST). Software errors cost u.s. economy $59.5 billion annually, June 2002. URL: http://www.abeacha.com/NIST_press_release_bugs_cost.htm.

2. Cambridge University. Cambridge university study states software bugs cost economy $312 billion per year, 2013. URL: http://markets.financialcontent.com/stocks/news/read/23147130/Cambridge_University_Study_States_Software_Bugs_Cost_Economy_$312_Billion_Per_Year.

3. Infinite monkey effect.