Miscomputation in Software Development. Learning to Live with Errors

Tomas Petricek
University of Cambridge, UK

Anyone who experienced a software error would agree that a computer program may not work correctly, but the notion of *malfunction* relies on an external specification, other than the program code[1]. If a program does not match its specification, this can be due to a number of reasons that have been classified in a recent taxonomy of *miscomputations* [7].

Large software systems rarely have a full specification, but they exhibit numerous kinds of errors. In this paper, I discuss different practical strategies that programmers throughout the history developed to deal with errors. Although our focus is not on the historical developments, we note an interesting change from the early days of computing (circa 1949):

> *Errors in coding were only gradually recognized to be a significant problem: a typical early comment was that of Miller, who wrote that such errors, along with hardware faults, could be "expected, in time, to become infrequent"* [11, p. 254]

Since 1949, programmers not only learned that coding errors are a significant problem, but they also found many techniques for dealing with them, ranging from ways to eliminate errors to methodologies that accept errors as a part of any complex system and found ways of living with them.

[1] The difficulty of treating software as malfunctioning artifact has been discussed in [6].

Error as a curse: Avoiding errors through proofs

The idealistic approach to miscomputation is that program behaviour should be formally specified and all possibilities of error should be avoided. This has first been made explicit in the Algol research programme, appearing in 1960s:

> *One of the goals of the Algol research programme was to utilize the resources of logic to increase the confidence (…) in the correctness of a program. As McCarthy had put it, "[i]nstead of debugging a program, one should prove that it meets its specifications (…)"* [11, p. 258]

Proving program correctness is only possible with respect to a specification. A full specification is rarely available, but many programming languages allow specifying some aspects of the behaviour. In typed programming languages, *types* can be seen as a limited form of specification. A type system ensures that "well-typed programs do not go wrong" (a famous slogan introduced by Milner in 1978 [10]), meaning that properties expressed in

terms of types are guaranteed. Recent developments in this direction aim to increase the expressivity of type systems, so that more precise specifications can be given[2].

The nature of proofs about programs is a complex topic, but it has attracted some attention from historians and philosophers of science [9]. In contrast, other approaches for dealing with errors are less studied – and even advocates of the use of proofs are surprised that the software industry manages to produce working software without proofs [8].

[2] A recent example in this direction is [5].

Error as a progress: Test-driven development

Part of the success of software engineering can be attributed to testing. Testing cannot prove absence of errors, but it can provide a (practically useful) confirmation [1].

However, a recent software development methodology called *test-driven development* (TDD) uses the notion of tests in a different and perhaps more interesting way. In TDD, we write automated tests that detect certain errors. However, tests do not serve only as a mechanism for avoiding errors. TDD uses them (and miscomputation) as the driving force behind development. As discussed by Beck, we "*write new code only if an automated test has failed*".

We should first produce an isolated miscomputation and then write code to remove it. Thus in TDD, tests replaces the (non-existent) formal specification – they define program behaviour by specifying which miscomputations *do not occur* when the program is run. The methodology is described in terms of the Red-Green-Refactor mantra:

> [1] *Red – write a little test that doesn't work (…).*
> [2] *Green – Make the test work quickly, committing whatever sins necessary in the process.*
> [3] *Refactor – Eliminate all of the duplication (…).* [3]

In TDD, introducing a miscomputation deliberately becomes the first part of the development cycle. Dealing with the error (fixing the test) is a way to implement the specification; the third step is then a place where generalization from the specific tests occurs (as part of duplication elimination).

Error as the unavoidable: Let it crash

Although different, both approaches discussed so far aim to eliminate miscomputation from completed running programs. The concurrent language Erlang takes a different attitude characterised by the slogan "let it crash". In the Erlang mind-set, an error refers to a situation where we do not have enough information to proceed:

> *Errors occur when the programmer does not know what to do. Programmers are supposed to follow specifications, but often the specification does not say what to do and therefore the programmer does not know what to do.* [2]

The citation explicitly quotes the situation when a program specification is incomplete

and does not provide an explicit guidance. In that case, an ideal program would perform the "right" operation – but without a specification, a program *miscomputes*. In Erlang, this is expected and programmers have a strategy for dealing with such errors:

> *What kind of code must the programmer write when they find an error? The philosophy is let some other process fix the error, but what does this mean for their code? The answer is let it crash.*

Erlang has a sophisticated supervision model – a supervisor process typically restarts the worker process (and logs the details of the miscomputation), so that the worker can continue performing other valid operations.

Thus miscomputation in Erlang is mitigated by the system and is expected during regular program execution.

Error as an inspiration: Live coding

Miscomputation takes yet another form when we treat computation as an *interaction*. The first programming environment that used this style was Smalltalk in 1970s. More recent work includes live coding environments for performing music. Our metaphors for miscomputation change accordingly:

An error in the performance of classical music occurs when the performer plays a note that is not written on the page. In musical genres that are not notated so closely (…), there are no wrong notes – only notes that are more or less appropriate to the performance. [4]

A musical genre that is not closely notated corresponds to a program that does not have a precise specification. This is typical for many software engineering projects – there are some general guidelines, but no full specification. In programming, the idea appeared through *live coding* where programmers interact with the running system live through code:

> *[Live coders may] accept the results of an imperfect execution. [They] might perhaps compensate for an unexpected result by manual intervention (like a guitarist lifting his finger from a discordant note), or even accept the result as a serendipitous alternative to the original note.* [4]

It is easy to understand this approach in music, but that is just one of the domains. In Smalltalk, live coding can be used to interactively change a running system in response to errors. The interesting point is that making miscomputation apparent (we hear a dissonant note) enables live coder to quickly react and correct the behaviour. The reaction time appears to be an important aspect of live coding – the approach can work well in scenarios where timely human interaction is possible (typical web applications, but not e.g. an airbag control system in a car).

Conclusions

Perhaps the most interesting message from the paper is that miscomputation does not always have to be fully avoided. Avoiding miscomputation at all cost (through formal proofs or thorough testing) is the most common technique, but there are interesting alternatives that embrace miscomputation and accept it as an ordinary part of software development, software execution or even observable software behaviour.

References

[1] N. Angius, "The Problem of Justification of Empirical Hypotheses in Software Testing", *Philosophy & Technology* 27(3), 2014, pp. 423-439.

[2] J. Armstrong, *Making reliable distributed systems in the presence of software errors*. PhD dissertation, 2003.

[3] K. Beck, *Test driven development by example*. Addison Wesley, 2002.

[4] A. Blackwell, N. Collins, "The programming language as a musical instrument", *Proceedings of PPIG*, 2005.

[5] A. Chlipala, *Certified Programming with Dependent Types*. MIT Press, 2013.

[6] L. Floridi, N. Fresco, G. Primiero, "On malfunctioning software", *Synthese* 192(4), 2015, pp. 1199-1220.

[7] N. Fresco, G. Primiero, "Miscomputation", *Philosophy and Technology* 26, 2013, 253-272.

[8] C.A.R. Hoare, "How Did Software Get So Reliable Without Proof?", *Proceedings of FME*, 1996.

[9] D. MacKenzie, *Mechanizing Proof: Computing, Risk, and Trust*. MIT Press, 2004.

[10] R. Milner, "A theory of type polymorphism in programming", *Journal of computer and system sciences* 17(3), 1978, pp. 348-375.

[11] M. Priestley, *A Science of Operations: Machines, Logic and the Invention of Programming*, Springer, 2011.