Miscomputation in software development
Learning to live with errors

Tomas Petricek
University of Cambridge, United Kingdom
tomas.petricek@cl.cam.ac.uk

Lesson 1: Introduction

> *If trials of three or four simple cases have been made, and are found to agree with the results given by the engine, it is scarcely possible that there can be any error (...).*[1]

TEACHER: (Reads the quote from the blackboard.)

TEACHER: Welcome and thank you for joining the seminar on miscomputation in software. The opening quote from Charles Babbage about the Analytical Engine suggests that Babbage did not see errors as a big problem. If Babbage was right, the software industry would save billions of dollars, but sadly, eradicating all errors from our software is far from easy.

In retrospect, it is interesting to see how long it took early software engineers to realise that coding errors (that is, errors in translating mathematical algorithm to program code) are a problem. As Mark Priestly writes:

> *Errors in coding were only gradually recognized to be a significant problem: a typical early comment was that of Miller [circa 1949], who wrote that such errors, along with hardware faults, could be "expected, in time, to become infrequent".*[2]

We have been living with errors for over 50 years now and programmers found different strategies for dealing with them. Some errors are simple slips, like forgetting a semicolon. Logical errors are harder to find, but at least we know that something went wrong. For example, if our algorithm does not correctly sort certain lists. But there are also issues that are harder to classify. For example, if a machine learning algorithm incorrectly classifies certain images.[3]

In this seminar, we will try to identify and understand different kinds of errors and we will look at different strategies for avoiding them. To open the discussion, do you think we will ever be able to eradicate coding errors?

PUPIL TAU: Errors are a curse and must be avoided at all costs. Most software we use today contains errors and if we are to make software engineering a reputable discipline, it is our duty to find a way of avoiding errors. In fact, if your program contains an error, you should not even call it a program!

We should build on strong mathematical foundations and write provably correct software. I think that dependently typed programming languages are the way to go.

PUPIL BETA: I can see how this would work for the factorial function or for Fibonacci numbers, but real software systems are not so simple. When you start building a real system, you do not even have a full specification, more so a formal one. This is why we should turn functional requirements into tests. A well-written test specifies one aspect of a software system and if it can be automated, it guarantees correctness.

PUPIL ALPHA: I agree with Beta, but I would go even further. Tests should be the driving force behind the whole development process. You should only write code after you write a failing test that specifies one aspect of your system.

PUPIL TAU: I'm glad we agree that correct software should be the ultimate goal, but substituting proofs for tests is never going to be enough. You are just showing the absence of certain errors, not proving your program correct.[4]

Using such sloppy methods has serious consequences – we rely on software so much that an error can kill people! If we know that we can prove our software correct, anything else is unethical.

[1] Babbage (1837) quoted in Priestley (2011).

[2] Priestley (2011).

[3] For a recent example of such miscomputation see Barr (2015).

[4] TAU is paraphrasing Dijkstra's (1970) famous quote: "Program testing can be used to show the presence of bugs, but never to show their absence."

PUPIL EPSILON: Oh, come on! You both really think you can eliminate all errors? Take any complex distributed system as an example. So many things can go wrong – one of your machines runs out of memory, your server receives an unexpected request. Even a cosmic ray may flip a bit in your memory if your data centre is large enough! The path to dependable software is to learn to live with errors.

PUPIL TAU: I would like to object; we can accommodate for all of these situations in our proofs, but I see there is only little time left and I'm curious if OMEGA can direct our discussion back into a more sensible direction.

PUPIL OMEGA: Yo, I tell you, errors are fun!

TEACHER: That is an interesting position OMEGA, but I think the class needs more explanation. What *exactly* do you mean?

PUPIL OMEGA: Okay, then... I was playing in a club yesterday. I accidentally put on a wrong sample and it turned out to be much better! If you make an error, you might be surprised.

And if TAU thinks it is unethical, wouldn't it be just as unethical as to limit our method of discovery? Penicillin was discovered by the kind of accident that TAU wants to ban!

PUPIL BETA: Surely, we can find a reasonable middle ground. Errors during live coded performance are one thing, but you would not want to lose your money in a banking system because of an error. And I for one will never get into an airplane with software created by OMEGA!

TEACHER: Thank you all for turning the seminar into a lively discussion. We all seem to agree that we should be producing software that serves the purpose it was designed for, but there is much disagreement about how to achieve this and it is perhaps not even clear what the *purpose* is. I'm afraid our today's lesson is over, but we will continue next week, by first giving more space to TAU to explain his position.

Lesson 2: Errors as a curse

TEACHER: Let us start with TAU's position. I am all eager to learn how we can avoid errors altogether.

TAU: My position is that the only way to increase the confidence in correctness of programs is to utilize the resources of logic. Instead of debugging a program, one should prove that a program meets its specification.[5] Then there will be no doubt that our software is correct and serves its purpose.

EPSILON: I like reading papers about programming language theory and I do enjoy an elegant proof, but those always make unrealistic simplifications. For example, if you model "year" as a natural number in your proof, you will not be able to catch errors like the Y2K bug that was caused by an "implementation detail" that proofs tend to ignore. So I am not yet convinced that your proofs will guarantee full correctness.

TAU: Proving properties on paper is nice for small theoretical models, but for software engineering, we need to make proof an inherent part of the development process. One way to do this is to use types. A type system can show the absence of certain kinds of errors in your program.

EPSILON: But you still need to prove that the type system is correct. Otherwise you have no guarantees!

TAU: Of course, you do. But type systems are small and tractable mathematical structures and there are standard ways for proving that they are sound. You prove the progress and preservation property and you have a guarantee that well-typed programs do not go wrong![6]

BETA: At first I thought that we disagree, but now it seems that I was mistaken. I too believe that types are useful for avoiding certain kinds of errors. But there are properties that cannot be checked using types and that's when we need tests!

TAU: There is no need for testing. The Curry-Howard isomorphism lets us import even more resources of logic into programming through types. With dependently typed languages like Idris, Agda or Coq, you can express the full program specification just in terms of types. And then we will finally be able to write provably correct software!

TEACHER: I think we can be convinced that with a sufficiently powerful type system, you could provide a formal specification in the type system and prove that the program matches it. But if this is the right way to go, why are not there more programs written in such a way?

TAU: Today, most people who write software, practitioners and academics alike, assume that the costs of formal program

[5] Historically, this position first appeared with the Algol language. To quote Priestley (2011), p258: "One of the goals of the Algol research programme was to utilize the resources of logic to increase the confidence that it was possible to have in the correctness of a program. As McCarthy (1963) had put it, "[i]nstead of debugging a program, one should prove that it meets its specifications, and this proof should be checked by a computer program".

[6] TAU refers to the Semantic Soundness Theorem introduced by Milner (1978). Following Wright, Felleisen (1994), this has become a standard method in theoretical programming language research.

verification outweigh the benefits[7] and that fixing bugs later is cheaper than investing into correctness from the very start.

But I am convinced that this is just a matter of time. The technology of program verification and dependently typed languages are mature enough today that it already makes sense to use it in many kinds of critical projects.

TEACHER: Very good. But let me remind you that certification of correctness is just one of the roles of proofs in mathematics. Proofs aren't there to convince you that something is true. They are there to show why it is true.[8] And sometimes they also play the role that OMEGA attributed to errors – they lead into unexplored territory of mathematics where we may even find different fundamental questions. I wonder if the same is true about proofs in software development…

EPSILON: I had a look at some Coq projects and I am convinced that the proofs are true, but I doubt they fulfil the explanatory role. Reading a sequence of tactic invocations is definitely harder than reading an implementation of the same problem in a simple language like Clojure!

TAU: I agree that Coq proofs can be complex, but the proof details do not matter. The future is dependently typed languages that infer a correct implementation from the type.[9] So, you will only need to write a sufficiently detailed specification using types, possibly with helper functions or lemmas.

EPSILON: And the circle is closed! You are proposing to shift all the complexity of programming from writing a solution to writing a specification of the solution.

I do not see how this is safer. In today's languages, we have to verify the implementation and we invent new abstractions and constructs to make this easier. In your dependently typed future, we will have to verify the equally complex specifications and we will presumably be inventing new abstractions and constructs to make this easier!

TEACHER: We came to an interesting point here. Is it easier to write a correct concrete implementation or a correct abstract specification? But I'm afraid we will have to leave this question to psychology and sociology of programming.[10]

We will return to this topic later though. To wrap up today's lesson, let me propose the following summary: Proving correctness of programs is promising for systems that have a simple specification. For systems with complex specifications, the complexity outweighs the benefits of being able to prove that an implementation follows the specification.

Lesson 3: Errors as progress

TEACHER: In the last lesson, TAU persuasively argued that software needs mathematical proof in order to be reliable. Yet, we daily depend on software consisting of hundreds of millions of lines of code and the number of disasters caused by software is relatively small. How did software get so reliable without formal proofs?[11]

BETA: It is not so surprising that we have gone overboard for the theory at the expense of practice. After all, we find prejudices in favour of theory as far back as there is institutionalized science.[12]

In reality, computer programming is an engineering discipline like any other. Miscomputation can be avoided only by professionalization of software engineering and by following disciplined practice and producing work that meets highest professional standards possible.[13]

TAU: I do not see how this differs from my position. Meeting the highest professional standards *means* using the resources of logic to prove your programs correct.

BETA: The problem is that even formal verification is subject to human errors and budget constraints. In practice, you only verify that software is 99% correct[14] and you will always make simplifying assumptions to a certain degree.[15]

The way to make sure that those do not lead to errors is to adopt code of ethics like in other engineering disciplines. Such code of ethics will require a careful analysis, specification and design followed by professional development and a phase of thorough testing.[16]

[7] TAU is paraphrasing the introduction from Chlipala (2013)

[8] A good discussion on the role of proof in mathematics can be found in Gold and Simons (2008). The above is quoting Auslander (2008). For more information on mechanized proofs, see also MacKenzie (2004).

[9] This is a common view in the dependently-typed community, however there does not seem to be a canonical reference making this exact point. A good example is McBride, C., and McKinna (2004).

[10] This is almost a non-existent field that would deserve more attention. Some work in the area has been done by Mayerovich and Rabkin (2012).

[11] This very same question has been asked by a proponent of formal methods and the Algol research programme; Hoare (1996)

[12] This is a paraphrase of an observation made by Hacking (1983), p.150

[13] Historically, calls for the professionalization of programmers first appeared around the time of the NATO conference in 1968. Chapter 8 in Ensmenger (2010) provides the historical context.

[14] For example, see Flaisher et al. (2007) for a discussion of the formal verification of the Intel Core 2 DUO CPU.

[15] Manolios (2008) discusses issues of (typically ignored) quantum effects.

[16] Anderson et al. (1993) discuss the ACM Code of Ethics.

TAU: The focus on testing is naïve. As I said before, testing can show the presence of bugs, but never their absence![17]

TEACHER: Let me clarify. This way of seeing testing is akin to Popperian[18] view of scientific theories. A theory is scientific if it can be falsified, but we can never prove it true.

Even in science, not everybody agrees. If a scientific theory makes a correct prediction or passes a test that had the could have disproved it, the probability that it is correct increases.[19]

BETA: I get it now! It's like in the Bayes' theorem. There is a prior probability that the software is correct, possibly based on the team behind it. Testing the software then provides new evidence and the Bayes' theorem prescribes how probabilities are to be changed in light of this new evidence. [20]

Of course, running the exact same test twice does not improve the probability of correctness, but this is already accounted for by the Bayes' theorem. So what we need is a way of finding relevant tests that do increase the probability.

TEACHER: It seems that what we are looking for now is a methodology of software development. Does anybody want to propose one?

ALPHA: I like the idea of treating tests as specification, but as the Bayesian philosophy of BETA teaches us, we need to make sure to add relevant tests that increase the confidence that our software behaves correctly.

I propose that we should first write an automated failing test that defines a new, not yet implemented, functionality. Only then we can write new code.[21] By writing a failing test first and providing implementation later, we know that the test also increases our knowledge about the system.

BETA: This idea of writing tests before code seems a bit odd to me. How can you test code that does not exist? Surely, you do not want to spend months making up tests for application that does not even exist.

ALPHA: Quite the opposite! Test-driven development gives developers as rapid feedback as possible. Rather than writing a full specification in advance, you add features one by one as the requirements become clear. You proceed in a loop, following the Red-Green-Refactor mantra: [22]

1. Red – write a failing test that specifies the feature.
2. Green – make the test work quickly, committing whatever sins necessary in the process.
3. Refactor – eliminate all of the duplication introduced through the imperfect implementation in step 2.

TEACHER: I propose to call this the test-driven development methodology. In other words, TDD makes a miscomputation an inherent part of the development cycle. We first produce an isolated miscomputation and then write code to eliminate it and we use automated tests to ensure that it has been eliminated for good.

The fact that we treat tests as specification means that their successful run provides a Bayesian confirmation about the correctness of our software.

Lesson 4: Errors as the unavoidable

TEACHER: So far, we discussed two strategies of eliminating miscomputation from software, so let's move to the third one. EPSILON, what is your strategy for dealing with errors?

EPSILON: Eliminating errors is an admirable aim, but it is like searching for the end of the rainbow. In reality, we can never eliminate errors altogether. It does not matter if we try to give absolutely precise formal specification or absolutely correct implementation. Any sufficiently complex software system will contain errors.

TEACHER: What makes you so convinced? Do you have experience with a particular application domain?

EPSILON: It is true that the unattainability is more visible in my domain of expertise, which is distributed systems. You cannot assume anything about other nodes and inputs, you need to perform live updates and in some cases, the correct behaviour is simply not defined.[23]

TEACHER: Now, what do you propose to do if an application reaches a state with undefined behaviour?

EPSILON: Just let it crash![24]

TAU: I expect that I'm speaking for ALPHA and BETA too if I say that this is asking for a disaster. Not only will your system

[17] Dijkstra (1970)

[18] Popper (1934)

[19] The use of Bayes' theorem as a model of theory confirmation is discussed in Chalmers (1999), chapter "The Bayesian Approach".

[20] Angius (2014)

[21] This is a paraphrase of Beck's (2003) quote "we (1) write new code only if an automated test has failed."

[22] The three points are adapted from Beck (2002)

[23] According to Armstrong (2003), "exceptions occur when the run-time system does not know what to do [and] errors occur when the programmer doesn't know what to do."

[24] The "let it crash" slogan appears in Armstrong (2003)

stop working until you restart it, but you are also likely to leave your data in an irrecoverably inconsistent state. Miscomputation must be avoided, even if it is more expensive!

EPSILON: Of course, I'm not suggesting to let the *whole* system crash. Instead, we need to compose systems from small isolated components, or processes, that can be safely killed. When a process dies, we let someone else deal with it.[25]

You will have a hierarchy of supervisor processes and when one process miscomputes, the parent is responsible for a correct recovery and restart of the child processes. And because processes are very lightweight, we can recover from errors without causing any system outage. In fact, this model is more reliable than avoiding miscomputations because we can recover even if a cosmic ray flips a bit in the memory!

TEACHER: We can find a useful analogy with a *safety factor* in other engineering disciplines. A civil engineer will calculate the worst case load for on a beam, but then build it ten times stronger, or at least twice as strong. Such over-engineering is extremely effective and is even required by law for bridge building.

BETA: The idea of over-engineering is something I would like to see in software engineering code of ethics. But what safety factor should we require for software products?

EPSILON: I never thought about it in such a formal sense, so I have to admit, I do not even know how to calculate such safety factor for supervisor-based applications.

TAU: Now you got me interested too. The safety factor for a bridge can be calculated based on the expected load, but in doing so, we assume a certain linearity. Increasing the strength of material twice provides a safety factor of 2. But software systems might involve feedback loops or non-linearity where safety factor of 2 requires tenfold over-engineering.

What we need is a theory of stability, or perhaps a type system that can guarantee that certain amount of supervision does, indeed, provide the required safety factor![26]

TEACHER: Judging by the way you are talking about the idea, TAU, it almost seems that EPSILON convinced you to embrace miscomputations! Or am I mistaken?

TAU: But we are not talking about miscomputation here at all! All the so called "invalid states" that EPSILON proposes to handle by supervision and recovery are now perfectly valid and expected! It is just a different way of expressing your program, but as all the ones we talked about earlier, it does eliminate all miscomputations from the software system.

TEACHER: In a philosophical sense, we could even claim that no software can ever miscompute or *malfunction*. It is always doing what it was programmed to do and thus it does not malfunction in the same way in which a screwdriver made from glass will fail to fulfil its function to turn screws.[27]

That said, if we write code to handle certain situations then the behaviour that results from the other, unexpected, situations is undefined and we can see it as miscomputations. It does not matter if the unexpected situation is caused by an incorrect input or a dog chewing a power cable. By following EPSILON's method, we accept miscomputation, but only at a lower level of abstraction. At a higher level that is visible to the user, miscomputation does not occur.

TAU: I'm afraid I already see where this is going. OMEGA will argue that we should permit miscomputation at the higher level of abstraction too...

Lesson 5: Errors as communication

TEACHER: I find it interesting that we discover different ways of understanding errors in software by viewing it through the perspective of other disciplines. First through mathematics with TAU and then through engineering with BETA, ALPHA and EPSILON.

OMEGA, you were talking about your musical live coding performance. Are you going to continue this trend by taking inspiration for programming from art?

OMEGA: That is correct. I find many links between software development and music. Many aspects of software development require an artistic approach and we can learn more about software development from the creative and explorative artistic process than from building bridges![28]

Just like music is not about notes written on a piece of paper, software is not about code in your source control. They are both about interaction and communication.

[25] See Armstrong (2003)

[26] We agree with TAU that this is an interesting future work. Type systems have been used not just for proving correctness, but also for computing complexity bounds, e.g. Girard et al. (1992) and we can certainly imagine type systems for guaranteeing minimal safety factor.

[27] The point that software artifacts cannot malfunction has been made by Floridi et al. (2015)

[28] OMEGA is paraphrasing Gabriel and Sullivan (2010).

TEACHER: How is this reflected in your approach to errors?

OMEGA: An error in the performance of classical music occurs when the performer plays a note that is not written on the page. But in genres that are not notated so closely, there are no wrong notes – only notes that are more or less appropriate to the performance.[29]

Musical live coding performance is also not closely notated. You issue commands. Some of them are more appropriate and some of them are less appropriate.

TAU: That is just great... Not only we do not have a complete specification, but now we do not even have to follow it!

OMEGA: I do not understand are you so afraid of errors. When you make an error in a live coding performance, you have many ways of dealing with it. You can compensate for an unexpected result by manual intervention (like a guitarist lifting his finger from a discordant note), or develop the unintended effect further as a serendipitous alternative.[30]

TAU: This is all very nice when you are doing a musical live coding performance, but I do not see how it is relevant to our discussion about serious software development.

EPSILON: I have no interest in live coded music, but I see another similarity. What OMEGA described sounds a bit like working with a REPL environment in Python or Racket.

There, you also type commands and run them immediately. Sometimes you type the wrong thing, see an error message and then you correct yourself. But that is still just a playground, not something you could use to build complex systems.

TEACHER: Should we then agree that OMEGA's live coded music does not teach us anything relevant to professional software development, or are there other areas of software that are similar to live coding?

EPSILON: I have one more example, but again, it is not what I normally think of as programming.

Doing data science using tools like Matlab or R feels like live coding too. You also issue commands, observe the results and correct errors. Error like typos are immediately obvious, but when running machine learning algorithms, it becomes hard to even say what is an error!

ALPHA: Well, if you take such broad view of programming, then TDD is live-coding too. A good test runner runs in the background and shows the failing tests immediately as you are writing your code to provide rapid feedback.

OMEGA: This is exactly how live coding works! In TDD or data science, you are trying to make errors more visible so that you can quickly correct them. It is the same as live coded music where errors are immediately apparent because you can hear them. I do not understand why you hesitate to treat these kind of errors in the same way as errors in business applications or any other kind of software...

TEACHER: This kind of hesitation may arise from different basic assumptions that we have about software, something that philosophers of science call research programmes.[31] Let us try to explore this idea further...

TAU: I do not see any questionable basic assumptions in the discussion. The only problem is that OMEGA is confusing what a program is with how it is created.

Following the same logic, you could say that interactive theorem proving in Coq is also live coding with errors, because you do write code interactively until you satisfy your proof obligations. But this is clearly not a miscomputation. Just a process of construction of provably correct program.

A program is just a linguistic entity that you can analyse for correctness and run. How it is written is not a concern of our present discussion!

TEACHER: Your last sentence is exactly the kind of assumption that defines a research programme![32]

OMEGA: And I think it is fundamentally flawed! Programming is not a task of constructing a linguistic entity, but rather a process of working interactively with the programming environment, using text simply as one possible interface.[33]

TEACHER: I think we finally understood what all the fuss was about. TAU talks about the programs as static entities while OMEGA talks about the whole process of interaction with the machine. But are these two really separate as TAU insists?

BETA: Now that you mention it, I worked on a banking system that was written in Smalltalk in 90s. In Smalltalk, you create software by interacting directly with the environment. The program runs at the same time as you are creating it.

[29] The quote is a paraphrase of Blackwell (2005)

[30] The quote is also a paraphrase of Blackwell (2005)

[31] Lakatos (1967)

[32] This is the Algol research programme as identified by Priestley (2011).

[33] Characterization of Smalltalk research programme, Priestley (2011).

This was useful for rapid feedback, but once the system was working, you did not continue to live code it, except sometimes when it went wrong and you needed to investigate why and fix the error.

OMEGA: But the decision when to watch the system and when to leave it alone was only yours. We need to abolish this artificial distinction between a phase when software is created and a phase when it is autonomously running!

In live coding, you want to interact with the system during the whole performance. In other applications, you interact more frequently in an early phase and less frequently in a later phase. But you still need to be able to interact! For example, when mitigating a hacking attempt, you should be able to connect to the system and live code a defence.[34]

TAU: Well, now I see what you are trying to achieve, but it gives me a headache! If we wanted to guarantee correctness of such systems, we would have to shift from proofs about static programs to proofs about dynamic systems and the interactions that they allow.

I'm not saying it is impossible. It might even be an interesting problem! But I would very much prefer to solve the easier task of proving correctness of static programs first.

Lesson 6: Identifying program properties

TEACHER: Unless anybody wants to propose yet another strategy for dealing with errors, I would like to spend the next two lessons discussing case studies that let us explore how the different strategies work in two scenarios.

The first one is the Y2K bug. This was caused by the fact that many old programs stored only the last two digits of a year in a date and so adding one day to 31/12/1999 produced a date that could be interpreted as 1/1/2000, but also as 1/1/1900.

First of all, let's try to classify the Y2K bug. What kind of error is it and what miscomputations does it cause?

EPSILON: Y2K is interesting. It is not an unavoidable hardware failure, like a cosmic ray flipping a bit in the memory.[35] It is also not a simple oversight, such as errors caused by a typo.[36]

So, it must be occurring at a higher level of abstraction, but how and where exactly, I'm not sure...

TEACHER: It will be easier if we consider a concrete instance of the problem. For example, my bank statement on January 1 2000 reported that $8617 got lost from my savings account! Fortunately, they returned my money back on the next day...

BETA: This an error in coding then. The specification for the banking system specifies how to calculate the compound interest for a given number of years, but due to a sloppy encoding of years, the system calculated the interest based not on +1 year, but based on −99 years![37]

OMEGA: See, this is a nice case where formal proofs are not going to save you. You can prove that your calculations and algorithms are correct, but when it comes down to metal, years won't be idealized natural numbers, but only two digits to save the memory space!

TAU: That would only be the case if you were using program proofs in an informal way.

OMEGA: Isn't that what programming language theory and proofs are all about? Writing proofs about lambda calculus with idealized extensions?

TAU: Proofs about lambda calculus are important for getting the foundations right, but if we talk about proving programs correct, we must not make any idealizing simplifications, but prove properties of the actual implementation.

In this case, it was not an error in coding, but an error, or incompleteness, in the specification. The specification must be detailed enough not to leave room for any ambiguity, such as how are the dates going to be represented. With such detailed specification, a correctness proof would find the issue.

TEACHER: But speaking of the banking system, what is the theorem that would not hold?

BETA: I'm not entirely sure we would find one easily... If the property we prove is that the system correctly calculates the return on investment using the compound interest formula for a given number of years, that still does not prevent us from accidentally calculating it for −99 years...

TAU: Of course, you would also have to specify the right properties for the functions operating on dates... perhaps a monotonicity of the ordering on dates with respect to the function that increments the date by one. Assuming n is a function that returns the next date and $\geq$ is the ordering on dates, we want to prove that $\forall d. n(d) \geq d$.

TEACHER: I wonder how this accounts for leap seconds...

[34] This example is taken from Chassaing (2015)

[35] Fresco and Primiero (2013) refer to these as "operational malfunctions."

[36] Fresco and Primiero (2013) refer to these as "slips."

[37] BETA makes a correct guess, $8617 is what you lose when you assume 2% interest rate on initial investment $10000 over −99 years.

OMEGA: Even now that we know what the error is, finding the right property that would have prevented it is surprisingly hard! How were we supposed to know that we need to prove this monotonicity property before?

ALPHA: It is all about finding the right properties, isn't it? The same problem arises when you are writing property-based tests with tools like QuickCheck[38], which check that a property holds for partially randomly generated inputs. With random testing tools, you can focus more on finding as many useful properties as you can, rather than on writing a longwinded proof for every single one of them.

For example, a test suite for a banking system would likely include a check for a property that adding calculated interest to a savings account never decreases the total balance. The Y2K bug would be easily discovered and miscomputation eliminated before it could happen!

EPSILON: This kind of assumption would be even easier to check in Erlang. You do not even need a complex proof or specification of a property. You would just check if the calculated interest is a positive number. If no, the process miscomputes and terminates itself.

A supervisor process ensures that the system continues to handle other requests. A manual intervention might still be needed later (if the error does not go away on January 2), but we would certainly not send you a bank statement with $8617 disappearing from your account.

TEACHER: No matter how we intend to eliminate miscomputation from our software and whether it is through proofs, testing or supervision, it seems that we have discovered that the key to finding miscomputation are properties.

Is there a chance that we will ever find the right properties and all miscomputations will be eliminated?

TAU: Judging by the difficulty we had when searching for the property to catch the Y2K bug, it seems harder than I was originally expecting, but we must try!

TEACHER: Last, but not least, OMEGA, do you see any way of addressing the Y2K bug with live coding?

OMEGA: Well, live coding is what most of the world did!

TEACHER: What do you mean?

OMEGA: Mission critical systems had an engineer on site to monitor them and fix any issues that may arise. The only problem is that nobody had the right tools for facing the errors.

Rather than being able to monitor the internals of the system and fix the issues as they happen, the only available live coding option was to shut the system down for maintenance!

I may be more interested in live coded music, but I believe the live coding concepts are equally (if not more) valuable when it comes to unexpected conditions in business applications and Y2K only supports my belief.

Lesson 7: The ethics of mission-critical systems

TEACHER: Let's look at another well-known bug. In August 2012, the trading firm Knight Capital lost $440m in less than 45 minutes due to an erroneous deployment of their software.

The company removed outdated code from the software and repurposed a flag that was used to trigger it. It then deployed a new version of the software on all servers except for one – and turned on the flag. This executed new routine on the updated servers, but it triggered the outdated code on the one server running the old version, causing the system to issue millions of erroneous trades.[39]

How could we avoid this kind of bug using the different strategies that we discussed earlier?

OMEGA: I cannot believe it took Knight Capital 45 minutes to turn off the malfunctioning server!

The lesson from live coding here is clear. You should design the system to make such errors immediately visible and you should be able to quickly manually intervene. You should be able to stop incorrect trading just like you can stop a discordant beat that you accidentally play in live coded music.

TEACHER: Should trading system be designed in the same way as live coding systems, or is that just OMEGA's perspective?

EPSILON: I can see how a manual coding intervention could correct the error promptly, but why not avoid it in the first place? The scenario sounds exactly like the situation for which Erlang and its "let it crash" approach has been designed! Erlang has been used in telecommunications for systems that need to run for years without an outage and are updated on the fly.

Rather than repurposing a flag, you would simply add a new kind of message. If an old system was running on one of the servers, it would crash when it receives a message it cannot handle – and it would get restarted while all the updated servers would continue working fine.

TAU: I find it surprising that the software industry rarely discusses its ethical obligations. Live coding might be a good

[38] Claessen, K., Hughes, J. (2011)

[39] USA SEC (2013)

workaround for the issue and the practical experience with Erlang in telecommunications is, indeed, noteworthy.

However, the only way to provide true guarantees of correctness is through formal methods and that is the ethical approach we should strive for![40]

EPSILON: Speaking of algorithmic trading and ethics might get a little odd... But I understand we are talking about mission-critical systems more generally.

ALPHA: The difficulty with proving the system correct lies in the problem discussed in the previous lesson. No matter if you use proofs or tests, what are the properties of the system that you want to guarantee?

TEACHER: This is a valid concern, but perhaps we could find some for a relatively well-defined system such as this one...

ALPHA: Presumably, trading systems try to use more and more clever strategies that go against the intuition of everyone else in the market. If we formalized basic assumptions about them, we might rule out new profitable strategies.

OMEGA: Then you could even claim that using formal methods is unethical, because of the missed opportunity cost it creates!

ALPHA: I also expect that more and more algorithms are using sophisticated machine learning methods. And even leaders in the field can get those terribly wrong. Just look at the recent case of Google Photos tagging black people as gorillas![41] This was a sad failure. Is there a formally provable property that could have avoided it? I doubt that...

TEACHER: Let me attempt to summarize todays lesson. When it comes to mission-critical software, all of us aim to create software that matches the highest quality standards we can think of, but we do not seem to agree what that means!

BETA: I may be wrong, but I suspect this is a question that is not going to be resolved anytime soon. This is why I was advocating a code of ethics for software engineers earlier. It does not give a definite answer, but it ensures that professional software developers are aware of their ethical responsibilities.

Lesson 9: Searching for a grand unification

TEACHER: This will be our last lesson and even though we will probably not find a grand unification of all the methods and ideas, we should try! After discussing them individually and in the context of two concrete miscomputations, we should be able to find similarities between some of the methods and perhaps also find ways in which they can work together to achieve the common goal.

In order to even recognize errors, or miscomputations, we need to know what does it mean for software to be correct?

BETA: The simple answer is that correct software corresponds to its specification, but there are sadly many issues with this answer... A complete specification is hard to obtain and for some systems, it may be complex or difficult to formalize.

TAU: The ideal state is when the specification can be written in terms of simple properties. For example, a compiler should preserve the semantics of the compiled program.[42] I do accept, though, that finding such properties is not (yet!) always easy...

ALPHA: As an aside, I always thought that types and tests are in an opposition in a way, but I have to agree with TAU here. No matter if you write tests or proofs, you are trying to verify properties of software!

TEACHER: Now, what about the software with not so clearly identifiable properties?

OMEGA: I believe this is the more interesting kind of software! If we cannot find simple properties, the only (and the best) specification is the source code itself.

I think those are two extreme ends of a spectrum and there is much in between. As we use more expressive languages, the source code becomes simpler, more declarative and closer to the description that you would write in a specification.

TAU: Similarly, better formal methods let you capture more complex program behaviour in terms of provable properties.

EPSILON: For the kind of software that is less amenable to formal specification (and concurrent or evolving systems are a prime example), we need to accept that miscomputations can happen and we need a way to live with them. This is where supervision and the "let it crash" methodology comes in.

OMEGA: Not just live with them! We also need an effective way of reacting to miscomputations. And let me add that those may occur even in systems with formally provable properties, until we find a way to prove that we found *all* such properties!

No matter if we need to correct a miscomputation (in a trading system) or use it as a creative inspiration (in a live coding performance), we need to be able to react quickly.

[40] This argument has been made, for example, by Abramson and Pike (2011)

[41] Barr (2015)

[42] Leroy (2012)

TEACHER: Even with the most effective live coding systems, the reaction speed seems to be limited to the speed at which the coder can change the source code. Should we then make our software slow enough so that a human can intervene?

OMEGA: Reaction time in live coding is certainly longer than lifting a finger when playing a guitar. But you can also live code your environment[43] and use it to run predefined commands. This is perhaps similar to how human brain works[44] – we also have a fast subconscious subsystem and a more deliberate but slower conscious subsystem. So the human brain can serve as an inspiration for future live coding systems.

TEACHER: I find out present discussion surprising in one way. We discuss how to approach miscomputation depending on how the system behaviour can be specified, but not whether the system is mission-critical or not.

This is an interesting change from the common narrative. The usual claim is that formal verification is especially important for mission-critical systems, but we talked about using it only for mission-critical systems with simple specification. What can we do about systems with complex behaviour that do not have easily identifiable properties?

TAU: Even in such complex systems, there should be some fundamental properties that define their correctness which we should be able to prove!

BETA: Perhaps we can structure such systems into multiple layers and start by finding some properties at the lowest and also the simplest level of abstraction?

TAU: Yes, I believe that is the right way of approaching the problem. In a mission-critical system, we should find basic properties that are sufficient for proving that the system will continue to function, but perhaps will not function optimally.

TEACHER: To conclude, TAU, you would be happy to get on an airplane that is live coded by OMEGA, provided that there are basic properties proving that it will not attempt to land at a high speed and it will not change direction or altitude in ways that would cause it to break. But where exactly will it take you and whether it will be in 3 or 5 hours, that is something that the live coder can decide based on his or her current mood!

References

Aaron, S. and Graham, J. (2015). *Meta-eX: Live coding duo*. Recordings available online at: http://meta-ex.com/

Abramson, D., Pike, L. (2011). *When formal systems kill: Computer ethics and formal methods*. In APA Newsletter on Philosophy and Computers.

Angius, N. *The Problem of Justification of Empirical Hypotheses in Software Testing*. Philosophy & Technology, Volume 27, Issue 3, pp. 423-439, 2014

Anderson, R. E., et al. (1993). *Using the new ACM code of ethics in decision making*. Communications of the ACM 36.2.

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

Auslander, J. (2008). *On the Rotes of Proof in Mathematics*. In *Proof and Other Dilemmas: Mathematics and Philosophy*. Gold, B. and Simons, R. A. (eds.). ISBN 0883855674.

Babbage, Ch. (1837). *On the Mathematical Powers of the Calculating Engine*. Reprinted in Texts and Monographs in Computer Science, ISBN 978-3-642-61814-7

Barr, A. (2015). *Google Mistakenly Tags Black People as 'Gorillas,' Showing Limits of Algorithms*. Wall Street Journal. At: http://blogs.wsj.com/digits/2015/07/01/google-mistake/

Beck, K. *Test driven development by example*. Addison Wesley, 2002. ISBN 978-0321146533

Blackwell, A., and Collins, N. *The programming language as a musical instrument*. Proceedings of PPIG 2005.

Chalmers, A. F. (1999). *What is this thing called science?* Open University Press. ISBN 0335201091.

Chassaing, J. (2015). *If you are not live coding, you're dead coding*. NCRAFTS, Online at: https://vimeo.com/131658147

Chlipala, A. (2013). *Certified Programming with Dependent Types*. MIT Press, ISBN 9780262026659

Claessen, K., Hughes, J. (2011). *QuickCheck: a lightweight tool for random testing of Haskell programs*. In ACM SIGPLAN Notices 46.4, pp. 53-64.

Dijkstra, E. (1970). *Notes On Structured Programming*. EWD249, Section 3 (*On The Reliability of Mechanisms*)

[43] A good example is the use of the Emacs editor by the live coding duo Meta-eX. See Aaron and Graham (2015).

[44] Kahneman (2011).

Ensmenger, N. L. (2010). *The Computer Boys Take Over*. MIT Press, ISBN 9780262050937.

Flaisher, A., Gluska, A., Singerman, E. (2007). *Case study: Integrating FV and DV in the Verification of the Intel® Core^{TM} 2 Duo Microprocessor*. In proceedings of FMCAD, pp. 192-195

Floridi, L., Fresco, N., Primiero, G. (2015). *On malfunctioning software*. In Synthese, Volume 192, Issue 4, pp 1199-1220

Fresco, N., and Primiero, G. (2013). *Miscomputation*. In Philosophy and Technology, 26,253–272.

Gabriel, R. P., and Sullivan, K. J. (2010) *Better science through art*. ACM SIGPLAN Notices. vol. 45. no. 10. ACM.

Girard, J.-Y., Scedrov, A., and Scott, P. J. (1992). *Bounded linear logic: a modular approach to polynomial-time computability*. In Theoretical computer science 97.1: pp 1-66.

Gold, B., Simons, R. A., eds. (2008). *Proof and Other Dilemmas: Mathematics and Philosophy*. ISBN 0883855674.

Hacking, I. (1983). *Representing and Intervening: Introductory Topics in the Philosophy of Natural Science*. Cambridge University Press. ISBN 0521282462.

Hoare, C. A. R. (1996). *How Did Software Get So Reliable Without Proof?* Proceedings of FME, Springer.

Kahneman, D. (2011). *Thinking, fast and slow*. Macmillan. ISBN 978-0385676533

Lakatos, I. (1976). *Falsification and the methodology of scientific research programmes*. Springer Netherlands, 1976.

Leroy, Xavier. (2012). *The CompCert C verified compiler*. Documentation and user's manual. INRIA Paris-Rocquencourt.

MacKenzie, D. (2004). *Mechanizing Proof: Computing, Risk, and Trust*. MIT Press. ISBN 9780262632959.

Meyerovich, L. A., Rabkin, A. S. (2012). *Socio-PLT: Principles for programming language adoption*. In Proceedings of ACM Onward! Conference, ACM.

McBride, C., and McKinna J. (2004). *The view from the left*. In Journal of functional programming 14.01. pp69-111.

McCarthy, J. (1963). *Towards a mathematical science of computation*. In: Popplewell, C.M. (ed.) *Information Processing 1962: Proceedings of IFIP Congress*, v. 62, pp. 21–28.

Manolios, P. (2008). *The Challenge of Hardware-Software Co-verification*. In Verified Software: Theories, Tools, Experiments (Bertrand M., Woodcock, J., eds.) pp 438-447.

Milner, R. (1978). *A theory of type polymorphism in programming*. Journal of computer and system sciences 17, 3, 348–375.

Priestley, M. (2011). *A Science of Operations: Machines, Logic and the Invention of Programming*. Springer.

Popper, K (1934). *The logic of scientific discovery*. ISBN 8130908115, Routledge, 2005.

United States of America, Securities and Exchange commission (2013). Administrative proceedings in the matter of Knight Capital Americas LLC, File No. 3-15570.

Wright, A. K., Felleisen, M. (1994). *A syntactic approach to type soundness*. Information and computation 115.1, pp 38-94.