Homologated software should come with a formal dialogue fragment

Joost J. Joosten
University of Barcelona

April 2022

This is a position paper presented and discussed at the

Transatlantic Dialogue on Humanity and AI Regulation Event

in the panel on AI and Human Decision Loops on Thursday, May the 12th, 2022.

`https://tinyurl.com/2p8uyfzd`

Paris Position Paper

Software is ever more present in our society and important tasks are entrusted to algorithms. How do we know that the algorithms do what they should be doing? Various court rulings have asked for homologation of software: a seal of quality so to say for software.

In this position paper I will make my point that we do not really know what software homologation is or should be. Next, I will expose the only serious proposal that I am aware of which is the one that we help to develop and put to practice in our group in Barcelona. I will finish by proposing a new ingredient to the homologation protocol: each homologated software should come with an identified well-behaved but expressive enough formal language that will enable the public officer or a citizen to enter in a controlled and well defined dialogue with the homologated software.

Before we start out, a disclaimer: of course we are aware that legal software can only be applied in a very limited area. But even if large part of the legal decision making is discretionary, once it is decided to include certain software in the process of decision making, these software should be subject to severe requirements at the risk of endangering civil rights. It is in this panorama that the need of software homologation/certification has been articulated.

As I mentioned before, there is no consensus on *what* software homologation exactly is. However, one important ingredient is omnipresent in all conceptions of software homologation, namely software homologation should at least certify that a program does what it claims it does.

Before continuing the exposition, it may be good to state that in a very precise sense one can prove that unrestricted homologation of software does not exist. To make this statement a bit more precise, let us define what we call here a *universal homologation program.* We call a program P a universal homologation program whenever P takes two inputs

1. another program Q in a language that can be recognized by P and,
2. a specification S in a language that can be recognized by P that describes behavior of programs;

and whenever given these two inputs Q and S, the program P outputs

"YES´´ if program Q does what the specification S says and, outputs

"NO´´ if the program Q does something different than as was claimed by S.

The non-existence of unrestricted software homologation can now be formulated as follows: given a minimal requirement on the expressibility of a programming language, a universal software homologation program cannot exist.

It is important to be aware of the nature of this statement. It does not say that we currently have not found such a program P. No, it says that it is a logical impossibility that such a program P can exist. This impossibility is a mathematical theorem.

Of course, in order to prove the theorem, one should make precise various notions such as an arbitrary program, what it means to be a specification, and what it means for a program to comply with a specification. However, after decades of mature mathematical logic and computer science, the communities have indeed made these concepts precise in a very satisfactory, convincing and widely accepted manner and the theorem can be proven.

This is bad news for regulators. Requiring that any possible software has to be homologated is in some sense like requiring that any police officer be a married bachelor: an outright impossibility. However, there is hope. In order to prove that no universal homologation program P exists, it is essential that one can input *any* arbitrary program Q and any arbitrary specification S to the alleged universal homologation program P. If we restrict the Q to a special class Γ of programs, then *restricted homologation programs* P will exist. For example, we can consider Γ to be the class of all programs that do not contain any loops.

The additional good news is, that most if not all automatisable problems that are encountered in the public administration can be solved with some program that falls within a simple class Γ of programs so that a restricted homologation program P' exists for Γ.

As such, the homologation of any program Π from the class Γ by a homologation program P' restricted to Γ shall exist of three ingredients $\langle \Sigma, \Pi, \Delta \rangle$:

1. A formal specification Σ in a language that can be recognized by P' that tells what the program Q should be doing: how the program behaves. So, Σ will tell us what to output on what input. For example, it should output "LEGAL´´ if a truck driver has driven only four hours in an entire week and it should output "ILLEGAL´´ if that truck driver has driven 14 hours straight without any break in between.

2. A program Π in a language that can be recognized by P' that should perform the task stated in the formal specification Σ.

3. A certificate Δ for which it is checked by P' that the program Π indeed complies with the specification Σ. Typically, Δ will be a mathematical *proof* that the program Π complies with the specification Σ.

The good news is, we can take Γ to be large enough so that practically any program that is used in public administration belongs to Γ. The bad news is, making a triple $\langle\Sigma, \Pi, \Delta\rangle$ is about a thousand times more expensive than conventional programming. However, I think that for *critical* software we have no other choice. We have seen the consequences of non-verified/homologated software: plane crashes, rocket explosions and seriously infrictions of civil rights.

I think that it is useful to observe that the hard part of software homologation is in *finding* the certificate/proof Δ. As a matter of fact, the programmer will normally find out that her envisioned program Π actually contains bugs wrt to the specification Σ when trying to come up with a proof/certificate Δ. One can only find a proof/certificate of correctness Δ in case there is absolutely no error in the program Π at all. Moreover, the proof Δ is checked by a small and trustworthy computer program called a *proof assistant.* Where finding the certificate is hard, checking that the proof/certificate is indeed a simple task.

The above sketched homologation method will warrant that the verified software will not contain a single error with respect to its formal specification (of course, given some reasonable assumptions). Clearly the specification may be wrong but at least the program is mathematically proven to be error free wrt the specification.

This is a huge step forward and I foresee that in the next decade or so administrations will become aware of this possibility of generating error-free software and simply require it for critical software.

The current set-up however does not seem to help much to fundamental questions like transparency and contestability: even by seeing and studying the certificate, the individual will not be enlightened much.

Now, why are we so much concerned with errors in computer programs? We know that judges, lawyers and other officers alike also make mistakes. The fundamental difference is, that we can enter in a *dialogue* with an officer. If we doubt something, we can ask for explanations and obtain more and more information and clarity of the officer about the motivation of the decision. With computer programs this is typically not the case.

To mitigate this situation, I propose that a homologation protocol should come with a well defined formal language fragment where the inquisitor can formulate questions about the behavior of the software. In particular, is this software treating all individuals equally? For example, in the formal language it can be formalised that any individual with two passports is treated equally to any individual with only one passport. And homologated software should come with a mechanism to decide and certify these kind of questions. It is thus to the logician and software engineer to work with feasible fragments of logic so that this dialogue becomes possible and feasible.

Once such a mechanism is there, the citizen can indeed enter in a dialogue with the software. In this particular sense, the software can *explain* the decisions it made. Only in such a situation can we start to consider homologated software accessible to the larger public.