Software != Programs
Software Behaviour as an Emergent Property

Anna Maria Eilertsen
Universty of Bergen, Norway
anna.eilertsen@uib.no

Anya Helene Bagge
Dept. of Informatics
University of Bergen
anya@ii.uib.no

⟨Programming 2018 Posters⟩

What's the Meaning of Meaning? When we talk about the behaviour of a computer program, 'behaviour' can have different meanings depending on the context. For example, we may consider a program as a function from input values to output values (a mathematically inspired notion); or we may consider it a construct in a logical system, and reduce it using the semantics of the programming language (a reductionist notion); or we may consider the program a collection of modules, each of which has its behaviour defined in documentation and unit-tests (a developer's notion); or, we may consider the program a tool to render a web page or control a robot, and consider its behaviour based on its ability to perform (the customer's and user's notion).

As researchers, this is a concern we face when working with "behaviour-preserving program transformations", like program optimisation, compilation, refactoring or architectural changes. In each of these fields the aim is to preserve *program behaviour*, but they each attribute different meaning to the term. Even within a single field, like refactoring, practitioners and researchers do not agree on the terminology used.

For example, a formal methods researcher may be concerned with preserving program 'meaning' in the sense explored by Floyd [1] and Hoare [2] and later attempts at formal specification. A practitioner, on the other hand, may be more pragmatic, and learn about refactorings from non-scientific literature (such as Fowler's book)), and care about "observable" (to a human) program behaviour. This is a property of their mental model rather than of the program semantics; which is evident by practitioners' use of refactoring as a terminology for "refactoring away bugs". For a researcher this cannot be correct: removing a bug is a change in program semantics, and thus is not behaviour preserving. Fowler calls this "preserving the program's *intended* behaviour".

The terminology discrepancy becomes a problem when researchers try to provide tools and theory that are useful in industry. When we consider program behaviour a property of the semantics of source code, it follows that to preserve behaviour the original program and the transformed program must be equivalent up to programming language semantics. This is typically done by aiming to prove that the program transformation is an identity transformation.

Now, most programming languages do not normally have formally specified semantics. Still, we know that there is a concrete mapping of the source code down to the hardware level: after all, it does run on the computer! Unfortunately, there are often several different code translators (the different C compilers, different Java compilers, and different hardware interpretations of the machine code. Which means that we have to either be able to map between compilers, or to repeat the work per one. However, for complex refactorings, this would nonetheless require rigorous proving (theoretical) and implementation (practical) efforts, both of which is quickly outdated as new language versions are released.

The feasibility of implementing semantics-preserving transformations is hampered by the quality of static analysis tools, which in some languages are unable to check preconditions required to guarantee semantic equivalence [3–5].

Preserve Software Behaviour, Not Program Behaviour! We suggest that what practitioners, and Fowler, talk about is in fact not "programs", but *software*. And we suggest that *software behaviour* is fundamentally different from program behaviour, and must be treated differently. Software behaviour can be considered an emergent property of the underlaying system (of programs

and hardware and libraries), and understood in terms of the ends it serve, not merely in terms of the causes that brought it about: *"When the effects not only happen, but are included in the idea of the object; when they are not only seen, but foreseen; not only expected, but intended."* [6].

While traditional programs can be seen as an execution of a mathematical or theoretical idea, software is the model of the solution to a real world problem. This does not mean that the problem must be one of people or cars (or rabbits): it can be the problem of making your font look right in your PDF-reader, or place the exit-button in the right place on your screen. But if your *software* is a PDF-reader, then it might consist of a program for rendering a font, a program for connecting that to the PDF-source code, a program to load system colours, et.c.

From this point of view, the original notion of program behaviour is much too strict. As we consider that software can be composed of programs in different languages, running on different hardware in different locations, and can be decomposed, reengineered, and even trained, we see the motivation for a terminology for a different kind of behaviour, one that does not require a complete syntactic mapping of all program elements of each involved version of each programming language. One that can be considered persistent across decomposition of monolithic systems into micro-services, or API refactorings.

This is the black box approach of observable behaviour. Conceptually, this is only somewhat different from the traditional way: we are concerned with the internal state of the computer, but only insofar as we can observe the state of values as they propagate through abstraction levels and affects the intersection between our software and the rest of the world. As long as it walks, swims and quacks like a duck, we don't care what's under its feathers.

In practise, this requires a shift in thinking; at least for academics with a formal methods upbringing. We do not argue for throwing away our traditional program behaviour (indeed, that is why we must stop overloading the term, and instead talk about *software behaviour*). The purpose of our code is not the code itself, but the effect is has on the surrounding system. If the effect is constant, then internal changes does not matter, even if they (technically or isolated) are semantic changes.

Now, a developer should not have to think about semantics. Rather, the developer should work with APIs, or models; abstractions of the real world (customers, insurance, cars, pedestrians) rather than abstractions of hardware operations. In practise, this means that refactorings available for your program can be generated from your tests, from your specification, or from collected information on client programs: as long as your program is observable equal to the world, you can change anything. If no one uses a feature of your library, then you can remove it. If an execution path in your program is never used, you may remove the relevant code.

Note that, this distinction between software and programs is not formalised stance in computer science. In fact, the seeming interchangeability between the terms complicates research, outreach, and cooperation with laypeople significantly. We hope to start a discussion that can result in a bridge over the gap between the strict academic notion of program behaviour and program correctness, and the messy reality encountered when one wish to use and implement tools used by the "general public" of developers.

References

[1] Robert W. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32. American Mathematical Society, 1967.

[2] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[3] BarbaraG. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In Görel Hedin, editor, *Compiler Construction*, volume 2622, pages 126–137. 2003.

[4] Anna Maria Eilertsen. *Making software refactoring safer*. PhD thesis, Master's thesis, Department of Informatics, University of Bergen, 2016.

[5] J. Brant and F. Steimann. Refactoring tools are trustworthy enough and trust must be earned. *IEEE Software*, 32(6):80–83, Nov 2015.

[6] Michael Ruse. Do organisms exist? *American Zoologist*, 29(3):1061–1066, 1989.