

Software != Programs

Software Behaviour as an Emergent Property

{Anna Maria Eilertsen, Anya Bagge} : University of Bergen, Norway

Software Behaviour

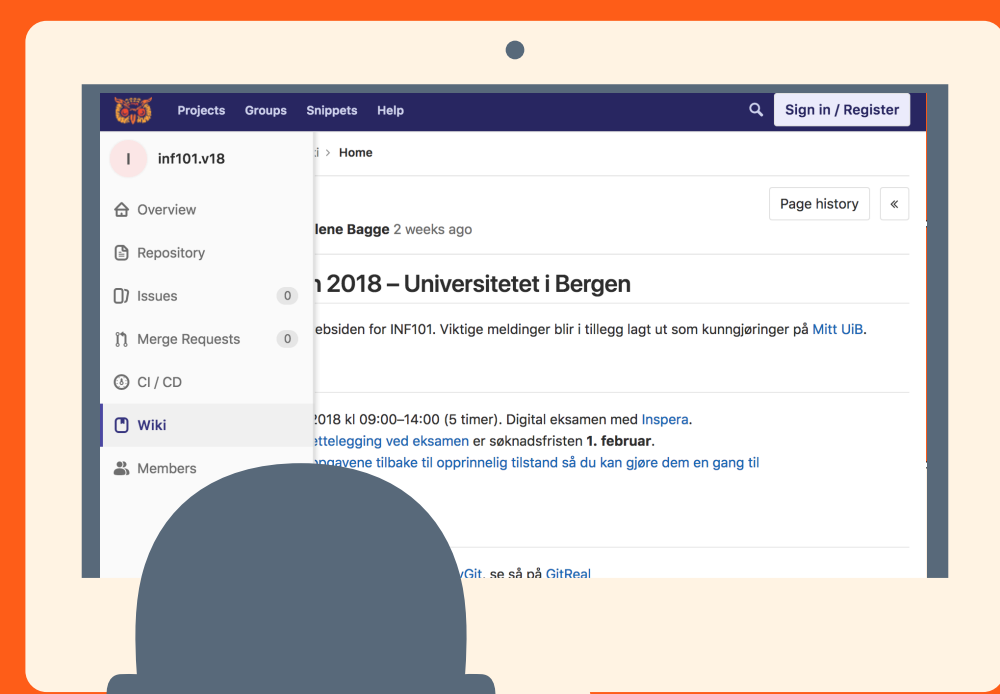
Software is intended to solve some problem.
Software is *experienced* by someone: a user, other software, hardware. It *does* something.
Software may consist of many programs, potentially in different languages.
Software as a solution *models* something.
Software may rely on, and co-evolve with, external libraries or APIs.
Different software may be interchangeable.
Software behaviour may be emergent:

Emergent properties are those that cannot be adequately described in terms of its underlying components. Software behaviour may need be expressed as its effect on its surroundings, and thus not reducible to low-level program elements.

Program Behaviour

Programs are high-level descriptions of low-level processes.
Programs are reducible to low-level operations.
Programs can be expressed in formalisms.
Programs do not carry inherent meaning.
Programs can be seen as a function from input values to output values.
Programs can be formally verified.
Programs are not emergent:

Emergent properties are those that cannot be adequately described in terms of its underlying components. Program elements can be reduced to low-level instructions, and are thus not emergent.



Preserves functionality and client happiness

Preserve program behaviour!

Provably semantically equivalent

```
private RefactoringStatus checkExpression() throws JavaModelException {
    Expression selectedExpression= getSelectedExpression().getAssociatedExpression();
    if (selectedExpression != null) {
        final ASTNode parent= selectedExpression.getParent();
        if (selectedExpression instanceof NullLiteral) {
            return RefactoringStatus.createFatalErrorStatus(RefactoringCoreMessages.
        } else if (selectedExpression instanceof ArrayInitializer) {
            return RefactoringStatus.createFatalErrorStatus(RefactoringCoreMessages.
        } else if (selectedExpression instanceof Assignment) {
            if (parent instanceof Expression && !(parent instanceof Parent)) {
                return RefactoringStatus.createFatalErrorStatus(RefactoringCoreMessages.
            } else {
                return null;
            }
        } else if (selectedExpression instanceof SimpleName) {
            if (parent instanceof Declaration) {
                return RefactoringStatus.createFatalErrorStatus(RefactoringCoreMessages.
            } else if (parent instanceof QualifiedName && selectedExpression.get
                || parent instanceof FieldAccess && selectedExpression
                return RefactoringStatus.createFatalErrorStatus(RefactoringCoreMessages.
            } else if (selectedExpression instanceof VariableDeclarationExpression) {
                return RefactoringStatus.createFatalErrorStatus(RefactoringCoreMessages.
            }
        }
    }
    return null;
}
```

Behaviour preserving transformation:

may change or improve your program

Semantics-preserving transformations:

will also preserve your errors

Fixing bugs

Fixing a bug implies changing the semantics of code - unless the bug was in unreachable code. A correct refactoring tool should, according to the definition, never fix bugs. However, it preserves the software's intention.

Migrating between languages

Unless there exist a formalism of the transformation between languages, or equivalent reductions of the original programs, we can usually not claim that they are semantically equivalent. It may still preserve more of the original data than Replacing your program with another that offers the same functionality.

Replacing a piece of code with one that runs faster

While commonly accepted as semantics-preserving, changes in run time may be considered behaviour changing by the user (in particular when if the code runs slower). It may also impact security, as in a timing attack. Consider the example of an app that checks your password: if an erroneous length produces a "wrong password" response within milliseconds because it is checked locally, while a wrong password of correct length takes noticeably longer to produce the error message, the security is changed, but the semantics is preserved.

Replacing your program with another that offers the same functionality

Normally not semantics-preserving. May in many cases still be ok: e.g. migrating libraries, replacing the program with one written in another language, but with equivalent behaviour. Hard to formalise. However, this is the case with model-generated code: we trust the generated code to fulfil the functionality as described in the model.

Decomposing monoliths to microservices

This shifts parts of the program communication style from language-level to API-level, thus leaving the semantics domain of the language. Decomposition like this is hard to capture as semantics-preserving; however the software behaviour is likely to be preserved

Removing unused code

Preserves semantics. May change software functionality, if it is used (undetectably) by external modules. May change developer's mental model of the code.

Changing error messages

Typically semantics-preserving, but it changes the user and developer experience and potentially the security.

Updating a library version

Not considered a semantics-preserving transformation, but is normally assumed to preserve behaviour unless update contains breaking changes.

Changing training data

If a software module relies on machine learning, then the functionality will change if the training data is changed: this is not about program semantics, but about software functionality. Then it can be very hard to properly define behaviour, as in the example of Google Translate interpreting Russia as "Mordor"[4].

Adding comments

Normally semantics-preserving. May improve software significantly.

Adding a method to your public library API

For API changes, the 'semantic preserving' must be regarding the client code. Unfortunately, API developers normally do not know how clients use their APIs. Adding a method with a unique name is normally a semantics-preserving change in a program. In the context of APIs however, it may have the same name as a client method, and thus produce conflicts upon client-site update [5].

Silver bullet

Clarify terminology

Overloading of terms complicates research, outreach and cooperation with practitioners.

When terminology is overloaded, we should consider the different ways in which it is used, and introduce differentiating terms with clear meaning.

Terminology discrepancies are not simply due to 'laypeople misuse' of terminology 'owned' by academics.

Tools and research should be clear on what kind of behaviour they intend to preserve. Refactoring tools may be very useful even if they are not semantics-preserving, but it must be clearly communicated.[1]

Formalise software specification

"Even perfect program verification can only establish that a program meets its specification. The hardest part of the software task is arriving at a complete and consistent specification, and much of the essence of building a program is in fact the debugging of the specification." [3]

To formalise software behaviour, we must talk about modelling, not just programming as mathematics and logics. "Good models need good modelling languages." [2] and "programming is mapping from a problem domain to an executable computer program" [6].

Related work:
[1] J. Brant and F. Steimann. Refactoring tools are trustworthy enough and trust must be earned. IEEE Software, 32(6):80-83, Nov 2015.
[2] Henderson-Sellers, Brian, et al. "Software modelling languages: a wish list." Modeling in Software Engineering (MISE), 2015 IEEE/ACM 7th International Workshop on. IEEE, 2015.
[3] Brooks, F., and H. J. Kugler. No silver bullet. April, 1987.
[4] Petricek, Tomas. "Miscomputation in software: Learning to live with errors." arXiv preprint arXiv:1703.10863 (2017).
[5] Jezek, Kamil, and Jens Dietrich. "API Evolution and Compatibility: A Data Corpus and Tool Evaluation." Journal of Object Technology 16.4 (2017).

[6] T.R.G. Green and M. Petre, Usability analysis of visual programming environments: A cognitive dimensions framework. Journal of Visual Languages & Computing, 7(2):131 - 174, 1996
[7] 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.
[8] C. A. R. Hoare. An axiomatic basis for computer programming. Communications of the ACM, 12(10):576-583, October 1969.
[9] Martin Fowler. Refactoring: improving the design of existing code. Addison-Wesley, 1999

