Introducing static analysis to a mature project

Jacob Sparre Andersen

JSA Research & Innovation

January 2017

Getting started...

I have previously been presented with anecdotal evidence that it is very hard to get any benefit from static analysis tools, if they aren't used on a project from day one, since they have a tendency to need the developers to stay within a limited set of patterns, which the tools can recognise as correct.

The task of introducing static analysis tools to the maintenance process for a close to 30 years old application, is thus not easy, but we're giving it a try anyway...

Magnitudes

The application used as a case:

- Number of actively maintained variants:
  5 (that I know of)
- Number of variants in production:
  $\approx$ 30
- Number of Ada source files (excluding generated ones):
  $\approx$ 3_500 (per variant)
- Lines of Ada source text (excluding generated code):
  $\approx$ 500_000 (per variant)

There is a big overlap between the different variants, but each variant is maintained as an independent project.

An example

Our first experiment[1] was with AdaControl and the rule "Unsafe_Paired_Calls", which we can use to find places where started SQL transactions risk not being committed or rolled back[2].

[1] Besides enabling more compiler warnings.

[2] Or where they can be rolled back without being started first.

An example

Our first experiment[1] was with AdaControl and the rule "Unsafe_Paired_Calls", which we can use to find places where started SQL transactions risk not being committed or rolled back[2].

It was very effective:

AdaControl found thousands of places, where the source text didn't match the two "Unsafe_Paired_Calls" rules we had formulated.

[1] Besides enabling more compiler warnings.

[2] Or where they can be rolled back without being started first.

Improving the static analysis tool

Given the history of the application we didn't quite believe things were as bad as AdaControl reported.

We started categorising the findings, and this lead to an expansion of the "Unsafe_Paired_Calls" rule in AdaControl, such that AdaControl now recognises the most commonly used "safe" pattern of unsafe paired calls in the application.

This made a significant reduction in the number of rule violations reported by AdaControl.

Zero warning policy

Even though improving the tool helped quite a lot, there are still many more rule violations than we have resources to fix immediately.

If we worked under a “zero warning policy” we would have to either:

- Remove the check.
  ⇒ Less tool support for improving quality.
- Take the time to do it anyway.
  ⇒ Not delivering on time to customers.

Neither is a good choice, but there is no doubt we want to deliver what and when we have promised to the customers.

Selective checking

We decided to try an intermediate solution in the form of this rule:

> *Full static analysis is enabled for*
> *all new compilation units, and for*
> *all compilation units subjected to significant changes.*

Selective checking problems

Theoretically this is a quite nice rule.

Selective checking problems

Theoretically this is a quite nice rule.

In practice somewhat less so.

The rule requires the developer to make a decision.

This decision will in practice be decided based on the spare time available to the developer and his/her estimate of how much time it will take to fix the revealed problems (and pattern mismatches).

Our experience so far is that developers only very rarely make significant changes to a compilation unit. – Practically no old compilation units have been added to the set of checked units.

Selective checking problems

We needed a better strategy...

Non-zero warnings policy

When a “zero warning policy” doesn’t work, why not try a “non-zero warnings policy”?

Non-zero warnings policy

When a “zero warning policy” doesn’t work, why not try a “non-zero warnings policy”?

- If you don’t have any warnings, you’re not checking hard enough.
- The number of warnings (rule violations per rule) should be decreasing with time.

And we can always add more rules later on.

Non-zero warnings policy: Tooling

A non-zero warnings policy is more challenging to implement than a zero warning policy, since you need tooling to **keep track** of the development in the number of warnings.

We have decided to keep track of:

Project	What we’re analysing.
Source file	What we’re analysing.
Run	A counter – providing temporal evolution.
Tool	How we’re analysing.
Rule	How we’re analysing.
Violations	Number of rule violations.

Non-zero warnings policy: Tooling

The build-analyse-test process needs a slight modification to implement a non-zero warnings policy:

Instead of just converting any warnings reported by the compiler and rule violations reported by the static analysis tools as errors, the number of warnings/rule violations should be reported to the **NZWP tracker**[3], which then reports pass/fail based on the best case so far for that specific combination of *project, source file*[4]*, tool and rule*.

[3] NZWP = Non-zero Warnings Policy

[4] Should that be omitted?

I believe I have shown that it **is** feasible to introduce static analysis tools to mature software projects, even if it isn't as simple as using them from day one.

This means that one can consider adding new static analysis tools to the analysis suite of existing projects as one becomes aware of them, without worrying about an infeasible start-up cost.

Contact

Jacob Sparre Andersen
JSA Research & Innovation
`jacob@jacob-sparre.dk`
`http://www.jacob-sparre.dk/`

Jacob Sparre Andersen
+45 21 49 08 04