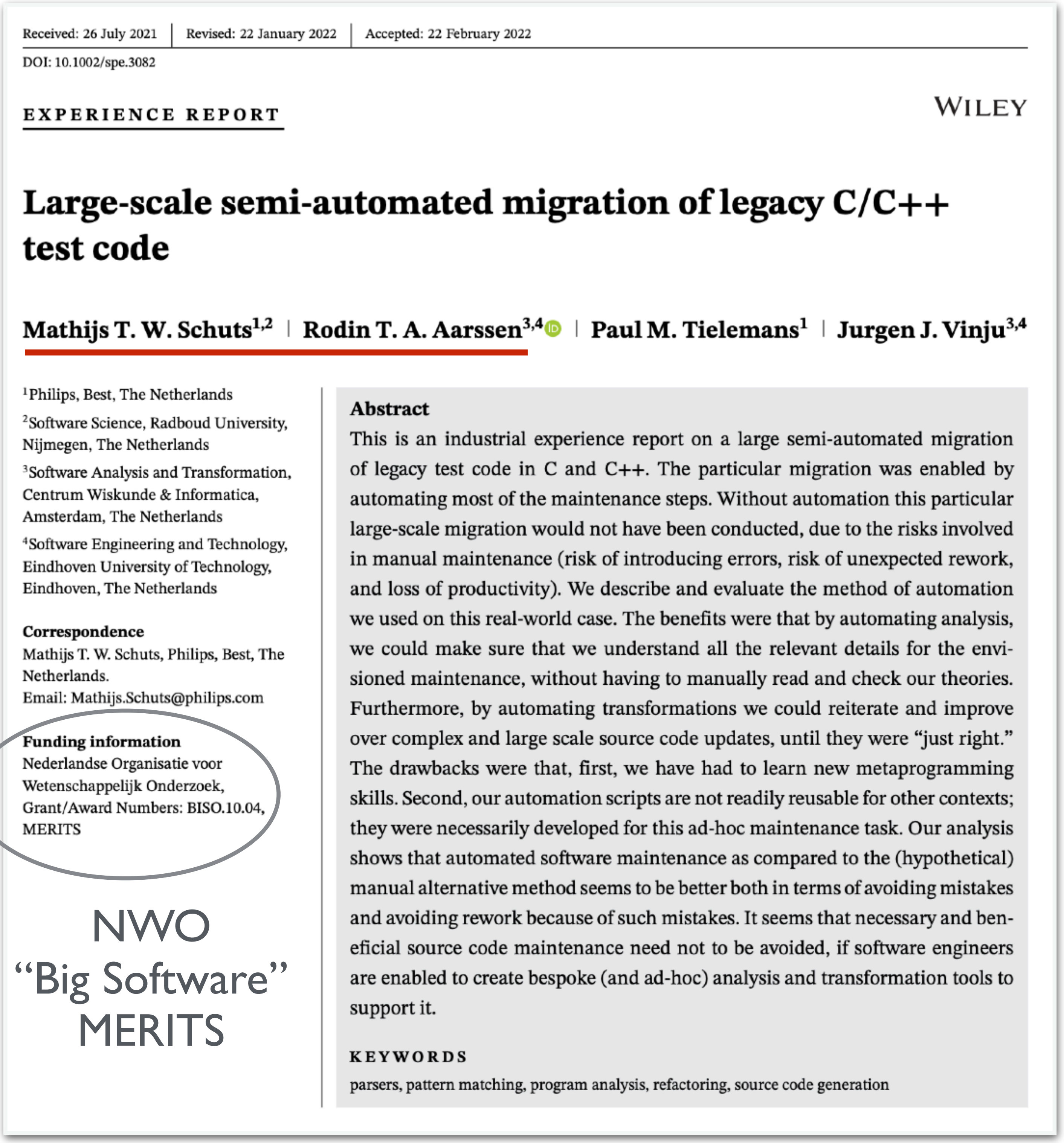
Automating maintenance: the way out of the software renovation paradox

Jurgen J. Vinju



Received: 26 July 2021 | Revised: 22 January 2022 | Accepted: 22 February 2022

DOI: 10.1002/spe.3082

EXPERIENCE REPORT

WILEY

Large-scale semi-automated migration of legacy C/C++ test code

Mathijs T. W. Schuts[1,2] | Rodin T. A. Aarssen[3,4] | Paul M. Tielemans[1] | Jurgen J. Vinju[3,4]

[1]Philips, Best, The Netherlands
[2]Software Science, Radboud University, Nijmegen, The Netherlands
[3]Software Analysis and Transformation, Centrum Wiskunde & Informatica, Amsterdam, The Netherlands
[4]Software Engineering and Technology, Eindhoven University of Technology, Eindhoven, The Netherlands

Correspondence
Mathijs T. W. Schuts, Philips, Best, The Netherlands.
Email: Mathijs.Schuts@philips.com

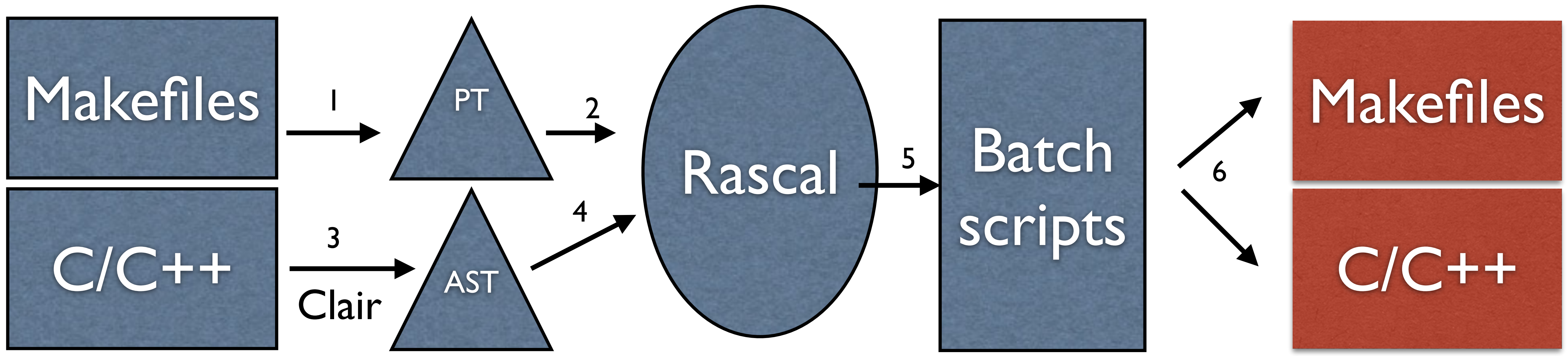
Funding information
Nederlandse Organisatie voor Wetenschappelijk Onderzoek, Grant/Award Numbers: BISO.10.04, MERITS

NWO
“Big Software”
MERITS

Abstract
This is an industrial experience report on a large semi-automated migration of legacy test code in C and C++. The particular migration was enabled by automating most of the maintenance steps. Without automation this particular large-scale migration would not have been conducted, due to the risks involved in manual maintenance (risk of introducing errors, risk of unexpected rework, and loss of productivity). We describe and evaluate the method of automation we used on this real-world case. The benefits were that by automating analysis, we could make sure that we understand all the relevant details for the envisioned maintenance, without having to manually read and check our theories. Furthermore, by automating transformations we could reiterate and improve over complex and large scale source code updates, until they were “just right.” The drawbacks were that, first, we have had to learn new metaprogramming skills. Second, our automation scripts are not readily reusable for other contexts; they were necessarily developed for this ad-hoc maintenance task. Our analysis shows that automated software maintenance as compared to the (hypothetical) manual alternative method seems to be better both in terms of avoiding mistakes and avoiding rework because of such mistakes. It seems that necessary and beneficial source code maintenance need not to be avoided, if software engineers are enabled to create bespoke (and ad-hoc) analysis and transformation tools to support it.

KEYWORDS
parsers, pattern matching, program analysis, refactoring, source code generation

A case of automated renovation at Philips



Philips high-tech interventional X-ray system for the diagnosis and treatment of cardiovascular diseases.

1 programmer

millions of lines of C++ code

move from own test library to google test library

5,840 surgical source code changes in millions

1 line Rascal per 40 lines of C++ changed

1 ad-hoc parser for CMake files

patching “unpreprocessed” code

generated batch scripts for running tests and committing changes

Software Maintenance

"changing source code after the initial release"

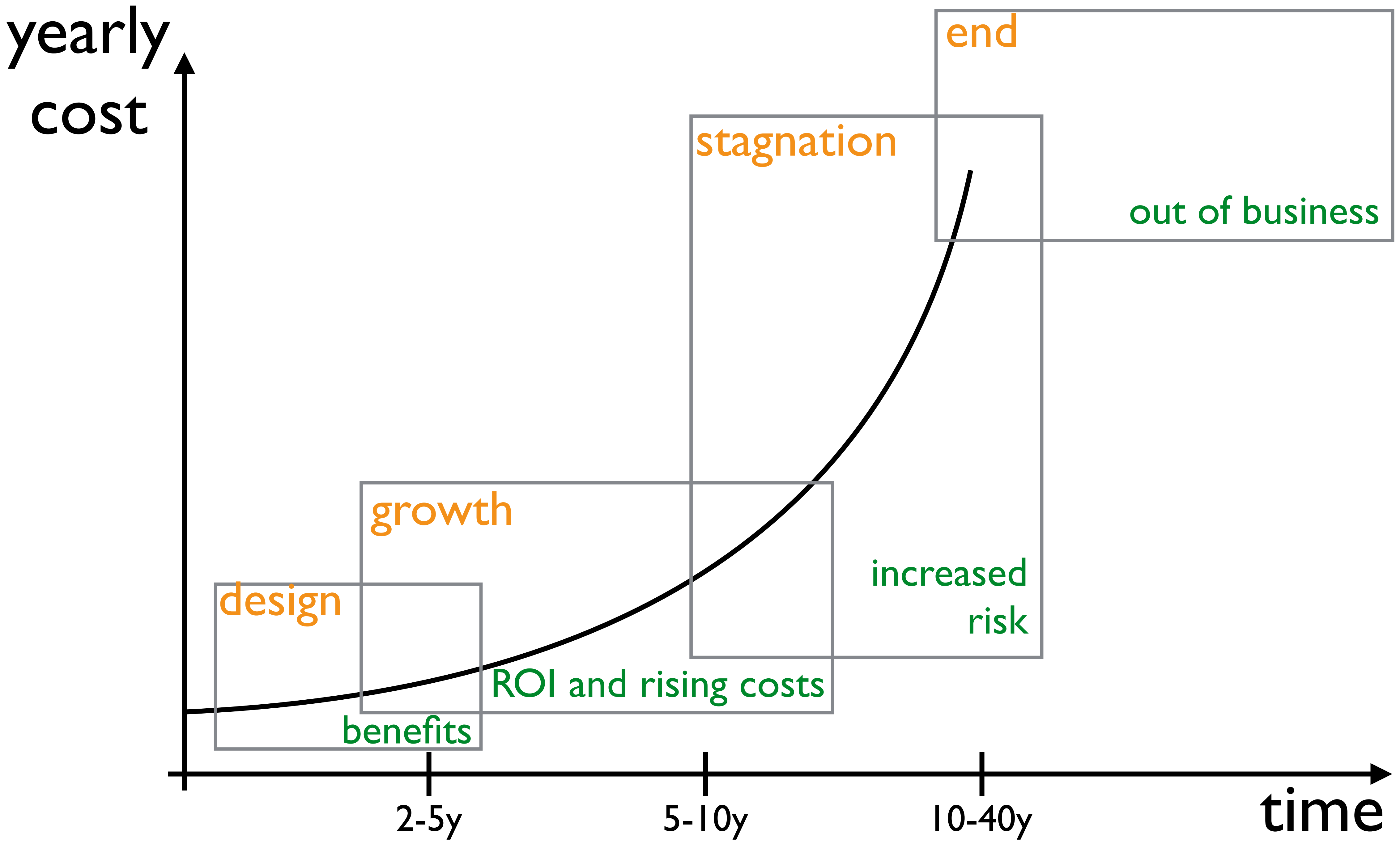
Engineering

- **Design** stage
  - System architecture is *designed*
  - Reversible design decisions
  - Short term successes
  - Testing is easy
- **Growth** stage
  - Incremental additions and corrections *grow*
  - Misconceptions and haste lead to design erosion
  - Co-evolution: changes become scattered
  - "Accidental" code, when it works -> commit
  - Testing becomes cumbersome
- **Stagnation** stage
  - Changes break the system; increasing focus on analysis
  - Working on bugs rather than features
  - "How did this ever work"?
  - Critical reading pushes out (creative) writing

Business

- Early **Benefits** of software ownership
  - Tactical advantage: fast time-to-market
  - Short horizons
  - Incremental costs
- ROI, growing **Cost** of software ownership
  - Lower margins over time
  - Increasing maintenance costs
  - Cost of replacement out-weighs the ROI
- Inevitable **Risks** of software ownership
  - Software becomes cause of stagnation
  - Employee turn-over rate too high
  - Cost of maintenance outweighs total value

The cost of success



Maintaining Code: **Necessary** and Challenging

certify GDPR compliance

add WWW interface

add live user feedback

fix performance bottleneck for peek user loads

upgrade to Windows 10 from Windows 95

scale to {giga,tera,peta}byte/s throughput

integrate 3D simulation

add live user feedback

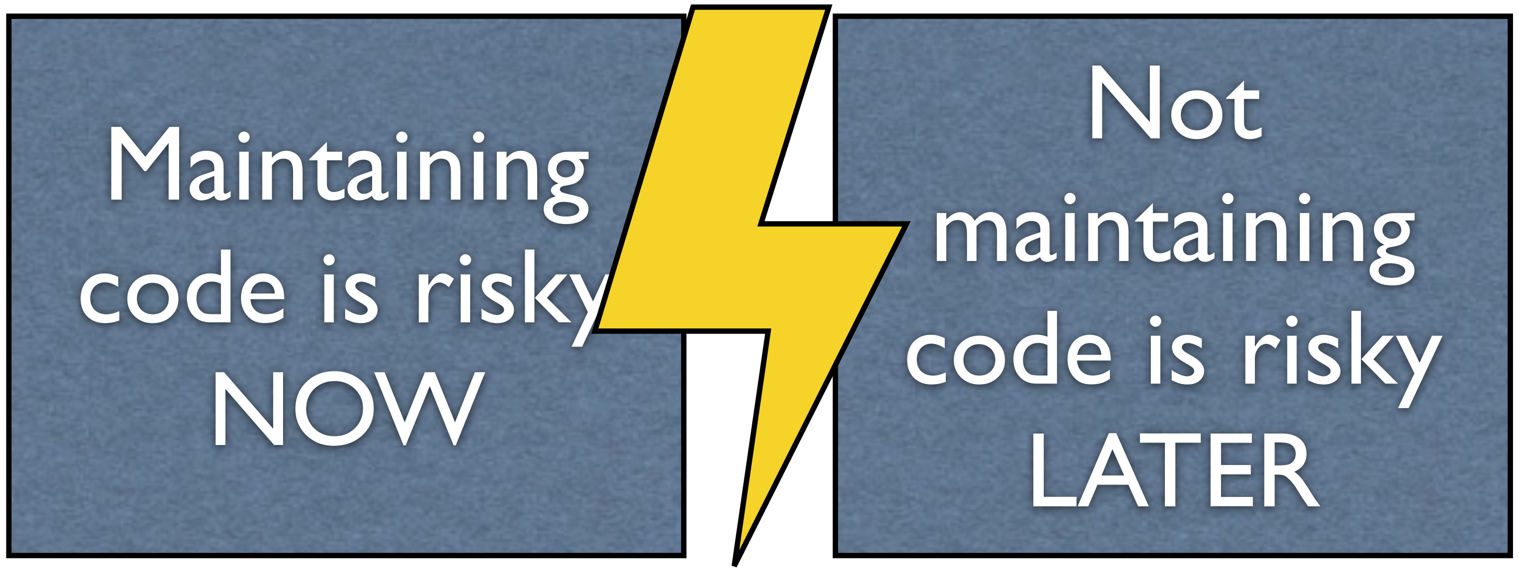
merge this acquired software "stack" into our own, with backward compatibility

port to Linux

switch to ARM

Not maintaining code: recipe for stagnation

- Loss of market potential (no innovation)
- Loss of strategic economic benefits (no flexibility)
- Loss of expertise (people move elsewhere)


Maintaining code is risky NOW
Not maintaining code is risky LATER

Maintenance Paradox

Never change a working component

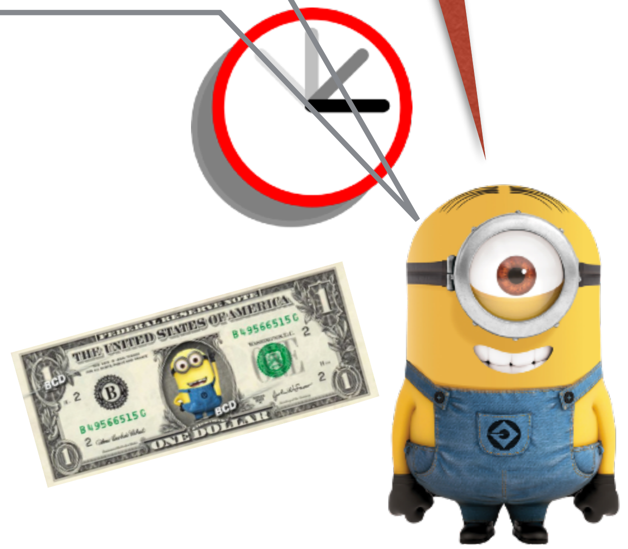
leads to more growth and technical debt

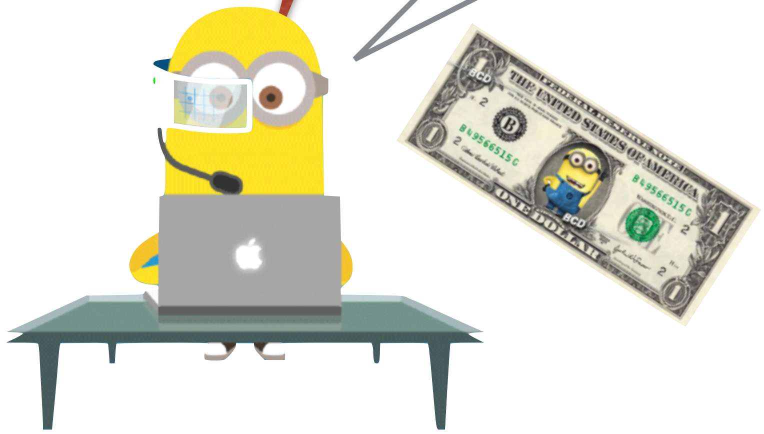
Always adapt to changing circumstances

leads to more instability

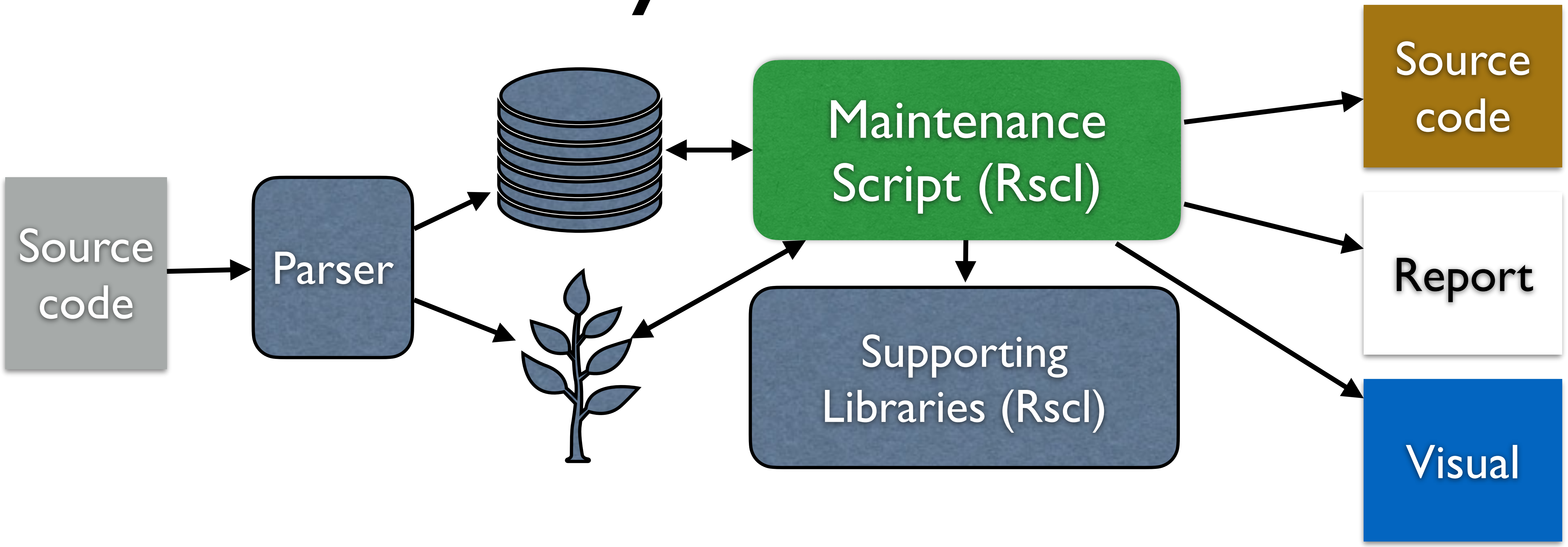
no time for new bugs

have to align to new tech and new business



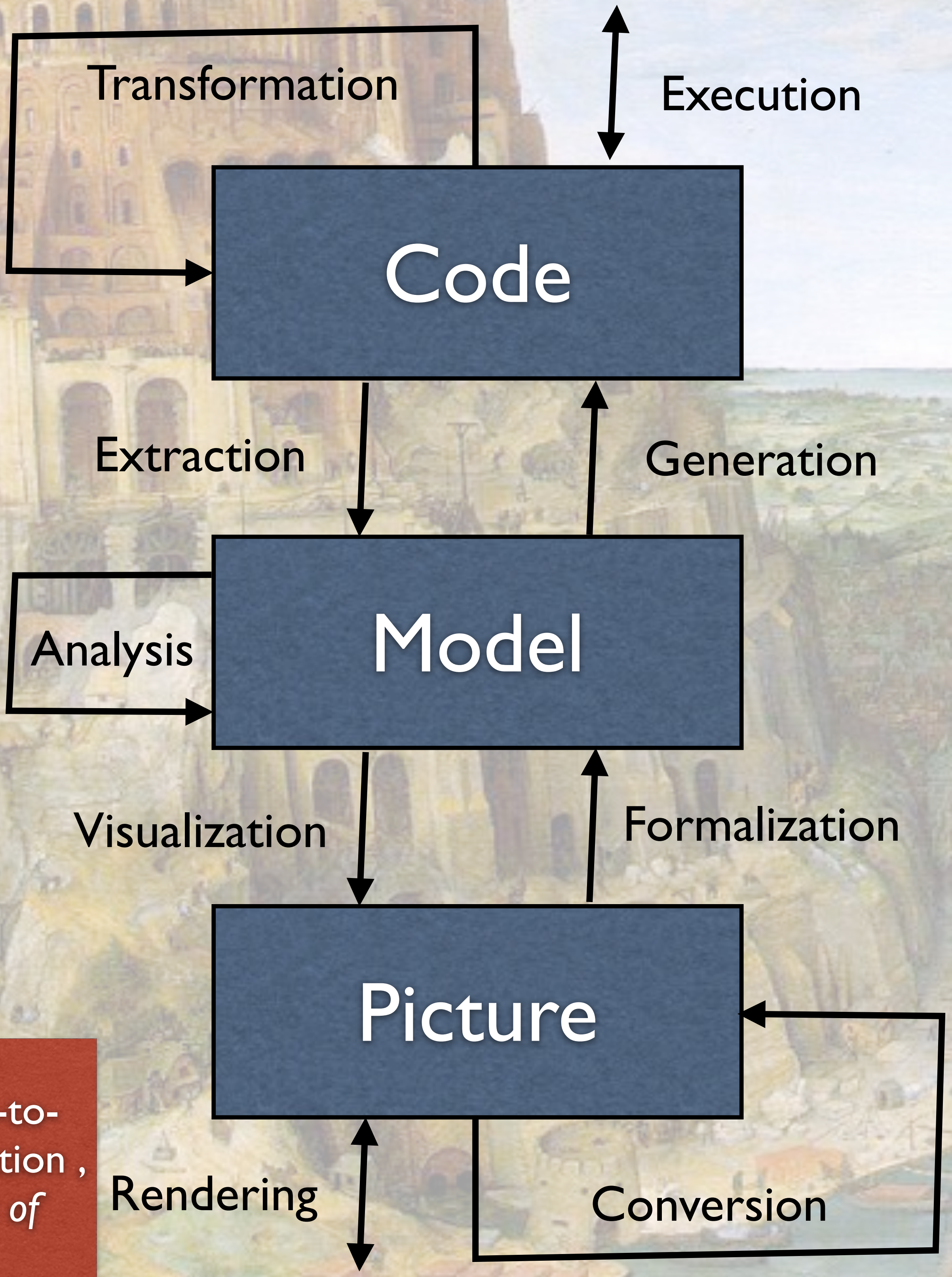


A way out: Rascal



- **Reuse** parsers, name/type analysis, pattern matching, tree traversal, relational queries, origin tracking, …
- **Script** “little” maintenance tasks in Rascal
  - High-level programming (meta-programming)
  - Automate the repetitive/boring parts
  - Avoid “inhumane” manual code analysis and transformation
  - Motivating for the maintenance experts!

Rascal
is a
"language
parametric"
DSL
for
meta
programming



We use Rascal for *all kinds of analysis and design tasks*: model-to-code, model-to-model, code-to-model, code-to-code, verification , static analysis, visualisation, model checking, and on *all kinds of languages* (programming, modelling, data, visual, textual).

(Brueghel, Tower of Babel)

Open-source Community

- Since 2008/2009 http://www.usethesource.io, http://github.com/usethesource
- **Academic/Educational**: Centrum Wiskunde & Informatica, Universiteit van Amsterdam, Vrije Universiteit, Hogeschool van Amsterdam, Open Universiteit, TU Eindhoven, Rijksuniversiteit Groningen, East Carolina University, Bergen University, and many more
  - Empirical study of software projects; software evolution, reverse engineering
  - Design and implementation of Domain Specific (Modeling) languages
- **Industrial collaborations**: SIG, Philips, ING Nederland, Canon
  - Automated code maintenance, Reverse Engineering
  - Domein Specific Languages and Model Driven Engineering
- **Online**: 406 stackoverflow questions, 317 github stars, 25 developers
- 100% Rascal shop: SWAT.engineering BV

Existing Front-ends

- Java code, JVM bytecode
- C/C++ code (**“Clair” Rodin Aarssen**)
- Lua source code
- PHP source code
- Python source code
- Javascript source code
- Ada source code <under development> (ESI)
- XML
- JSON
- CSV
- JDBC
- .NET assemblies
- C# source code
- Jupyter Notebooks
- Halide C++ syntax
- Oberon: full compiler
- … context-free grammars for building your own front-end …

Lessons learned

5,840 surgical source code changes in millions

1 line Rascal per 40 lines of C++ changed

1 ad-hoc parser for CMake files

patching “unpreprocessed” code

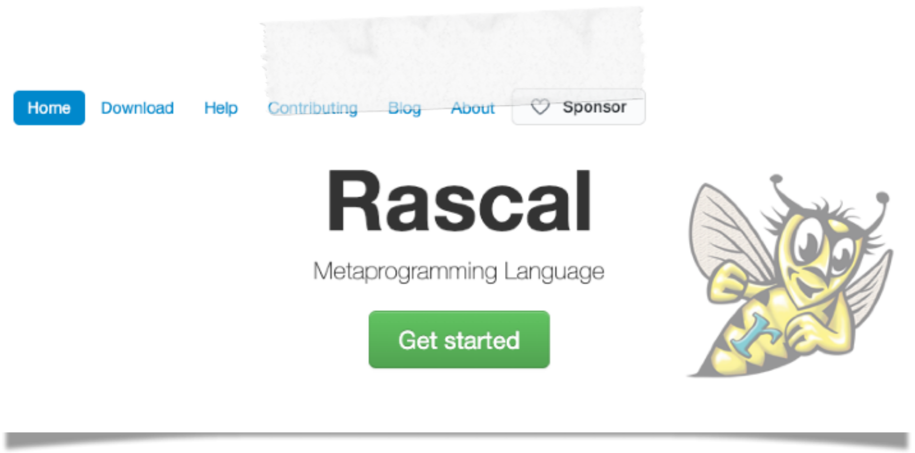
generated batch scripts for running tests and committing changes

- Rascal enables easy maintenance scripting
- Existing Philips **regression tests** are very important
- C/C++ preprocessor problems can be avoided
- Always more kinds input files then expected (CMake)
- AST origin tracking of Rascal is a key enabler
- More in the paper :-)

Take aways

1. Short-term vs long-term priorities clash in software maintenance
2. Automating maintenance with Rascal is a way out
3. Scripting maintenance is a motivating task (think **retention**!)

Mathijs.Schuts@philips.com Rodin.Aarssen@swat.engineering Jurgen.Vinju@cwi.nl



http://www.rascal-mpl.org

http://www.swat.engineering