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

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

1 | INTRODUCTION

The software of complex high-tech systems consists of many interacting components,[1] which typically evolve (more-or-less) independently. The components are members of so-called "product families"; versions of each component are used to compose different integrated product versions which are deployed at customer sites.[2] Typically, successful product families last many years as their accumulated values grows. It is not unheard of that a high-tech product family grows to millions of lines of code developed during several decades.

Although a successful product family such as described above will live for a long time, its components may live shorter lives. Components may become obsolete (unused), or their maintainability has deteriorated over the years (accumulated technical debt) such that replacement or rejuvenation[3,4] is required, or a newer, better component has arrived which can replace the "legacy" component. From the perspective of the entire product family, replacing the old component with the new component is called a "refactoring." A refactoring in general is a semantics-preserving code change. The system changes, but its observable behavior from the outside does not change.

A software maintenance paradox

The question we address in this article is how to replace an existing component with a new one, given a complex product family ("legacy system") that depends on the old component in unforeseen ways. Performing such maintenance on a legacy system is very labor-intensive. The internal details of a system can only be read from the code, as are the complex interactions between subsystems, demanding a significant learning effort from a maintenance engineer. Furthermore, manual large-scale maintenance tends to be error-prone: due to the sheer size of the task at hand, it is likely some cases are missed, and due to the frequently occurring repetitiveness, accidental errors are easily introduced. For software maintainers, it is extremely difficult to guarantee that their changes are indeed a "refactoring." For these reasons, among others, responsible stakeholders are often hesitant to approve preventive maintenance replacing legacy components. Since significant development effort is required to perform such a replacement, and no new features are introduced in the system, the return-on-investment seems low, while the risk-of-failure seems high. In other words, necessary maintenance on the product family is not done to avoid risks, while not doing the maintenance also entails big risks. We are stuck.

A case of API migration

In this article, we take an "industry-as-lab" approach.[5] We describe a real case of a component replacement refactoring carried out at Philips. A legacy library component for unit testing was replaced with a modern library for unit testing. The quality of the testing code that uses the legacy library is absolutely critical to the business; if tests would be lost or their effectiveness in detecting errors reduced, this would entail a commercial risk to all of our stakeholders. In our organization, passing all the automated tests is one of the (many) required quality gates for every code change to the product family, including changes to the tests themselves.

Thus, it is our task to correctly and completely migrate the code that uses the API of the legacy library to using the API of the modern library. Furthermore, all references of said legacy library in build automation scripts and other software artifacts must be removed or replaced by references to the new library. To avoid the inherent risks of manual maintenance, all changes to existing code were automated using metaprogramming scripts. The current article reports on our experiences with automating this API migration in C and C++ code, and the surrounding build artifacts.

The process of API migration

Figure 1 describes an iterative process for an API migration. On the left a hypothetical manual process is depicted and on the right our semi-automated process is depicted. Note that our starting point is a version (snapshot) of the source code of a system that is passing (and has passed) all existing quality assurance gates.

Step 0 In this (optional) step, the codebase can be partitioned into smaller chunks at the metaprogrammer's discretion. In principle, there is no limit to the size of the object software system for either process variant. However, our mandatory formal code review policy (cf. **Step 5**) requires that an independent colleague manually check all code changes; the task of having to do this for the complete codebase at once would simply be too large.

Step 1 The system under study comprises millions of lines of code and in principle any part of the system could depend on the legacy library. Next to code dependencies, references to the library also appear in other software artifacts, such

as makefiles and configuration scripts. So, every iteration starts with an analysis which answers the questions: (a) where are dependencies on the old library and (b) how would we update these dependencies to the new library in principle? Is there a common change pattern that we can apply?

Step 2 Based on the knowledge acquired in **Step 1**, we start changing code. In the manual process this is done file-by-file, while in the automated process we first write a script that captures the knowledge acquired as an executable metaprogram. This difference has two effects. First, running the automated script scales much more easily to be applied to hundreds or even thousands of files without additional manual effort, so this is a great time saver. Second, we can now rerun improved versions of the maintenance scripts again and again based on improved insights, without damage to our efficiency. This leads to improved consistency and thus correctness and understandability, as compared to the manual process.

Step 3 We now (automatically) compile the entire refactored system and run *all* of the test suites of the product family using the existing build and test infrastructure.

Step 4 Failing tests are triaged and diagnosed manually. In the manual case, we would try and run the new tests on a few of the files first, before having refactored the entire system. However, the system is probably not organized in a way as to allow such individual tests to run, so this comes at a cost of changing makefiles and isolating the tests of (parts of) components. The automated refactoring is run on the entire system and we simply continue our analysis where the first errors start to appear. Note that it can be assumed that any failing tests are due to our latest changes per our starting assumption mentioned above.

Step 5 The next quality gate to pass is a pre-delivery manual code review by an independent colleague. If this fails, we go back to the drawing board; if it succeeds, the code is accepted into the main branch of the version control system. The new code will eventually undergo manual and automated integration tests and pass many other quality gates until it reaches (different) product deployments. This is outside our scope and influence. The entire process only stops when all tests are succeeding and all traces of the old library have disappeared from the source code and from other software artifacts.

For the sake of clarity, we repeat here: the manual process was not actually conducted and without an automated alternative this particular API would not have been migrated.

Roadmap

The remainder of this article is organized as follows. In Section 2, we will introduce the details of the case study. The metaprogramming skills we needed to automate the maintenance steps are described in Section 3. It can be read as a quick introduction into the RASCAL metaprogramming language. Then, in Section 4, we will detail all of the maintenance scripts we have developed for the case study. In Section 5, we will reflect back on our experiences, zoom out of this specific case and discuss related work, before concluding in Section 6.

We hope that others who are stuck in a similar situation, where high-risk maintenance on a product family is necessary, can learn from the experiences we describe here and become unstuck like we did.

2 | PROBLEM ANALYSIS: MIGRATING A LEGACY C++ TEST API TO GOOGLETEST

The software components we analyze and manipulate for this case control a Philips high-tech interventional X-ray system for the diagnosis and treatment of cardiovascular diseases. The software controls the collaborating machines in an operating theater, where X-rays are used both for live imaging features as well as for physical interventions. Typically there are several human operators at work at the same time.

The software controlling these devices consists of millions of lines of C and C++ code. For this case, we are focusing on the subsystem (a collection of components) that is responsible for the positioning of the X-ray beam with respect to the patient. It controls the motors that move robot arms as well as the patient support table. This "Positioning Subsystem" comprises well over half a million of source lines of code (SLOC, i.e., non-empty and non-commented lines of source code).

Before the migration, the codebase contains two separate testing frameworks: GOOGLETEST is used for younger components, while the older components are tested using a much older proprietary framework called Simple Test eXecutor (STX). The frameworks have slightly different API and run-time semantics, which may lead to future confusion, and STX itself also requires maintenance which seems redundant now that an open-source alternative exists. Migrating away from STX in favor of GOOGLETEST should improve the quality of the tests and allow us to focus on maintaining other code than STX itself. The fact that GOOGLETEST is a lively open-source project is also considered beneficial, since it will allow us to surf on improvements and extensions without having to maintain our own testing framework. The fact that GOOGLETEST is open-source is also essential for our exit strategy: in case new releases of the GOOGLETEST project should become unstable or unreliable we would be able to fork an earlier stable snapshot and fall back to our previous strategy of maintaining our own testing framework.

However, the task of migrating away from STX is substantial: the codebase contains over 150 STX test suites, each ranging from a few hundreds to several thousands of lines of code. Not surprisingly, the application programming interfaces (APIs) of STX and GOOGLETEST do not match. This impacts *client code*, the code that is written against the test API. It must be rewritten in order to even use the GOOGLETEST API correctly, and also to benefit from testing and reporting features that are present in GOOGLETEST but not in STX.

Referring to Figure 1, the current section describes the results of the Problem Analysis (**Step 1**). We describe how to make an STX test case in Section 2.1, and show the corresponding intended counterpart in GOOGLETEST in Section 2.2. In Section 2.3, we summarize the required editing steps to migrate STX test suites to GOOGLETEST.

The Positioning Subsystem uses CMAKE to automate the build process. CMAKE is a version of the Unix MAKE command which offers "build rules" to enable the conversion of input files (i.e., source code and libraries) to output files

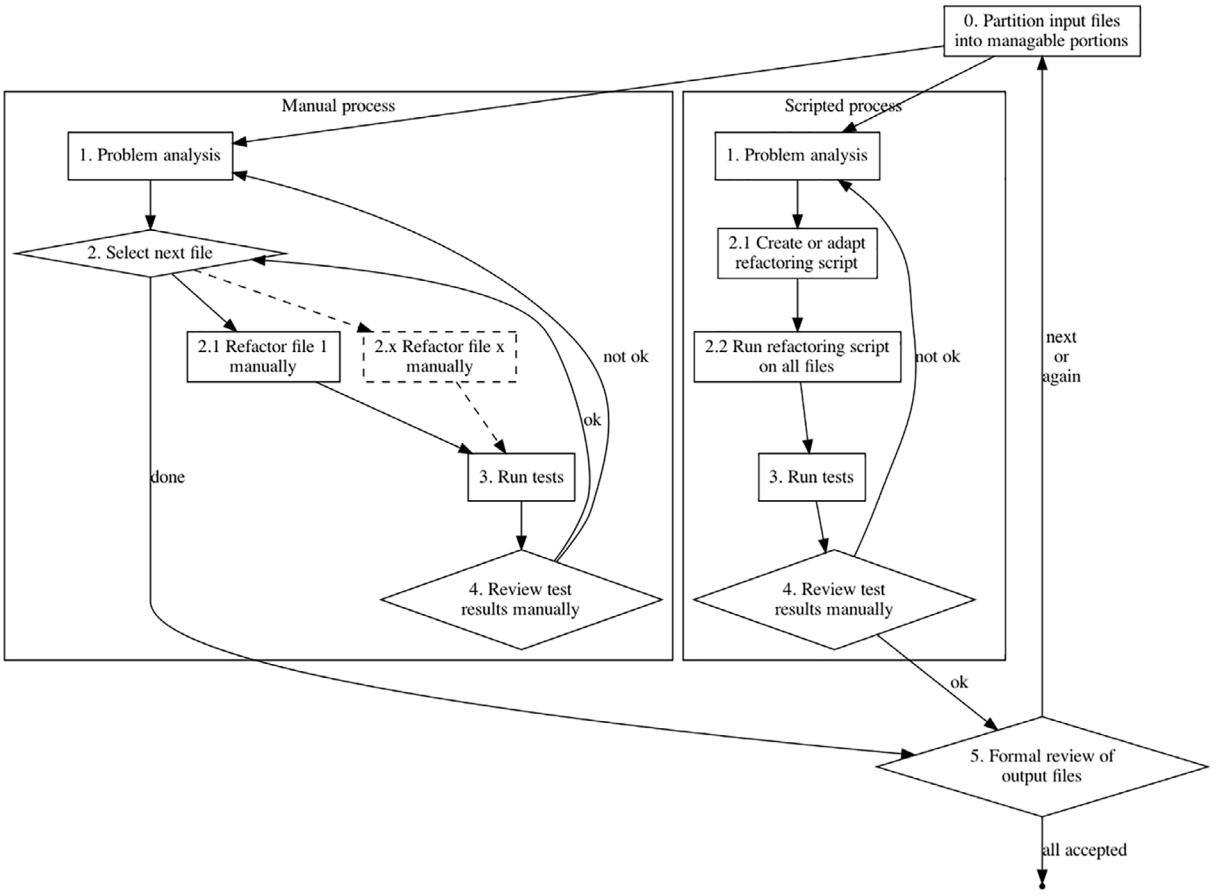

FIGURE 1 Two API migration processes; one manual and one automated, as embedded into the general quality assurance/code review process

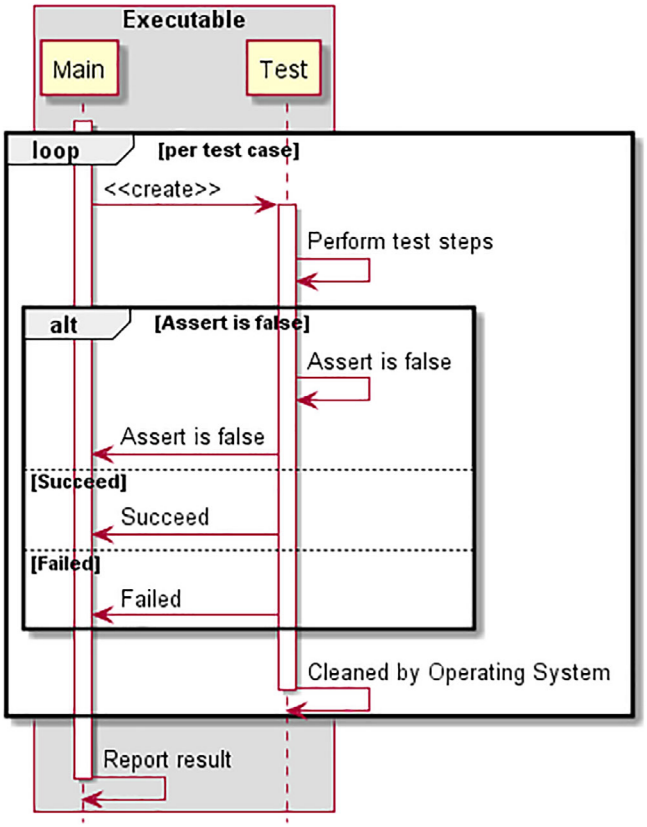

FIGURE 2 UML sequence diagram depicting the lifetime of a single STX test

(binary executables) using compilation and linking steps. As part of the migration, CMAKE files with references to the STX library have to be changed as well. We will show a CMAKE file for STX in Section 2.1, followed by the corresponding CMAKE file for GOOGLETEST in Section 2.2. Finally, in Section 2.3, we summarize how to migrate CMAKE files from STX to GOOGLETEST.

2.1 | The legacy STX testing framework

The STX test framework is several decades old, and is used at Philips to test software units written in the C and C++ programming languages. A single compiled test suite, colloquially referred to as "an STX," is a Windows executable file of which the name ends with `_stx`. Such an executable contains production code, statically linked as libraries to one or more compiled test cases. STX can be used to test both C and C++ code. Figure 2 provides an overview of the execution lifetime of a single STX, according to the following pattern.

1. The main process creates a new process per test case.
   (a) The new process executes the test steps.
   (b) The test case terminates in one of three ways:
      - an assertion in the test code evaluates to false;
      - the test case indicates success; or
      - the test case indicates failure.

      The main process is informed of the outcome through inter-process communication.
   (c) If the current test case terminated successfully, the next test case is started, when applicable.
2. The main process reports the results and terminates.

```
1  #include "STXServer.h"
2  static void example_test_1();
3  static int argc_input;
4  static char **argv_input;
5  void GEN_p_cold_entry() {
6    PP_p_PF_printf("\nRegistering test functions...\n");
7    RegisterTestFunction(example_test_1, "Example normal test 1", STX_SHORT_TIMEOUT, Normal);
8    PP_p_PF_printf("\n#Arguments = %d...\n", argc_input);
9    StartTestServer(argc_input, argv_input);
10   exit(⊘);
11 }
12 void example_test_1() {
13   t_dword  coreleft_before_test;
14   t_dword  coreleft_after_test;
15   Example_p_TM_init();
16   coreleft_before_test = TOS_p_SEG_coreleft();
17   Example_p_TM_test();
18   coreleft_after_test = TOS_p_SEG_coreleft();
19   GEN_m_assert(coreleft_before_test == coreleft_after_test);
20   NotifyTestPassed();
21 }
22 int main(int argc, char *argv[]) {
23   argc_input = argc;
24   argv_input = argv;
25   TOS_p_STP_start_continue_set(GEN_p_cold_entry);
26   TOS_p_STP_boot();
27   return ⊘;
28 }
```

Listing 1: Example STX test suite, containing a single STX test case

Listing 1 depicts an example test suite. The `main` function registers the `GEN_p_cold_entry` function and boots the runtime environment. The functions on line 7 and line 9 are part of the STX framework. The `GEN_p_cold_entry` function registers one test case called `example_test_1` (line 7). The `StartTestServer` function creates a process to execute the test case (cf. step 1). The test case performs some initializations (line 15), after which it stores the current memory usage (line 16), executes the test function (line 17), and stores the memory usage a second time (line 18). The `GEN_m_assert` assertion macro checks that the memory usage has not changed by executing the test steps (line 19). If this is indeed the case, the test process informs the main process that it has succeeded (line 20).

When an assertion is violated, the file name, line number, and a stack dump are written to a log file. In addition, the test framework is informed of the assertion failure, and the executable exits with a nonzero exit status. The call to `exit` on line 10 is only reached after successful termination of all test cases.

Test cases report on their successful or unsuccessful outcome to the main process thought the `NotifyTestPassed` and `NotifyTestFailed` functions.

To compile a STX test suite there are specific CMAKE rules written for each STX. Listing 2 shows a CMAKE file for a hypothetical `example_stx` file. In this script, the following variables are set:

- The target name of the build is set to `example_stx`.
- The folder in which the project is accessible in the Microsoft Visual Studio IDE is stored in `IDE_FOLDER`.
- The `DIRS` variable is assigned additional include directories, in particular the `STXServerLib` headers directory.
- The `SOURCES` gets a list of source files that are to be compiled. In this example, there is a single C file `example_stx.c`.
- Production libraries are specified in the `DEPS` variable.

Finally, the user-defined `configureTestExecutable` macro adds test-specific dependencies to the `DEPS` variable, and sets the output location of the STX executable.

```
# example_stx
set(TARGET_NAME example_stx)
set(${TARGET_NAME}_IDE_FOLDER "PosGen/bb/Test")
set(${TARGET_NAME}_DIRS
    ${POS_GEN_PATH}/bb/inc
    ${POS_TEST_PATH}/STX/STXServerLib/src
)
set(${TARGET_NAME}_SOURCES example_stx.c)
set(${TARGET_NAME}_DEPS ${COMMON_STX_DEPS})
configureTestExecutable(${TARGET_NAME} OBJ_TN)
```

Listing 2: Example CMAKE file for STX, containing build information for a hypothetical `example_stx` test file

2.2 | GOOGLETEST framework

GOOGLETEST is a unit testing framework, developed by Google. Figure 3 depicts an overview of the GOOGLETEST approach. A test suite runs one or more test cases, according to the following pattern.

1. The main process sequentially executes the test cases. For each test case, the test steps are executed. A test case can either succeed or fail.
2. The main process reports the test results and exits.

Note that GOOGLETEST always executes all registered test cases, regardless of the outcome of previous test cases. Because our starting assumption is that all STX tests succeed, this does not entail an observable change in the semantics of the test system. However, when we introduce a change in the system that would impact several tests, the GOOGLETEST

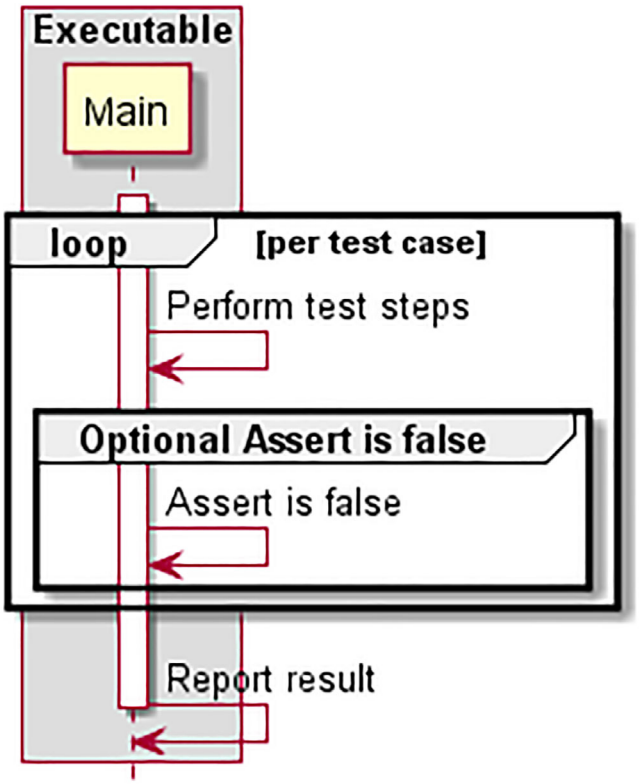

FIGURE 3 UML sequence diagram depicting the lifetime of a single GOOGLETEST test

framework is able to report all failing tests at once while STX could have hidden failing tests behind a failing test. This is considered to be another advantage of introducing the GOOGLETEST framework, since it makes diagnosing new failures easier by providing more complete information sooner.

```
1  #include "SetupTestDependencies.h"
2  static int argc_input;
3  static char **argv_input;
4  class ExampleStx : public SetupTestDependencies {};
5  void GEN_p_cold_entry() {
6    PP_p_PF_printf("\n#Arguments = %d...\n", argc_input);
7    int testResult = SetupTestDependencies::GoogleTest(argc_input, argv_input);
8    exit(testResult);
9  }
10 TEST_F(ExampleStx, example_test_1) {
11   t_dword  coreleft_before_test;
12   t_dword  coreleft_after_test;
13   Example_p_TM_init();
14   coreleft_before_test = TOS_p_SEG_coreleft();
15   Example_p_TM_test();
16   coreleft_after_test = TOS_p_SEG_coreleft();
17   ASSERT_EQ(coreleft_before_test, coreleft_after_test);
18 }
19 int main(int argc, char *argv[]) {
20   argc_input = argc;
21   argv_input = argv;
22   if (SetupTestDependencies::IsListArgumentSpecified(argc_input, argv_input)) {
23     return SetupTestDependencies::GoogleTest(argc_input, argv_input);
24   }
25   TOS_p_STP_start_continue_set(GEN_p_cold_entry);
26   TOS_p_STP_boot();
27   return ∅;
28 }
```

Listing 3: Example GOOGLETEST test suite, containing a single GOOGLETEST test case. This is the intended counterpart of the STX example of Listing 1

GOOGLETEST can be used to test code written in C++ only. Listing 3 shows the same test case as Listing 1, now implemented using GOOGLETEST, instead of STX.

The `TOS_p_STP_boot` function prints boot strings to screen before GOOGLETEST is called in the `GEN_p_cold_entry` function. The scripting we use to execute the test suites on offload systems requires a clean output of the `-gtest_list_tests` command-line argument. For this reason, the `main` function first checks on line 22 whether this flag is specified. If so, booting is aborted, and instead the output this flag generates is printed to screen by evaluating the return argument on the next line. In absence of this flag, the `main` function registers the `GEN_p_cold_entry` function (line 25) and boots the run-time environment (line 26). At line 4 the `ExampleStx` class is defined. It inherits from `SetupTestDependencies`, which takes care of starting and stopping test dependencies. The test cases are started by GOOGLETEST on line 7. The test case is defined on line 10 and line 17 contains an assert to check whether `coreleft_before_test` is equal to `coreleft_after_test`. GOOGLETEST then prints the expected and actual values.

```
1  # example_stx
2  set(TARGET_NAME example_stx)
3  set(${TARGET_NAME}_IDE_FOLDER "PosGen/bb/Test")
4  set(${TARGET_NAME}_DIRS
```

```
5       ${POS_GEN_PATH}/bb/inc
6       ${GTEST_INCLUDE_DIR}
7       ${GTESTSETUP_INCLUDE_DIR}
8       ${POS_GEN_PATH}
9   )
10  set(${TARGET_NAME}_SOURCES example_stx.cpp)
11  set(${TARGET_NAME}_DEPS ${COMMON_STX_DEPS})
12  configureUnitTestExecutable(${TARGET_NAME} OBJ_TN)
```

Listing 4: Example CMAKE file for GOOGLETEST, containing build information for a hypothetical `example_stx` test file. This is the intended counterpart of the STX example of Listing 2

To compile a GOOGLETEST test suite also the CMAKE file is slightly different. Listing 4 shows the GOOGLETEST equivalent of the STX CMAKE file in Listing 2. The scripts are very similar, differing in the `DIRS`variable—GOOGLETEST directories are included instead of STX directories. Also, the `SOURCES`variable now contains a C++ file. Finally, the last line has a different user-defined macro `configureUnitTestExecutable` that, next to adding test-specific dependencies, sets the output location for the GOOGLETEST executable.

2.3 | Refactoring test suites from STX to GOOGLETEST

In this section, the required changes to refactor test suites from STX to GOOGLETEST are introduced.

Change 1: C to C++

The GOOGLETEST framework only supports C++. Hence, this requires test suites written in C to be converted into C++. Since these new C++ files may still import C headers containing production code, include directives for such headers must be encapsulated in an `extern "C"` block to avoid name mangling issues. Since most of our test suites do not have a header file, we define the test class in the source file instead (cf. Listing 3, line 4). Finally, we change the extension of C files from `.c` to `.cpp`.

The C++ compiler is more strict than its C counterpart. Test code that previously compiled successfully with the C compiler, may therefore introduce warnings at compile time with the C++ compiler.* Being in a safety-critical domain, our build pipeline instructs the compiler to treat warnings as errors. Because of this, all new warnings and errors that are introduced by the change from C to C++ need to be resolved.

Change 2: Replace asserts

STX uses the `GEN_m_assert` macro to compare the actual value of some variable to an expected value. Occurrences of these macros in the test code need to be replaced with their GOOGLETEST counterparts.

In this refactoring, we also want to improve the reporting of failing assertions. In STX, the equality of two values is checked with the `==` operator (Listing 1, line 19). When the comparison expression evaluates to `false`, it is only reported that the values were unequal. To obtain the actual values, a developer has to attach a debugger to the test executable. In GOOGLETEST, the expected and actual values are passed to the framework (Listing 3, line 17). This way, GOOGLETEST can report the values, in case an assertion fails, removing the need to attach a debugger to the executable. We have to detect the use of the `==` and `!=` operators under the `GEN_m_assert` macro and replace it with calls to `ASSERT_EQ` and `ASSERT_NE`, respectively.

*For example, while a cast to `void*` is perfectly valid for the C compiler, the C++ compiler produces a warning for such a construct.

Change 3: Replace header inclusion

Our test suites include a specific header file, depending on the testing framework being used. The inclusion of the `"STXServer.h"` header file needs to be replaced by an inclusion of the `"SetupTestDependencies.h"` header.

Change 4: Rewrite test function

In an STX test suite, functions containing test code are defined using a regular function prototype. GOOGLETEST comes with the `TEST_F` macro that, besides expanding to a function prototype, registers this function in the GOOGLETEST runtime implicitly. The macro takes as its two arguments the name of the test class and the name of the test function.

Change 5: Refactor entry function

In STX, the entry function contains a print statement indicating test functions are being registered, followed by calls to the registration function (Listing 1, lines 6 and 7). These lines are to be removed, since test function registration is implicit in GOOGLETEST. In addition, the STX call to `StartTestServer` is to be replaced by a call to `GoogleTest` (Listing 3, line 7). Finally, the result of this call is passed as the exit status (Listing 3, line 8).

Change 6: Rewrite reporting of test verdict

An STX test case signals its outcome by calling `NotifyTestPassed` or `NotifyTestFailed` for success or failure, respectively. Calls to the latter are to be replaced by GOOGLETEST's `FAIL` macro. In GOOGLETEST, test cases are successful implicitly when no assertion violations occur; calls to `NotifyTestPassed` therefore have to be removed.

Change 7: Rewrite main function

The `main` function needs to get an `if` statement with the `–gtest_list_tests` check. For the true condition the `GoogleTest` function, and for the false condition the `TOS_p_STP_start_continue_set` and `TOS_p_STP_boot` functions are printed.

Change 8: Rewrite CMAKE files

To refactor CMAKE files from STX to GOOGLETEST, the following three changes are required. First, as GOOGLETEST only supports C++, all C source files are changed to C++. To reflect this change in our CMAKE files, the extension of C files has to be changed to `.cpp`. Second, the STX include directories have to be replaced by the GOOGLETEST include directories. Finally, the user-defined `configureTestExecutable` macro, belonging to STX, needs to be substituted with `configureUnitTestExecutable` for GOOGLETEST.

2.4 | Conclusion

This concludes a rough informal analysis of what needs to be done to the source code of the Positioning Subsystem to be able to migrate from STX to GOOGLETEST. The next step in our process (**Step 2** in Figure 1) is to start automating these eight changes. The automation scripts are detailed in Section 4, but first we must introduce the necessary features of the metaprogramming language we used in Section 3: RASCAL.

3 | KEY FEATURES OF THE RASCAL METAPROGRAMMING LANGUAGE

The code transformations in this article were implemented in RASCAL, a metaprogramming language and language workbench.[6] In this section, we introduce language features of RASCAL that are needed to understand the remainder of this article. In particular this introduction will enable the reader to assess the code fragments shown in Section 4. For a more complete description of the language, we kindly refer to RASCAL's documentation.[7†] At its core RASCAL is a type-safe programming or scripting language, with immutable data, high-order functions, structured control flow, with builtin pattern matching, search and relational calculus. RASCAL is not an object-oriented programming language. It is a functional, procedural and logical programming language with a Java-like syntax. The RASCAL code fragments in Listing 5 will be explained in the remainder of this section.

```
1  |file:///Users/kees/.bashrc|(100,20,<2,0>,<2,20>)
2  |cpp+class:///MyNamespace/MyCppClass|
3
4  data Boolean = true() | false() | and(Boolean lhs, Boolean rhs); //initial definition
5  data Boolean = or(Boolean lhs, Boolean rhs);                     //extending Boolean with another constructor
6  data Statement = \if(Expression c, Statement tt, Statement ff); //definition of a typical AST node
7
8  layout Whitespace = [\r\n\ ]*;
9  lexical IntLit = lit : [0-9]+ val;
10 start syntax Expr
11   = literal : IntLit i
12   | paren : "(" Expr e ")"
13   | add : Expr lhs "+" Expr rhs;
14
15 (IntLit) `1`
16 (Expr)    `(1+2)`
17 (Expr)    `1  +((    2) )`
18
19 data Expr
20   = literal(IntLit i)
21   | paren(Expr e)
22   | add(Expr lhs, Expr rhs);
23
24 [X, 1, Y] := [1, 1, 2]          //true: X = 1, Y = 2
25 [X, 1, X] := [1, 1, 2]          //false
26 [*X, 1, *Y] := [1, 1, 2]      //true: X = [1], Y = [2] or X = [], Y = [1, 2]
27 [*X, 1, *X] := [1, 1, 2]       //false
28 [*X, *X] := [1, 1]             //true: X = [1]
29
30 /at/ := "match"                  //true: "at" is a substring of "match"
31 /at$/ := "match"                 //false: "match" does not end with "at"
32 /AT/i := "match"                 //true: case insensitive match
33 /<as:a*><bs:b*>/ := "aabbb"  //true: as = "aa", bs = "bbb"
34
35 add(literal(0), x)     //an add node with a literal node nested as its first field and a variable x as its
36  second
37 add(literal(0), _)     //use of a wildcard pattern nested under a node pattern
38 a:add(x, x)            //"non-linear" patterns test for equality: the second x should be equal to the first.
39  Variable a is bound on a successful match
40
41 (Expr)`0 + <Expr x>`
42 (Expr)`0 + <Expr \_>`
```

†https://docs.rascal-mpl.org/

```
43 (Expr)`<Expr x> + <Expr x>`
44
45 /paren(paren(\_)) := e                  //finds any directly nested paren expressions in e
46 for (/literal(n) := e) println(n);      //loops over all literal nodes anywhere in e and prints their literal
47 visit (e) {                             //visit is like switch, but finds the patterns anywhere deeply nested
48   case literal(n) : println(n);
49 }
50
51 [1 .. 5]                          //[1, 2, 3, 4]
52 [n*n | n <- [1 .. 5]]             //[1, 4, 9, 16]
53 for (int i <- [1 .. 5]) println(i);
54
55 str s = "This is
56          "a single string literal."
57 str w = "world"
58 println("Hello, <w>!");           //prints "Hello, world!"
```

Listing 5: Extensive RASCAL code fragment, containing all code examples referred to throughout Section 3

Primitive types and locations

RASCAL features: (a) `int`, `real`, and `rat` as numerical types, (b) `bool` for booleans, (c) polymorphic `lists`, `sets`, `maps` for collections, (d) `datetime` for absolute time values, and finally (e) `loc` for location constants. On line 1, there is a constant that points to a file with the file scheme and selects the part on line 2 between the left margin and the 20th column. Line 1 contains a logical reference to a C++ class name as it could be produced by a C++ name analysis stage in its compiler. Locations are used to refer to source files and are frequently kept with the information that was extracted from source files, to help referring back to the source and also to uniquely identify names to avoid confusion.

Algebraic data types

RASCAL supports the definition of user-defined algebraic data types (ADTs) with their constructor functions to create values to inhabit these types. A many-sorted algebraic data type can be used to define the shape of abstract syntax trees or the shapes of other structured symbolic values such as types or constraints. In RASCAL ADT definitions are modularly extensible; an additional definition of an existing algebraic data type will extend the type rather than override it.

The fragment on lines 4–6 shows a declaration of a data type for a representation of Boolean expressions on the first line, using three constructors. The next line contains a declaration for the same data type, effectively adding an alternative to the existing declaration. RASCAL's reserved keywords are not permitted as names of constructors. Since `if` is a reserved keyword in RASCAL, it must be escaped when used as the name of an algebraic constructor: `\if`.

Context-free grammars

Where ADTs are used for abstract syntax trees, context-free grammars are used for concrete syntax trees. RASCAL's built-in context-free grammar notation corresponds in many ways to EBNF. From a grammar definition, RASCAL automatically generates a parser, and supports construction of, and pattern matching on, parse trees and abstract syntax trees over this grammar.

Lines 8–13 show a grammar for a simple expression language. First, any number of new line characters and spaces are declared as whitespace. Then, the `IntLit` nonterminal is defined as any non-empty sequence of digits. Finally, the `Expr` nonterminal is defined with three productions. As the `Expr` nonterminal is declared as `syntax`, rather than `lexical`, the productions are internally augmented to accept layout between symbols. For example, the `add` production

is changed to `add : Expr lhs Whitespace "+" Whitespace Expr rhs`. Similarly, the `Expr` nonterminal is declared as a `start` nonterminal, which adapts the nonterminal to accept layout before the first symbol and after the last.

Concrete syntax expressions are used in RASCAL to construct syntax trees from embedded input strings (lines 15–17). Each expression starts with the nonterminal to use when parsing the following string between backquotes. RASCAL's parser then produces a syntax tree, at compile-time, of which the top type is equal to the given nonterminal. All syntax (sub)trees have a `.src` field, which defines exactly the file and part of the file that the current tree encompasses using a location value (see line 1).

In the example syntax definition on lines 8–13, all productions and symbols are labeled. RASCAL uses these labels to generate implicit abstract syntax ADTs. Such an implicit ADT abstracts away from terminal symbols in the corresponding productions, uses production labels for constructor names, and uses symbol labels for constructor argument names. For example, the ADT defined on lines 19–22 is automatically generated.

Pattern matching

For a large part, metaprogramming is about analyzing syntax trees. Pattern matching is a high-level language feature in RASCAL that helps to avoid writing repetitive nested conditionals and loops which are otherwise necessary to detect patterns in syntax trees. In RASCAL pattern matching surfaces in different parts of the language: switch case distinction, dynamic function dispatch, generators (with the `:=` and `<-` operators), and the `visit` statement for recursive traversal.

So-called "open" patterns bind variable when they match. Some patterns can match in multiple ways; RASCAL uses these multiple solutions as generators that the programmer can use to search (with backtracking) or loop through. For example, the `for` loop loops over all bindings of a pattern while the `if` statement finds a first match that satisfies all conditions.

RASCAL supports pattern matching on all values, including ADTs, parse trees, strings, sets, and lists. Finally, descendant patterns—patterns preceded by a `/`—can be used to match a pattern at arbitrary depth in a value. We will show and discuss some code fragments containing examples of pattern matching in RASCAL.

List and set patterns

The fragment on lines 24–28 illustrates list matching. Set matching works similarly, but the order and duplicity of elements is irrelevant. On a successful match, the (fresh) variables `X` and `Y` are bound to the appropriate subvalues. An asterisk symbol `*` indicates a multi-variable in the context of set and list matching; a variable `*X` thus represents a list or set of values, depending on the context. Nonlinear matching (e.g., on the second, fourth and fifth lines) is supported. The third line is an example of a pattern match with multiple solutions.

Regular expression patterns

Regular expressions are used to match against string values. RASCAL's regular expression language is largely equivalent to the Java Regex language. Regular expression patterns are delimited by slash symbols `/`. As usual, the `^` and `$` characters denote the beginning and end of a line, respectively. Lines 30–33 illustrate matching using regular expressions. A trailing `i`, like on the third line, sets matching mode to case-insensitive. RASCAL allows variables to capture groups of characters, like on the fourth line.

Node patterns

RASCAL performs pattern matching on nodes by comparing the constructor name, then testing the amount of children and then matching the arguments recursively. It is important to know that all pattern types can be nested arbitrarily.

Patterns may be labeled with a (fresh) name; on a successful match, the appropriate (sub)tree is bound to the variable. The fragment on lines 35–37 shows several (nested) patterns.

Concrete syntax matching

Concrete syntax patterns may be used to match against parse trees. Such a pattern uses concrete syntax of the object language, augmented with syntax for (typed) metavariables, for example, `<Type id>`. When matching with concrete syntax patterns, the layout (whitespace and comments), inside the pattern as well as inside the subject value, is ignored. Lines 39–41 contain the concrete counterparts of the (abstract) node pattern referred to in the previous paragraph.

Deep matching and traversal

A descendant pattern—a pattern preceded by a `/`—finds a match by traversing all sub-values of the subject value and succeeds when a match is found anywhere (line 43). Descendant patterns usually produce more than one solution, therefore they often occur in `for` loops (line 44), `visit` constructs (lines 45–47), or comprehensions such that the programmer can collect, filter or do something with all instances.

Comprehensions and generators

Comprehensions, as illustrated on lines 49–51, are a means of generating new values by enumerating over existing values. An expression `[a .. b]` generates the list of integers from `a` up to, not including, `b`. We show a list comprehension: square numbers are generated (`n*n`) by enumerating the provided list on the right (using `<-`). Generators can also be used in control structures, such as `for`, to iterate over a value. Next to list comprehensions, RASCAL also supports set and map comprehensions and value reducers that produce a single value in comprehension style instead of a collection.

Multi-line strings and string interpolation

String literals are not delimited by line endings in RASCAL. Instead, string literals may span multiple lines. Single quote characters `'` may be used to allow such strings to be indented nicely: these characters, and preceding whitespace characters, are not part of the string literal. Lines 53 and 54 show an example of a multi-line string.

The actual value of a variable can be directly spliced into a string by using string interpolation, as illustrated on lines 55 and 56. This is not limited to string-typed variables; all RASCAL values may be interpolated. Since interpolation of `if`, `for`, and `while` constructs is also allowed, RASCAL has full-blown string template programming capabilities.

CLAIR

The RASCAL standard library does not contain a concrete grammar for C++. CLAIR[‡] (C++ language analysis in RASCAL) is a separate plug-in project, providing an abstract syntax definition for C++, based on the C++ parser in Eclipse CDT.[8] CLAIR was primarily developed to parse C++; while C is not a strict subset of C++[§], CLAIR does parse C files that are also valid C++. Throughout the rest of this article, we use the abstract syntax ADTs as defined in CLAIR.

[‡]https://github.com/cwi-swat/clair
[§]For instance, `class`, `private`, and `public` are valid identifiers in C, but keywords in C++.

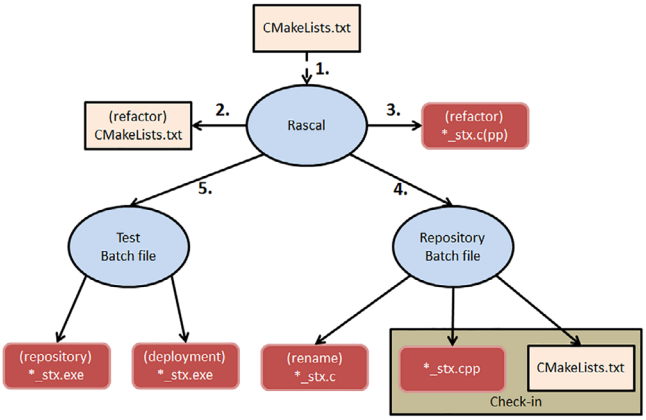

FIGURE 4 The key components and five steps of the automated refactoring process with their implementation languages and their inputs and outputs

4 | CASE STUDY: A SEMI-AUTOMATED APPROACH TO API MIGRATION

In Section 2, we introduced our case study to migrate away from STX in favor of GOOGLETEST. In this section, we explain all RASCAL code we used to automate these changes.

For each test suite, the refactoring is structured as depicted in Figure 4. A CMAKE file serves as seed input for our automated refactorings; from these files we learn which other C or C++ files to process. In Section 4.2, we describe how such a file is parsed. Next, in Section 4.3 we show how CMAKE files are refactored. In Section 4.4, we show how we carried out the refactoring subtasks (cf. Section 2.3). A batch file is generated that renames C files to C++ and generates a command to check in the changes in the version control system; the creation of this batch file is shown in Section 4.5. Finally, Section 4.6 displays the generation of another batch file that executes the refactored test suites. If the tests execute successfully, the changes can be delivered to the version control system.

4.1 | Writing changes to file

As discussed in Section 3, RASCAL/CLAIR does not contain a grammar for C++, but provides an AST abstraction. Finding refactoring candidates is done by matching abstract syntax patterns against ASTs. Once we found out where to change the files, all we need to do is generate the changes.

Because CLAIR does not (yet) come with a mapping back from C++ AST nodes to source code, we had to devise a creative solution. Every AST node has a source location attribute `src`, containing the filename and offset and length of the source code fragment it represents. We use this source location information to identify the exact fragments that are to be replaced and associate this location with a new code fragment that has to replace it. We store a series of these "edits" in a RASCAL relation with three columns: `<int startIndex, int endIndex, str changeWithString>`. Such as relation is called an "edit script," since it can be executed automatically by locating each position in the file and replacing the selected portion with the `changeWithString` value, and keeping tabs on the shifting of the indexes due to each replacement.

Listing 6 shows how changes are applied to a source file.

- The `changeSource` function takes the original source code and a set of changes as its parameters. The offset of a change is based on the unchanged source code. Since replacement source code and the deleted substring are typically not of the same length, we keep track of this discrepancy in the `offset` variable, initialized at `0`.
- Before applying the changes, the unsorted collection is sorted on the `startIndex`. The changes are then applied one by one, by concatenating the prefix, the replacement source code, and the postfix.[¶] After every replacement, the `offset` variable is updated with the difference in length of the deleted and inserted strings.

[¶] Omitting a lower or upper bound in string slices means a slice from the start or to the end, respectively.

- Finally, after application of all changes, the function returns a string containing the refactored source code.

The `changeSource` appears in multiple (disjoint) parts of the transformation, and is used at most once per file. The preconditions for applying this function correctly are that the `changesList` is based on the current state of the input `source` and that none of the intervals in the `changesList`overlap.

```
1  str changeSource(str source, rel[int, int, str] changesList) {
2    offset = 0;
3    for (<startIndex, endIndex, changeWithString> <- sort(changesList)) {
4      source = source[.. startIndex + offset] + changeWithString + source[endIndex + offset ..];
5      offset += size(changeWithString) - (endIndex - startIndex);
6    }
7    return source;
8  }
```

Listing 6: Function that applies an edit script (`changesList`) to source code

4.2 | Parsing CMAKE

Recall from Figure 4 that CMAKE files serve as input for all our refactorings. In Section 2.1, we introduced the structure of the files that are used at Philips. To be able to parse these files, we defined a grammar that accepts CMAKE files, including our company-specific macros. Listing 7 contains a grammar for these files in RASCAL's grammar formalism.

First, we define the syntax of whitespace and that # is the comment prefix (lines 1–4). On the following lines, the syntactic categories are declared that define general CMAKE concepts as well as our company-specific uses of CMAKE. This syntax definition is complete for us, in the sense that if we would have forgotten about one of the macros that we use, the generated parser would have produced a parse error. However, it is not a syntax definition for any CMAKE file in the world, for the same reason.

```
1  layout Layout = WhitespaceAndComment* !>> [\ \t\n\r#];
2  lexical WhitespaceAndComment
3    = [\ \t\n\r]
4    | "#" ![\n]* $;
5
6  start syntax Build = build: Section+ sections;
7  syntax Section = section: Target target Options options;
8  syntax Target = target: "set" "(" Id targetMacro Deps+ deps ")";
9  syntax Deps = deps: D dep;
10 syntax Options = options: Sources* sources CompileFlags* Configure;
11 syntax Sources = sources:  "set" "(" "$" "{" Id targetMacro "}" TT targetType SourceList+ sourceList ")";
12 syntax SourceList = sourceList: "\\""? "$"? "{"? Id sourceFile "}"? "\\""?;
13 syntax CompileFlags = compileFlags: "set" "(" "$" "{" Id targetMacro "}" "\_COMPILE\_FLAGS" "\"" Id compileFlags "\"" ")";
14 syntax Configure = configure: ConfigureLibrary*  ConfigureTestExecutable*;
15 syntax ConfigureLibrary = configureLibrary: "configureLibrary" "(" "$" "{" Id targetMacro "}" Id buildTarget ")";
16 syntax ConfigureTestExecutable
17   = configureTestExecutable: "configureTestExecutable" "(" "$" "{" Id targetMacro "}" Id buildTarget ")"
18   | configureUnitTestExecutable: "configureUnitTestExecutable" "(" "$" "{" Id targetMacro "}" Id buildTarget ")"
19   | configureExecutable: "configureExecutable" "(" "$" "{" Id targetMacro "}" Id buildTarget ")";
20
21 keyword Reserved
22   = "set" | "$" | "{" | "}" | "_COMPILE_FLAGS" |  "configureLibrary" | "configureTestExecutable"
23   | "configureUnitTestExecutable" | "configureExecutable";
24
```

```
lexical Id = ([a-zA-Z/.\-][a-zA-Z0-9_/.]* !>> [a-zA-Z0-9_/.]) \ Reserved;
lexical TT = ([_][A-Z_]* !>> [A-Z_]) \ Reserved;
lexical D =  ([a-zA-Z][a-zA-Z0-9${}\_.]* !>> [a-zA-Z${}\_.]) \ Reserved;
```

Listing 7: Concrete grammar of CMAKE as used at our company, in RASCAL's built-in grammar notation

4.3 | Refactoring CMAKE

As described in Section 2.3, refactoring our CMAKE files consist of three actions: changing the language (if applicable), replacing include directives, and replacing the STX macro. Listing 8 shows the functions we used. The `modifyCMakeLists` function first matches out sections containing an STX dependency (line 4). The traversal in the `modifyStxTarget` function has a case for each of the actions.

- Changing the language is only necessary for C files; references to C++ files remain unchanged. Recall that all our test suite file names end with `_stx`. The `modifyCTarget` function in Listing 8 first matches out filenames containing `"_stx.c"` from `SourceList` productions (line 30). For each file, the file extension is then altered when applicable (lines 31–34).
- To replace the `stxserverlib` include with the `GTEST_INCLUDE_DIR` and `GTESTSETUP_INCLUDE_DIR` includes, we first search for a `DIRS` directive containing `"stxserverlib"`, and replace this line with the two new includes (lines 15–20).
- Finally, the `configureTestExecutable` macro needs to be replaced with `configureUnitTestExecutable`. We search for `configureTestExecutable` and overwrite it with `configureUnitTestExecutable`, with the same arguments (lines 21–23).

```
1  rel[int, int, str] modifyCMakeLists(Build build) {
2    changes = {};
3    visit (build) {
4      case sec:section(target(_, [*_, deps(/_stx/i), *_]), _) :
5        changes += modifyStxTarget(sec);
6    }
7    return changes;
8  }
9  rel[int, int, str] modifyStxTarget(Section section) {
10   changes = {};
11   visit (section) {
12     case sources(_, "_SOURCES", sources) : {
13       changes += modifyCTarget(sources);
14     }
15     case sources(_, "_DIRS", [*_, l:sourceList(_, _, _, /stxserverlib/i, _, _), *_]) : {
16       changes += <l.src.offset, l.src.offset + l.src.length,
17         "${GTEST_INCLUDE_DIR}
18         '    ${GTESTSETUP_INCLUDE_DIR}
19         '">;
20     }
21     case t:configureTestExecutable(a, b) :  {
22       changes += <t.src.offset, t.src.offset + t.src.length, "configureUnitTestExecutable(${<a>} <b>)">;
23     }
24   }
25   return changes;
26 }
27 rel[int, int, str] modifyCTarget(SourceList+ sources) {
28   changes = {};
```

```
29    visit (sources) {
30      case l:sourceList(_,_,_, t:/_stx.c/i,_,_) :   {
31        if (!contains(t, ".cpp")) {
32          offset = l.src.offset + size(t);
33          changes += <offset, offset, "pp">;
34        }
35      }
36    }
37    return changes;
38  }
```

Listing 8: Metaprogram to refactor CMAKE files belonging to STX to their GOOGLETEST counterparts

The `processTestSuitesFromCMakeLists` function, which is not depicted, is a wrapper for the functions listed in Listing 8.

4.4 | Refactor STX API client code to GOOGLETEST API client code

In this section, we show the scripts we used to automate the refactorings described in Section 2.3.

Change 1: C to C++

Because the C++ compiler is more strict than the C compiler, we have to make certain changes to our existing code to satisfy it. In particular, there is a `TOS_p_SEG_create` function in our OS abstraction layer that acquires some memory from the heap. The return type of this function is pointer-to-void (`void*`), whereas the declared type of a variable such a value is assigned to is typically a pointer to an actual type. Such an assignment of a `void*` value to a more defined pointer type is flagged by the C++ compiler. To overcome this, we try to find such a variable's type, and add a typecast to this type.

- In Listing 9, the `voidPointerAssignments` function traverses the AST of a source file, searching for locations where the result of a call to `TOS_p_SEG_create` is assigned to some variable. For all such assignments, it calls the `findTypeCast` function, which traverses the AST to find the declaration site of the variable, and tries to extract its type from it. Notably only the local name of the variable is used here (as opposed to a fully qualified name), and the first declaration AST for this local name is used. So this works under the following assumption: the first declaration of that name is indeed of the expected type for the call to `TOS_p_SEG_create`. The way the test setup code is structured guarantees this assumption.
- When successful, syntax for a cast to this type is returned.
- If it fails to find the type, it returns `"(FIXME)"`, which leads to a compilation error later in the pipeline when we would try to compile our new test code. The `FIXME` identifier does not occur elsewhere in our code. In this way, we make sure we can never accidentally accept ill-understood test code, which would otherwise lead to a deterioration of the quality of our test suites.

```
1  rel[int, int, str] voidPointerAssignments(Declaration ast) {
2    changes = {};
3    visit (ast) {
4      case assign(idExpression(n:name(_)), f:functionCall(idExpression(name("TOS_p_SEG_create")),_)): {
5        changes += <f.src.offset, f.src.offset, findTypeCast(ast, n.\value)>;
6      }
7    }
8    return changes;
9  }
```

```
10 str findTypeCast(Declaration ast, Name n) { \label{lst:typecase}
11   visit (ast) {
12     case simpleDeclaration(
13         namedTypeSpecifier(_, name(t)),
14         [declarator([pointer(_)], name(n),_), *_]):
15       return "(<t>*)";
16   }
17   return "(FIXME)";
18 }
```

Listing 9: Metaprogram transforming `void*` casts to casts to a more defined pointer type

The second part of this change handles potential name mangling issues introduced by changing C headers to C++. The inclusion of a header file compiled in C in a C++ unit results in linking errors, which can be resolved by wrapping the inclusion in an `extern "C"` block. Our C header files end with `li` or `pi` (from local or public include), which we can use to identify relevant include directives for refactoring. Since include directives are preprocessor statements, they do not appear in the AST of a file. Therefore, for this refactoring, we operate on source files in string representation.

Listing 10 shows the `functionNameMangling` function. It iterates over a source file line by line, wrapping groups of consecutive C header inclusions in an `extern "C"` block.

```
1  list[str] functionNameMangling(list[str] lines) {
2    first = true;
3    for (i <- [0 .. size(lines)-1]) {
4      if (contains(lines[i], "#include") && (contains(lines[i], "pi.h") || contains(lines[i], "li.h"))) {
5        if (first) {
6          lines[i] = "extern \"{C}\" {
7                     '<lines[i]>";
8          first = false;
9        }
10       if (!(contains(lines[i+1], "#include") && (contains(lines[i+1], "pi.h") || contains(lines[i+1], "li.h")))) {
11         lines[i+1] = "}
12                      '<lines[i+1]>";
13         first = true;
14       }
15     }
16   }
17   return lines;
18 }
```

Listing 10: Metaprogram wrapping C header inclusions in an `extern "C"` block to solve name mangling issues

Change 2: Replace asserts

Here we arrive at a more interesting transformation. This refactoring actually changes the semantics of the test code a bit, in order to get the benefits of the more precise GOOGLETEST asserts. The `modifyAsserts` function in Listing 11 changes STX asserts to GOOGLETEST asserts.

- It traverses the provided AST, matching all function calls to STX's `GEN_m_assert`, binding the function name and the argument to the `func` and `arg` variables. To determine which GOOGLETEST assertion should be inserted, it performs a case distinction on the argument.
- In case this is an equality (`a == b`), the STX macro is replaced with `ASSERT_EQ` (line 7). Since this macro expects two arguments, the equality operator—located between the two operands—is changed into a comma (line 8). Similarly, for

an inequality (`a != b`), the macro is replaced with `ASSERT_NE`, and the inequality operator is replaced by a comma (lines 11 and 12).

- In case the asserted value was a negation (`!a`), the macro is replaced with `ASSERT_FALSE` (line 15), and the negation operator is removed (line 16).
- In any other case, the asserted value is left untouched, and the macro is replaced with `ASSERT_TRUE` (line 19).

In Section 4.9, we will show that this subtask has indeed improved our error reporting.

```
1  rel[int, int, str] modifyAsserts(Declaration ast) {
2    changes = {};
3    visit (ast) {
4      case functionCall(idExpression(func:name("GEN_m_assert")), [arg]):
5        switch (arg) {
6          case equals(a, b): {
7            changes += <func.src.offset, func.src.offset + func.src.length, "ASSERT_EQ">;
8            changes += <a.src.offset + a.src.length, b.src.offset, ", ">;
9          }
10         case notEquals(a, b): {
11           changes += <func.src.offset, func.src.offset + func.src.length, "ASSERT_NE">;
12           changes += <a.src.offset + a.src.length, b.src.offset, ", ">;
13         }
14         case n:not(a): {
15           changes += <func.src.offset, func.src.offset + func.src.length, "ASSERT_FALSE">;
16           changes += <n.src.offset, a.src.offset, "">;
17         }
18         default:
19           changes += <func.src.offset, func.src.offset + func.src.length, "ASSERT_TRUE">;
20       }
21   }
22   return changes;
23 }
```

Listing 11: Metaprogram generating GOOGLETEST assertions, based on the asserted value in STX

Change 3: Replace header inclusion

The STX header files have to be removed, and a new `SetupTestDependencies` header file is to be added as the final inclusion. Since the script from Section 4.3 modified the file before, this step is performed after writing the changes of the other scripts to disk. Again, as include directives are not present in ASTs, we perform this refactoring by handling files as strings.

- The `modifyHeaders` function in Listing 12 first reads in the file line by line. If a line contains an include directive, the line number is stored in the `last` variable. Furthermore, if the included file is an STX header, the include directive is removed (line 8).
- After iterating over all lines, the `last` variable holds the line number of the final include directive. To ensure the new header file is not included inside an `extern "C"` block, the script checks whether there is a closing curly brace on the line trailing the final include directive, and increases the `last` variable if needed.
- Finally, an include directive for the new GOOGLETEST header is appended to the appropriate line (line 15).

We realize that the above transformation is risky in the sense that we have a number of assumptions on the source code we want to transform. In particular, the } check could go wrong for a macro or class definition. However, since our coding standards do not allow this, this does not occur in the codebase. It is interesting how knowing our context helps

with simplifying assumptions that enable us to make rapid progress on the automated refactoring rather than having to consider how our refactoring would perform on any code in the wild.

```
1  void modifyHeaders(loc f) {
2    lines = readFileLines(f);
3    last = 0;
4    for (i <- [0 .. size(lines)-1]) {
5      if (contains(lines[i], "#include")) {
6        last = i;
7        if (contains(lines[i], "STXServerLib.h") || contains(lines[i], "STXServer.h")) {
8          lines[i] = "";
9        }
10     }
11   }
12   if (contains(lines[last+1], "}")) {
13     last += 1; // place outside extern "C" block
14   }
15   lines[last] += "\r\n\r\n#include \"SetupTestDependencies.h\"";
16   writeFileChanges(f, lines);
17 }
```

Listing 12: Metaprogram removing STX header inclusions and adding a GOOGLETEST header inclusion

Change 4: Rewrite test function

Listing 13 describes the two functions that make up this refactoring step.

- First, all test case names are collected using the `findTestCases` function, which searches for the test registration function and stores the first argument, containing the test case name.
- Second, the `visitTestCases` function iterates over the found test cases. For each test name, the AST is then traversed, matching function definitions and function declarations having this name. The declarations are removed (lines 15 and 16), and the definitions are replaced by the GOOGLETEST macro (lines 13 and 14).

```
1  list[str] findTestCases(Declaration ast) {
2    testCases = [];
3    visit (ast) {
4      case functionCall(idExpression(name("RegisterTestFunction")), [arg0, *_]):
5        testCases += arg0.name.\value;
6    }
7    return testCases;
8  }
9  rel[int, int, str] visitTestCases(Declaration ast, list[str] testCases, str testClassName) {
10   changes = {};
11   for (testCase <- testCases) {
12     visit (ast) {
13       case functionDefinition(ret, f:functionDeclarator(_,_, name(testCase),_,_),_, body):
14         changes += <ret.src.offset, f.src.offset + f.src.length, "TEST_F(<testClassName>, <testCase>)">;
15       case f:simpleDeclaration(_, [functionDeclarator(_,_, name(testCase),_,_), *_]):
```

```
16          changes += <f.src.offset, f.src.offset + f.src.length, "">;
17      }
18    }
19    return changes;
20  }
```

Listing 13: Metaprogram that replaces STX test case specifications with a GOOGLETEST definition

Change 5: Refactor entry function

Listing 14 shows the metaprogram to perform this threefold change.

- The `removeRegisterPrintfs` in Listing 14 removes occurrences of the printing function `PP_p_PF_printf` if what is to be printed contains `"register"`.
- The name of the entry function is extracted using the `getColdEntry` function. It searches for calls to the `TOS_p_STP_start_continue_set` function, which takes the entry function name as its first argument.
- This entry function name is subsequently used by the `modifyColdEntry` function to match out the body of the entry function (line 22). In the body, calls to the STX function `RegisterTestFunction` are removed (lines 24 and 25), and calls to the `StartTestServer` function are replaced (lines 26–29).

```
1  rel[int, int, str] removeRegisterPrintfs(Declaration ast) {
2    changes = {};
3    visit (ast) {
4      case f:expressionStatement(functionCall(idExpression(name("PP_p_PF_printf")), [stringLiteral(arg0), *_])):
5        if (contains(arg0, "register")) {
6          changes += <f.src.offset, f.src.offset + f.src.length, "">;
7        }
8    }
9    return changes;
10 }
11 str getColdEntry(Declaration ast) {
12   visit (ast) {
13     case functionCall(idExpression(name("TOS_p_STP_start_continue_set")), [arg0, *_]):
14       return arg0.name.\value;
15   }
16   throw "Could not find an entry function.";
17 }
18 rel[int, int, str] modifyColdEntry(Declaration ast, str testClassName) {
19   changes = {};
20   coldEntry = getColdEntry(ast);
21   visit (ast) {
22     case functionDefinition(_, functionDeclarator(_,_, name(coldEntry),_,_),_, body): {
23       visit (body) {
24         case f:expressionStatement(functionCall(idExpression(name("RegisterTestFunction")),_)):
25           changes += <f.src.offset, f.src.offset + f.src.length, "">;
```

```
26          case f:expressionStatement(functionCall(idExpression(name("StartTestServer")),_)):
27            changes += <f.src.offset, f.src.offset + f.src.length,
28              "int testResult = SetupTestDependencies::GoogleTest(argc_input, argv_input);
29              'exit(testResult);">;
30        }
31      }
32    }
33    return changes;
34  }
```

Listing 14: Metaprogram rewriting test entry functions from STX to GOOGLETEST

Change 6: Rewrite reporting of test verdict

In Listing 15, the `replacePassFail` function localizes STX notification functions by traversing a provided AST.

- Calls to STX's `NotifyTestFailed` are replaced with GOOGLETEST's `FAIL`.
- Since tests pass implicitly, STX calls to `NotifyTestPassed` are simply removed.

```
1  rel[int, int, str] replacePassFail(Declaration ast) {
2    changes = {};
3    visit (ast) {
4      case f:expressionStatement(functionCall(idExpression(name("NotifyTestFailed")),_)):
5        changes += <f.src.offset, f.src.offset + f.src.length, "FAIL();">;
6      case f:expressionStatement(functionCall(idExpression(name("NotifyTestPassed")),_)):
7        changes += <f.src.offset, f.src.offset + f.src.length, "">;
8    }
9    return changes;
10 }
```

Listing 15: Metaprogram removing STX's positive verdict notification and replacing STX's negative verdict notification by the GOOGLETEST counterpart

Change 7: Rewrite main function

Listing 16 provides the metaprogram that was used to rewrite our main functions.

- In the STX source files, we encounter `main` and `_tmain` that can serve as main functions. The `modifyMain` function locates functions with either of these names in the provided AST. The function body is then passed to the `changeMainBody` function.
- Calls to `TOS_p_STP_start_continue_set` are removed (lines 12 and 13).
- Calls to `TOS_p_STP_boot` are replaced by the `if` statement and accompanying statements (lines 14–23).
- The return value is changed from `0` to `testResult` (lines 25 and 26).

```
1  rel[int, int, str] modifyMain(Declaration ast) {
2    visit (ast) {
3      case functionDefinition(ret, functionDeclarator(_,_, name("main"),_,_),_, body):
```

```
4         return changeMainBody(body);
5       case functionDefinition(ret, functionDeclarator(_,_, name("_tmain"),_,_),_, body):
6         return changeMainBody(body);
7     }
8   }
9   rel[int, int, str] changeMainBody(Statement body) {
10    changes = {};
11    visit (body) {
12      case f:expressionStatement(functionCall(idExpression(name("TOS_p_STP_start_continue_set")),_)):
13        changes += <f.src.offset, f.src.offset + f.src.length, "">;
14      case f:expressionStatement(functionCall(idExpression(name("TOS_p_STP_boot")),_)): {
15        changes += <f.src.offset, f.src.offset + f.src.length,
16          "  int testResult = 0;
17          'if (SetupTestDependencies::IsListArgumentSpecified(argc_input, argv_input)) {
18          '    testResult = SetupTestDependencies::GoogleTest(argc_input, argv_input);
19          '} else {
20          '    TOS_p_STP_start_continue_set(GEN_p_cold_entry);
21          '    TOS_p_STP_boot();
22          '}
23          '">;
24      }
25      case  f:\return(integerConstant("0")):
26        changes += <f.src.offset, f.src.offset + f.src.length, "return testResult;">;
27    }
28    return changes;
29  }
```

Listing 16: Metaprogram rewriting the main functions of our test suites

Change 8: Rewrite CMAKE files

Listing 17 glues the previously introduced RASCAL functions together.

- The `refactorTestSuite` function is called with the STX source file it needs to refactor. First, it generates the GOOGLETEST class name (line 2). The file is then passed to CLAIR to generate an AST (line 3), which is passed on to the refactoring functions (lines 7–13). Optionally, if the provided file is a C file, the additional refactoring function is called (lines 15–17).
- The accumulated changes are then applied to the file (lines 19–21).
- Finally, the headers are modified (line 22).

We did not provide the code of the `satisfyCppCompiler` function (line 16), which is simply a wrapper for the `functionNameMangling` and `voidPointerAssignments` functions. The `addTestClass` function is also not provided; this function trivially generates a single line of code containing a forward declaration of the new GOOGLETEST test class (cf. Listing 3, line 4).

```
1  void refactorTestSuite(loc f) {
2    testClassName = createTestClassName(f);
3    ast  = parseCpp(f);
4    testCases = findTestCases(ast);
5
6    changes = {};
7    changes += visitTestCases(ast, testCases, testClassName);
```

```
8     changes += addTestClass(ast, testCases, testClassName);
9     changes += modifyColdEntry(ast, testClassName);
10    changes += modifyAsserts(ast);
11    changes += replacePassFail(ast);
12    changes += modifyMain(ast);
13    changes += removeRegisterPrintfs(ast);
14
15    if (isC(f)) {
16      changes += satisfyCppCompiler(ast);
17    }
18
19    fc = readFile(f);
20    fc = changeSource(fc, changes);
21    writeFile(f, fc);
22    modifyHeaders(f);
23  }
```

Listing 17: Wrapper function around the metaprograms related to refactoring C/C++ code

4.5 | Repository batch file generation

In this section, we describe the generation of a batch file to store the automated refactorings in our version control system, Rational Team Concert (RTC).[9] This `rtc.bat` file has two responsibilities: changing the extension from C files to `.cpp`, and checking in the changed files into the repository. Listing 18 shows how we generate these batch files. We use the command line tool `lscm` for RTC automation. The `generateRtcRenameLines` function generates the required lines for changing the extension of all C files. We use `lscm` to `move` names, since this allows us to preserve the files' version history. The function `generateRtcChangeSetLines` builds a string for checking in the changed CMAKE file and all changed C++ files, including a comment listing all changed test suite names. `generateBatchScript` calls these functions and writes the combined output to an `rtc.bat` file. All three functions get the AST of a CMAKE file as input.

As an alternative we could also have executed these changes using RASCAL's IO library. However, the batch files are an independent stage in our refactoring process that can be scrutinized by our colleagues without knowledge of RASCAL, and we can run these scripts on machines where RASCAL was not installed.

```
1   str generateRtcRenameLines(Build ast) {
2     rVal = "";
3     for (fileName <- StxCFiles(ast)) {
4       rVal += "call lscm move path <fileName> <fileName>pp\n";
5     }
6     return rVal;
7   }
8   str generateRtcChangeSetLines(Build ast) {
9     rVal = "call lscm checkin CMakeLists.txt";
10    for (fileName <- StxCppFiles(ast)) {
11      rVal += " " + fileName;
12    }
13    for (fileName <- StxCFiles(ast)) {
14      rVal += " " + fileName + "pp";
15    }
16    rVal += " --comment \"{R}efactor";
17    for (fileName <- StxCppFiles(ast) + StxCFiles(ast)) {
18      rVal += " " + fileName;
```

```
19    }
20    rVal += " STX test cases.\"";
21    return rVal;
22  }
23  void generateBatchScript(Build ast, loc f) {
24    rVal = "";
25    name = |file:///| + getPath(f) + "rtc.bat";
26    rVal += generateRtcRenameLines(ast);
27    rVal += generateRtcChangeSetLines(ast);
28    writeFile(name, rVal);
29  }
```

Listing 18: Metaprograming generating the repository batch scripts

4.6 | Test changes

To verify that the changes generated by the automated refactoring preserve did not break any tests, we use the `generateTestScript` function from Listing 19 to generate a batch file that executes the refactored test suites. This batch file is used to run the converted test suites in the output directory of the repository, run the test suites from the repository location, copy the test suites to the deployment location, and run them from there as well. The results of both runs are stored in a `logfile.txt` file.

The `generateTestScript` function gets the AST of a CMAKE file as input parameter. It generates a batch script by iterating over all C and C++ files, changing their extensions to `.exe` to get the executable names, and adding several batch commands to the intermediate result. Finally, the full batch script is written to a `test.bat` file.

```
1   void generateTestScript(Build ast, loc f) {
2     path = "\"{C}:\\path\\to\\deployment\\location\\\"";
3     name = |file:///| + getPath(f) + "test.bat";
4     batch = "cd \\path\\to\\build\\output\n";
5     for (file <- StxCppFiles(ast) + StxCFiles(ast)) {
6       file = changeExtension(file, ".exe");
7       splitted = split("_", file);
8       testName = "";
9       for (sp <- splitted) {
10        testName += replaceFirst(sp, sp[0], toUpperCase(sp[0]));
11      }
12      testName = replaceAll(testName, ".exe", "");
13      logSuffix = "1$>$$>c:templogfile.txt2\>&1";
14      batch += "<file> --gtest_filter=*<testName>.* <logSuffix>
15                'echo Exit Code for <file> is
16                'copy <file> <path> <logSuffix>
17                '<path><file> <logSuffix>
18                'echo Exit Code for <file> is
19                '";
20    }
21    writeFile(name, batch);
22  }
```

Listing 19: Metaprograming generating the test batch scripts

4.7 | Putting it all together

We now describe how the previously described RASCAL scripts are called. The `main` function has a list of CMAKE files; these are the input for all our refactorings. The function iterates over all input files:

- it parses the input file;
- it refactors the test suites;
- it refactors the CMAKE input file;
- it generates an `rtc.bat` batch file;
- it generates a `test.bat` batch file.

Listing 20 shows the entry function of the full refactoring.

```
1  void main(list[loc] cmakeFiles) {
2    for (file <- cmakeFiles) {
3      build = parse(file);
4      refactorTestSuites(build, file);
5      generateBatchScript(build, file);
6      generateTestScript(build, file);
7    }
8  }
9  void refactorTestSuites(Build build, loc file) {
10   changes = {};
11   processTestSuitesFromCMakeLists(build, file);
12   changes += modifyCMakeLists(build);
13   fc = readFile(file);
14   fc = changeSource(fc, changes);
15   writeFile(file, fc);
16 }
```

Listing 20: Main function of our semi-automated refactoring

4.8 | Application of the RASCAL metaprograms on an older product line

As described in Section 1, we applied the automated refactorings on our new architecture, in which the legacy STX test framework was reused from an older version of the system. In the same department, we also perform maintenance on an older version of the system that is still in operation at customer sites. This line does not have GOOGLETEST and uses STX for all unit tests. Since we want engineers to be able to easily switch between the two product lines, we decided to also refactor the STX test suites from the old product line from STX to GOOGLETEST.

The new product line uses CMAKE files from which we generate a Visual Studio solution and project files. However, the old product line uses a Visual Studio solution and project files directly; they are not generated. Similarly to the CMAKE files in the newest product line, these files contain references to STX include directories that need to be removed, and GOOGLETEST dependencies that need to be added. Using a 26 line RASCAL function, we read in all solution and project files and changed them accordingly (code not shown). All other scripts related to the refactoring of STX source files and the creation of the repository and test batch scripts could be fully reused.

This experience is an indication that there is opportunity for reuse for such besproke refactoring scripts, even when we know how they depend on specific assumptions regarding C/C++ coding conventions and style within the company.

4.9 | Quantifying the automated refactoring

We applied our semi-automated refactoring strategy to the current product line and an older version of the system. Table 1 provides an overview of the number of affected files and the number of modifications. In total, we used 24 change sets

TABLE 1 Overview of the number of files and modifications affected by our semi-automated refactoring

	Change sets	Suites	C files	C++ files	Modifications
Old PL	11	77	46	31	3100
New PL	13	75	42	27	2740
Total	24	152	88	58	5840

TABLE 2 Quantification of modifications in terms of source lines of code (SLOC)

	Removed SLOC	Added SLOC	Changed SLOC
Old PL	603	539	2497
New PL	522	525	2218
Total	1125	1064	4715

TABLE 3 Number of generated lines of batch script

	Repository scripts	Test scripts	Total
Old PL	57	385	442
New PL	55	375	430
Total	24	152	872

TABLE 4 Distribution of generated GOOGLETEST asserts

	`ASSERT_EQ`	`ASSERT_NE`	`ASSERT_TRUE`	`ASSERT_FALSE`	Total
Old PL	1159	23	137	11	1330
New PL	1532	52	232	19	1835
Total	2691	75	369	30	3165

to commit all changes to the version control system. On the new product line 75 STX test suites were changed, where on the old product line 77 STX test suites were adapted. The majority of STX test suites were changed from C to C++; only a quarter was already in C++ before refactoring. A grand total of 5840 edits were made to the source code files.

In Table 2, we zoom in into these modifications. We distinguish three types of modifications: removal of a source line of code (SLOC), addition of a new SLOC, and changing of a SLOC. We count both alterations and complete line replacements as a changed SLOC.

Table 3 provides an overview of the number of generated lines of batch script, for both the repository and the test scripts. The length of the repository script depends on the number of change sets, and the number of source files that were converted from C to C++ (cf. Listing 18). The length of the test script is linear in the amount of test suites (cf. Listing 19). In total, 872 lines of batch script were generated.

On the new product line, a total of 192 lines of CMAKE were changed. We do not use CMAKE on the old product line, but use a Visual Studio solution and project files directly; in total, our metaprograms changed 893 lines in these build configuration files.

The GOOGLETEST `ASSERT_EQ` and `ASSERT_NE` macros are the preferred assertion, since they provide detailed information in case of a violation. Based on the operator that was used in the argument of the preexisting STX assertion macro, our scripting automatically deduced which GOOGLETEST macro to replace it with. Table 4 shows the distribution of the assertion macros generated by the automated refactoring. As the vast majority of STX asserts could be automatically transformed to an `ASSERT_EQ` or `ASSERT_NE` assert, we were able to greatly improve error reporting. Only a small portion of asserts was transformed to `ASSERT_TRUE` or `ASSERT_FALSE`, for which error reporting remains similar to the STX situation.

One of the requirements for **Step 5** to be successful is that all references to STX had to be removed. To verify that this is indeed the case, we removed the STX library itself from the codebase and performed a full build. As this was successful, and all tests succeeded, we conclude that our refactoring metaprograms did not miss anything.

5 | EVALUATION AND DISCUSSION

We described the process and the code we wrote to automate a large-scale refactoring of C++ test code. To evaluate the approach, informally, we are interested in answering the following evaluation questions:

- What is the quality of the resulting code? Does it compile, run and test correctly? What is the maintainability of the output code as compared to the original code? (Section 5.1)
- Was the effort of (a) learning how to automate refactorings and (b) scripting the refactoring worth it, as compared to the hypothetical manual approach? (Section 5.2)
- Which parts of the current approach can be reused in future large-scale C++ maintenance scenarios, either (a) at our company, or (b) for others? (Section 5.3)
- What are possible alternative implementation technologies comparable to RASCAL that could have been used, and what are their drawbacks and benefits? (Section 5.5)

5.1 | Quality of the refactored code

The following observations pertain to the quality of the result of our automated refactoring, in terms of functionality and maintainability of the output code:

- Since only test framework API client code was changed, the code quality (readability) of the bodies of the test code is similar to (not worse than) the old situation. The resulting code as a whole after the transformation was acceptable by company standards, which was confirmed by code review by an independent colleague.
- The quality of test reporting has increased significantly, as an effect of the use of specific assert methods of the new framework API. If a test fails, the report contains more detailed, easier to diagnose, information than before.
- By design, after completion of the semi-automated refactoring process, the whole codebase compiles and runs; all tests also succeed.
- The maintainability of the codebase as a whole has improved: we can stop maintaining the old proprietary STX framework, and we now consistently use the same framework across the entire codebase.

5.2 | The effort of automating a refactoring and learning how to do it

Designing and implementing this automated refactoring came with an upfront investment in (a) learning metaprogramming in RASCAL and (b) writing the bespoke refactoring code. Was this worth the effort or not? To the best of our knowledge there exist only few publications on A/B testing such a situation. The use of DSL tools was A/B tested, assessing implementing domain-specific language processors against implementing DSLs in a GPL;[10] their conclusion does not apply to the current situation since we are analyzing C++ by reusing an existing front-end. Their method is also not applicable; they started from a clearly delineated set of features to implement, while our situation requires discovery and backtracking based on emerging insights.

In the current section we therefore propose a thought experiment. We imagine, by observing the (complete) work done by the automated refactoring what a hypothetical and optimal manual process would have been and what it might have cost. We first describe the experience of the first author while creating the RASCAL refactoring metaprograms. Then, we reflect on the key characteristics of RASCAL that have enabled our refactoring in Section 5.2.3. In Section 5.2.4, we compare the two workflows of Figure 1 in terms of the (estimated) effort required.

5.2.1 | Experiences with and evolution of the RASCAL metaprograms

The RASCAL scripts described in Section 4 were written by the first author of this article. His first steps with RASCAL were made two years before this project, during a course on Software Evolution at the Open University.[#] He started with the creation of the `refactorTestSuites` function, and applied it on two example test suites. The use of a source code repository meant that it was trivial to undo changes made by the scripts on source files. After deeming the automated refactoring sufficient, he decided that CMAKE files would be the perfect files as input for the refactoring. Because these files themselves needed to be adapted as well, he wrote a CMAKE grammar. The next dry test was on a CMAKE file that defined the build targets for approximately 20 STX test suites. The metaprograms were slightly adapted until the CMAKE file and the STX source files were all refactored correctly. During this process the first author never needed external guidance other than from the RASCAL documentation.

After the dry runs, he followed the process described in Section 1, and the refactoring metaprograms were repeatedly applied to a partition of the Positioning Subsystem. For each partition, this consisted of the following steps:

- The RASCAL scripts received a number of CMAKE files as input, and changes to the source files were generated accordingly.
- The changes were informally reviewed by the same person. When required, minor improvements were made (cf. Section 4.4, **Change Language**).
- The resulting code was checked-in into the source code repository and a change set was created using a repository script.
- The tests that were changed were built (compiled and linked).
- The test script was executed to check whether the refactored tests still succeed. It is important to note that all tests succeeded before the refactoring.
- According to company policy, all change sets were formally reviewed by senior software engineers. Review comments were processed, and the reviewers formally signed off for approval.
- Also according to company policy, the changes were built and tested on an offload system. When successful, the change sets were accepted into the source code repository.

5.2.2 | Issues detected in RASCAL and CLAIR

During the creation and application of the RASCAL scripts we encountered some issues with the C++ front-end CLAIR. In some test suites, assertions (cf. Section 2.3) were placed in a macro definition at the top of a source file. AST nodes belonging to expansions of such macros receive the source location of the macro occurrence, which gave unexpected results in our scripting. CLAIR was fixed to add an attribute to AST nodes, indicating whether a given node is the result of a macro expansion. Then we changed our RASCAL scripts to not apply any changes to macro expansions, but report the macros in need of manual intervention to the metaprogrammer instead.

It would sometimes occur that our metaprograms identified refactoring targets that were located in included files, rather than in the source file under analysis. This is due to the semantics of CLAIR, which runs the C preprocessor before parsing the input file. In order to keep processing files one by one, we eventually added a check to verify that only changes in the actual file under analysis were propagated. This was easy since every AST node produced by CLAIR has a field to indicate its source location.

RASCAL does not come with a feature or library to model edits on a source file. Most RASCAL programs either transform parse trees (from which source code can be derived by unparsing) or they transform abstract syntax trees, which require a pretty-printing function that produces source code again. The first author designed a model consisting of a relation between slices of a file and the text to substitute there, and a function which applies these edits to an existing file. A slightly generalized version of this feature would look nice in RASCAL's standard library, since it circumvents the use of a (bespoke) pretty-printer and also entails high-fidelity transformations (no unnecessary loss of indentation or source code comments).

[#]https://www.ou.nl/en/-/IM0202_Software-Evolution

If CLAIR would have provided a pretty-printer that maps ASTs back to source code, we could have written our transformations as simple AST-to-AST transformations, and derive the (same) edit scripts from diffing the AST nodes and pretty-printing only the new parts. In our current scripts the syntactical change code is tangled with collecting edit scripts; the code would be a little simpler without this tangling. However, a pretty-printer is a nontrivial piece of work for an elaborate language such as C++.

Another way of circumventing this tangling of code changes with their implementation as edit scripts would have been to use RASCAL's experimental concrete syntax features for abstract syntax trees.[11] This would allow us to write both the patterns and their substitutions in the C++ source language, while the same ASTs would be used under the hood. This would be a readable solution and the edit scripts would be derivable without the need for a pretty-printer; in fact, this would even allow us to directly unparse modified syntax trees. However, this feature is not released with RASCAL yet.

5.2.3 | Technological enablers in RASCAL

In Section 3, we introduced concepts of the RASCAL metaprogramming system that were used to carry out the refactoring from Section 4. In particular, we made heavy use of deep matching, that is, traversing a tree and matching patterns at arbitrary depth. Throughout Section 4, we used abstract patterns, which required knowledge of abstract syntax of the object language. The use of concrete syntax pattern—patterns that are expressed in actual surface syntax of the object language—would discard this requirement, making the conceptual entry barrier lower.

As we defined a concrete grammar for CMAKE (cf. Section 4.2), RASCAL immediately supports concrete syntax for CMAKE. For C++, however, there is no concrete grammar available in RASCAL. Augmenting the available functionality for C++ with concrete syntax pattern, for instance by implementing the Concretely framework,[12] would certainly make our patterns more concise and more readable. Furthermore, this would fully eliminate the bookkeeping of applied code changes, including file offset corrections, as (changed) parse trees could simply be unparsed.[11]

One of our key observations is that when conducting a significant refactoring task such as the one described in Section 2, it is imperative to be flexible. In this particular case, for instance, the initial assumption was that only C++ files would be affected, and hence, being able to parse and analyze C++ files would suffice. During the process, however, it became obvious that CMAKE files would serve as a better starting point for analysis. RASCAL facilitated seamless integration of the newly created CMAKE grammar (cf. Section 4.2) into the C++ analysis code.

5.2.4 | Comparison of the two workflows

In Section 1, we described our semi-automated refactoring process and a hypothetical manual alternative. In this section, we compare the required effort for each step in both processes.

Step 0 As we perform partitioning solely to make the formal reviewing effort tractable, this step does not depend the chosen refactoring process.

Step 1 The effort for problem analysis is identical for both processes; after all, while the two approaches differ, the task at hand is identical. This applies to both the initial problem analysis, and further analyses to improve previous solutions that caused errors in **Step 4**.

Step 2 In the manual process, each time this step is reached, a single or a few files are selected and altered, according to the strategy devised in **Step 1**. In the scripted process, the strategy from **Step 1** is turned into a metaprogram, which is applied to all files.

Step 3 In both processes, all test suites are executed. Note that in the manual process, the tests are run after each batch of changed files; in the scripted process, the tests are run once after *all* files were changed automatically.

Step 4 If the test infrastructure signals that errors have been introduced, the applied code changes are reverted and each approach loops back to **Step 1**. Otherwise, for the automated process, all tests succeeding directly implies that the process is finished. The manual process, however, loops back to **Step 2** for the next batch of target files, until the complete codebase has been taken care of.

Step 5 For both approaches, the process ends when all references to STX have disappeared, all tests are succeeding, and an independent colleague gives consent.

The main difference in effort lies in **Step 2**. After problem analysis, a prospective solution is applied to one (or several) files, after which the tests are run. If successful, more files are treated in the same way. In case the tests eventually signal a flaw in the refactoring strategy, all target files have to be reverted to their original state, and a new solution has to be proposed. In the scripted process, a metaprogram is created, which is automatically applied to the complete codebase. If the tests identify an error, only the metaprogram needs to be adjusted. It is worth noting that in the manual process, **Steps 2** through **4** are encountered multiple times, and that the test environment is called in each iteration.

Comparing these ways of working, we identify several differences. The automatic process eliminates the possibility of human errors, and is more consistent. Furthermore, metaprograms can be applied to all target files simultaneously, facilitating quick detection of faults. For the manual case, encountering a flaw in the current solution late in the process would yield all previous manual refactoring effort in vain.

A notable effort that is only applicable to the scripted process is that the software engineer undertaking the refactoring needs to get acquainted with a metaprogramming language, if not already proficient. While we will not attempt to quantify the amount of time this takes as this may differ from person to person, we do note that this need not be very hard or time-consuming.

The number of files that are to be refactored plays an important factor in the total effort for the manual process: as each file requires manual intervention, the total effort for **Step 2** is linear in the amount of files to edit. For the scripted process this is not the case: as a script can be applied to all files directly, this is a constant effort. The effort to create or adapt a refactoring script is also a constant effort, so the total effort of a manual process will always exceed the effort of a scripted process if the size of the codebase is sufficiently large. In general, the return on investment for projects such as ours is largest when there are few patterns with many matches.

We now evaluate the efficiency of the semi-automated refactoring in terms of the number of lines of RASCAL code written relative to the number of lines of C++ and CMAKE code that were affected. In total, a grand total of 6904 SLOC were affected by the semi-automated refactoring. For these changes, we required only 371 lines of RASCAL code, including the CMAKE grammar. In terms of lines of code, our semi-automatic refactoring is almost 19 times more efficient.

5.3 | Analysis of generalizability

The STX framework is a proprietary test framework that is used only within Philips. Therefore, the specific patterns that are used to match in this refactoring are not reusable outside of the company. In Section 4.8, we showed that the RASCAL code was nearly fully reusable on an older product line of the initial target system. Therefore, within Philips, it can be reused to refactor other software suites to phase out STX in favor of GOOGLETEST.

The scripts we wrote to represent edits and apply them to files are, in principle, reusable for other analysis scripts than can produce (a) the location of a change and (b) the text to substitute. This method of applying changes to source files seems new in the RASCAL ecosystem and it could be a valuable addition to the standard library.

The grammar we wrote for CMAKE is reusable in principle in other C and C++ transformation scripts that require information from CMAKE files. However, we wrote the grammar until we could parse the targeted set of files in the test suite. It is not unthinkable that more extensions are required to be able to parse any CMAKE file.

The personal C++ analysis and transformation and RASCAL meta-programming skills we used to create our scripts, however, are usable again. If another large-scale renovation is motivated, we can jump-start the automated refactoring using our knowledge of CLAIR and the pattern matching, analysis and templating features of RASCAL.

5.4 | Lessons learned

5.4.1 | Adopting RASCAL in industry

In this section, we describe several aspects that are relevant for the adaption of RASCAL in industry. We start with the required steps to be able to use a new tool at Philips. We then describe how Rascal is distributed to the engineers. Finally, we elaborate on the effort that is required for the introduction of RASCAL in an industrial setting.

Tool validation

At Philips, we recognize the challenges that come with legacy systems and the promise metaprogramming could bring. For this reason, we supported the work of the second author to create ClaiR and extend Rascal with a C/C++ front-end. The first author arranged formal approval for the usage of Rascal within Philips. The formal approval consists of a number of steps. Note that because Philips is a company that creates medical systems, it has to adhere to the rules created by regulators worldwide (e.g., the Food and Drug Administration [FDA] in the USA). Only tools that are validated and are part of our tool register may be used. The first step of the validation process was to convince all software managers of the business unit that Rascal is an useful addition to the tools that we use. To this end, the first author highlighted the benefits Rascal could bring and that there was no other tool in the tool register with similar capabilities. The second step of our tool validation process consists of creating a document according a prescribed template, in which the name and version of the tool and its intended use need to be filled in. In addition, an initial risk assessment needs to be added. We reasoned that a failure in a Rascal script will always be caught in a later phase of the software development process, and classified Rascal as low risk. The document was then signed by all required managers, completing the validation process.

Tool distribution

Now that we were allowed to use Rascal, we included Rascal in our Eclipse distribution. Next to Rascal, this Eclipse distribution contains several domain specific languages and support other tools; it is distributed via the source code archive. We do it in this way to make sure that all tools are in line with a particular snapshot of the archive. This is especially important because we use trunk-based development and create a branch for each product release. An engineer tasked with fixing a bug for an older product release checks out the branch and immediately has the correct tools at his disposal.

Tool introduction

We continue with an exploration of the magnitude of the required changes when introducing Rascal in a software department. Fowler and Levine[13] describe a model that, given the magnitude of technological change, predicts the learning and time that is required for adoption (Figure 5). Depending on the learning and time that is required for the technological change, the approximated time needed for an organization to adjust is either short or long. We reason that for the adaptation of Rascal within an organization, engineers should acquire new skills. They should be able to use Rascal. The procedures do not need to change as we only automate a process that would be similar as if it would have been conducted manually (cf. Figure 1). As there is no procedural change, also the structure, strategy, and culture of an organization do not need to be changed. Looking at Figure 5, we see that for the adoption of Rascal, limited time is required. This matches the experience of the last author who has a lot of experience in teaching Rascal to university students. He has taught Rascal to students enrolled in a Master's programme. For this course on Software Evolution, students learn Rascal in approximately five days. In our opinion, when applying Rascal the main driver for success is knowledge about the object language (in our case C/C++). If we translate these observations to industry, we see one

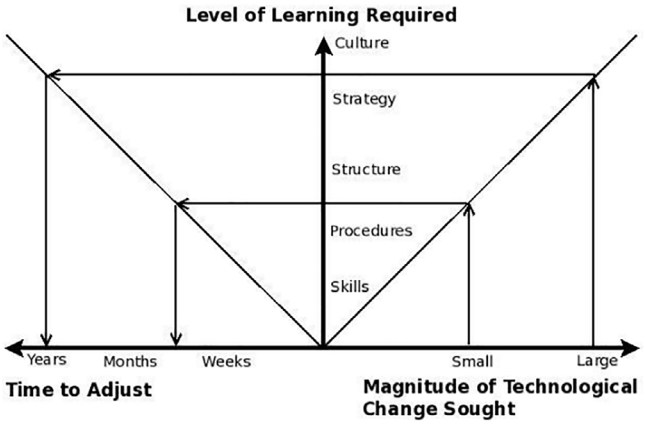

FIGURE 5 Dimensions of change[13]

major difference. In Dutch industry only a minority of software engineers has an academic background, while a majority of software engineers has had a higher professional education. Software engineers with a higher professional education background may experience more difficulty in understanding mathematical concepts of programming languages. As such, we think it might be more challenging to learn RASCAL for these professionals. In practice, project teams typically consist of software engineers from various backgrounds. A engineer with excellent metaprogramming skills can team up with an engineer that is, for instance, a domain expert. Even with only a single engineer that has metaprogramming skills per project team, we think that such a team can be very productive in performing automated maintenance activities.

C/C++ are notoriously difficult languages for automated transformation tools. With the recent addition of CLAIR to RASCAL, we think we have solved some of the challenges. Giving the experiences described in this article, we think RASCAL is ready for wide-spread adoption in industry. The existence of these kind of tools is not widely known by industry. For this reason, we think the RASCAL community can improve by advertising the possibilities to industry.

5.4.2 | Improving the migration process

The migration process could be improved as follows.

In **Step 2** of Figure 1, we run the scripts on the source code. The automated process is iterative. Hence, **Step 2** is performed multiple times. RASCAL will use CLAIR to parse the source files using CDT and to create an AST. This tree is used by the metaprogram to create a list of changes that need to be applied to the source file. Depending on the contents of a source file, creating an AST may take up to 10 seconds. As the resulting AST will always be the same, we could improve the process by only parsing files once and storing ASTs as files on disk.

We have run the STX test suites sequentially on a single PC. Some of these STX test suites take up to 45 min to execute; running all STX test suites takes a full night. To shorten the feedback loop, we could run the refactored test suites in parallel on multiple offload systems.

While working with the legacy code and preforming the migrations, we have learned a lot about the legacy system. This is inherent to the process. New refactorings on the same legacy system are likely to take less time, because of the things we have learned about the system with the reported activities.

5.5 | Related work

In this part, we position alternatives to RASCAL, the implementation language of our refactoring, by describing related work. We have chosen RASCAL as our implementation language for our refactoring. In principle, we could have picked other metaprogramming systems that allows simultaneous analysis of multiple languages, such as ASF+SDF,[14] Stratego,[15] TXL,[16] DMS,[17] or, more geared to C++, Proteus[10] or CodeBoost.[18] The current section is not a feature comparison between all these systems, but rather an enumeration that documents the wealth of metaprogramming systems available, which may or may not be relevant to the reader.

Already in 1995, Aigner and Hölzle[19] describe a case in which they refactored C++ language instances to improve the performance of executables. They accomplish this by implementing a source-to-source transformation, rewriting virtual function calls using the `inline` keyword. The field started as a spin-off of Compiler Construction techniques: transformation of (annotated) abstract syntax trees. CodeBoost[18] is a source-to-source optimizer for C++ that shares many features with the technology described here.

Mass maintenance on legacy software took to flight in earnest around the year 2000: the Y2K problem motivated a principled and automated approach to maintaining large legacy code bases. The focus was on COBOL systems. Due to the omnipresence of the Y2K problem in COBOL systems and the abundance of large COBOL at that time, Van den Brand promotes the use of "software renovation factories": systems taking context-free grammars as input, in which users can create bespoke code analyses and transformations.[20] After the initial Y2K fix, applications of the developed technology fanned out into more complex code enhancements. For example, Veerman[21] used sets of rewrite rules to transform and modernize legacy COBOL code, transforming spaghetti code into well-structured programs. He implemented the transformations in a predecessor of RASCAL, called ASF+SDF.[22] Other systems from that era which are still applied to legacy software analysis and transformation in both industry and academia are TXL (Turing eXtender Language),[16] DMS,[17] and Stratego/XT.[15] All these systems provide pattern recognition and substitution on abstract or concrete syntax trees.

ClangMR is a tool based on the combination of the Clang compiler and the MapReduce parallel processor.[23] The transformations are, again, based on the AST structure. ClangMR has been applied on large C++ codebases at Google to refactor callers of deprecated APIs. Mooij et al. use small iterative code refactoring steps before extracting models.[4] The rationale is that the intermediate refactoring steps increasingly reduce noise in the code, making model extraction easier. They also used the Eclipse CDT parser like we did in the current article.

"High-fidelity" source-to-source transformations were pioneered by Vinju[24,25] in the ASF+SDF system, by Waddington et al. with the Proteus C/C++ transformation tool,[10] and by Thompson with the HaRe system[26] for refactoring Haskell. The goal is to transform the source code without losing arbitrary indentation or source code comments. In the current article, by generating edit scripts that are applied to the unprocessed source files we circumvent the loss of indentation and comments that is associated with a compilation pipeline that goes through different (abstract) representations.

The method of generating edit scripts is also the method used by the Eclipse Refactoring Framework.[27] A "refactoring" is a source-to-source transformation with the goal of improving internal code structure without changing the observable behavior. The input of a generic refactoring tool could be any code that passes the compiler. The complexity of generic automated refactoring tools lies therefore in checking the preconditions for such a code change and making sure that the applied change will not change observable behavior. This requires advanced theoretical models, such as for example "type constraints."[28] Our current work however does not require this since we are writing transformations for a specific system with specific code patterns. Steimann[29] explores an interesting middle ground where a generic system of semantical constraints (on Java code) can be used to implement "ad-hoc" refactorings comparable to our use case. However, we did not require such deep analysis to know that our transformations were correct; also the C++ language would be much harder to model semantically than Java.

Object-language-agnostic systems such as TXL, DMS, Stratego/XT, and ASF+SDF require the specification of a parser (usually in some form of BNF). Writing a parser for C++ is a daunting exercise, however. The language is inherently ambiguous and its disambiguation requires (deep) semantic analysis. Also the C preprocessor literally adds a level of complexity to the analysis of C++ input programs. The Proteus system provides parsing of C++ to the Stratego/XT environment by parsing C++ with an external parser and providing the ASTs to the rewriting engine. The current article uses ClaiR[8] in a similar fashion by reusing the Eclipse CDT parser in front of the rewriting features of Rascal. The C-Transformers project used a post-parse disambiguation stage to simplify C++ parse forests to single trees.[30]

The srcML family of tools[31,32] reuses open compilers to markup source code with abstract syntax tree nodes, for different programming languages including C and C++, but not CMake. This too enables high-fidelity source code analysis and transformation, using general XML tools such as XPath or XSLT.

To summarize the related work: all systems for large scale source-to-source transformations lean heavily on parsers to produce (abstract) syntax trees. The parser does most of the "heavy lifting." After this, they provide pattern matching and traversal facilities, and sometimes tree substitution. The high-fidelity source-to-source systems either edit the input file using edit scripts, or retain all information during parsing in a concrete syntax tree. The current transformations of this article also required simpler scripting tasks: file handling and such. Since Rascal is a programming language, the scripting could be done close to the code analysis and transformation code in the same language.[33] Other systems for AST analysis, such as Stratego/XT and ASF+SDF require scripts in an external scripting language such as Python or Bash.

6 | CONCLUSION

6.1 | Results

This article describes our approach to the semi-automatically refactoring of a legacy software system. As compared to a hypothetical manual refactoring, we have argued that the benefits of our automatic approach outweigh its own particular challenges: the automated approach has granted us repeatability, (limited) generalizability, elimination of human error introduction, and early detection of errors, at the cost of having to acquire experience in metaprogramming. By automating our refactoring, we have solved our being stuck and allowed ourselves to perform a useful, necessary refactoring that would not have been carried out manually. In our particular refactoring case, our 371 lines of metaprogramming code identified and changed approximately a 19-fold of lines of target code, indicating that even for one-off refactorings, automating the job makes sense.

6.2 | Lessons learned

In this section, we describe our lessons learned.

We created a parser for CMAKE. Parts of the CMAKE parser could be reused and extended for future refactorings. Our RASCAL scripts for transforming the legacy code are tailored for this case. Since the refactorings are highly context-specific, they are not easily generalizable and cannot deal with any other potential context. In fact, their specificity is part of what makes them so powerful, and is actually an enabler to perform these kind of refactorings.

In this article, we presented a way to write changes to a file without the need to unparse or pretty print the abstract syntax tree. This way of changing source files could be added to the standard library of RASCAL.

In the remainder of this section, we split up the lessons learned for software engineers, software managers, and tool manufacturers.

Implications for software engineers

Legacy code is code that is typically written by someone who is retired, works at another department, or has left the company. This code is written with old and obsolete tools and technologies. Replacing or rejuvenating legacy code is labor-intense and intellectually uninteresting. For software engineers, maintenance projects are not popular, because for most it is much more attractive to create something new. Software maintenance can become fun when it consists of creating state-of-the-art RASCAL scripts that do the repetitive work in an automated fashion. In fact, the software engineer is writing challenging new code in the form of RASCAL metaprograms.

Implications for software managers

The fact that software maintenance projects can become more attractive for software engineers is a big plus for software managers. It could become more easy to convince the best employees to work on maintenance projects.

Maintenance projects are important for existing customers and for reusing existing components in new products.[1] From literature, we also know that a large portion of time is spent on maintenance. The promised productivity gains from applying metaprogramming techniques directly results in a more efficient software development process.

Introducing RASCAL in a department requires some change management. Our experience from the financial world, in which RASCAL has been used for a longer period of time, is that it works best to pair domain experts with metaprogramming experts. Both can complement each other and be successful from the start. We think initial success is key to a successful adoption of RASCAL in industry.

For software managers, it is important to have someone to go to when there are issues with a tool. The availability of commercial support is a requirement for the adoption of a tool. For RASCAL, such support is available.

Implications for RASCAL manufacturers

The CLAIR C/C++ front-end was recently added to RASCAL. CLAIR is the enabler for adopting RASCAL in industry. However, the awareness of RASCAL in industry can be improved. Improved awareness can be achieved by not only publishing academic papers, but also publishing in nonacademic magazines read by industry experts.

Our experience is that questions about RASCAL on StackOverflow[‖] are answered in a few hours. The RASCAL community should keep on doing this. However, we think that the need for asking questions could be reduced when error reporting is improved.

The documentation of CLAIR could be improved. For the core RASCAL language there is a well-written and well-maintained tutor available online.[**] For applying CLAIR, one has to look at the source code when creating metaprograms.[††] For an improved user experience, CLAIR (and other newly created language front-ends) should be documented in a similar fashion as the core language.

[‖] https://stackoverflow.com/questions/tagged/rascal
[**] https://docs.rascal-mpl.org/
[††] https://github.com/cwi-swat/clair/blob/master/src/lang/cpp/AST.rsc

6.3 | Future work

The case described in this article involved the refactoring of test suites. All test cases of the test suites passed before refactoring and passed after refactoring. For legacy production code, the test coverage is typically unsatisfactory, hampering the refactoring process, since refactoring code in an industrial setting which does not have good test coverage can be tricky. Writing a formal proof of the equality of the operational semantics before and after refactoring is not realistic for an industrial software engineer. To check whether new code behaves as intended, we would like to investigate the ability to automatically generate a test suite. The test suite should cover the code that needs to be transformed. It would provide some confidence that the transformation is indeed a refactoring when the generated test cases pass before and after the transformation.

AUTHOR CONTRIBUTIONS

Mathijs Schuts carried out the refactoring, drafted the first version of the manuscript, and revised the paper. Rodin Aarssen and Jurgen Vinju assisted with the development of the Rascal metaprograms, and restructured the paper. Rodin Aarssen revised the manuscript and added several sections. Paul Tielemans helped with organizing and proofreading the paper, and provided input on the industrial perspective of the contribution.

CONFLICT OF INTEREST

The authors declare no potential conflict of interests.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are available from the corresponding author upon reasonable request.

ORCID

Rodin T. A. Aarssen https://orcid.org/0000-0002-9077-5517

REFERENCES

1. Brinksma E, Hooman J. Dependability for high-tech systems: an industry-as-laboratory approach. Proceedings of the 2008 Conference on Design, Automation and Test in Europe; 2008:1226-1231; ACM, New York, NY.
2. Breivold HP, Crnkovic I, Larsson M. A systematic review of software architecture evolution research. *Inf Softw Technol*. 2012;54(1):16-40. doi:10.1016/j.infsof.2011.06.002
3. Veerman NP. Automated mass maintenance of software assets. Proceedings of the 11th European Conference on Software Maintenance and Reengineering; 2007:353-356; IEEE.
4. Mooij AJ, Ketema J, Klusener S, Schuts M. Reducing code complexity through code refactoring and model-based rejuvenation. Proceedings of the 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering; 2020:617-621; IEEE.
5. Easterbrook S, Singer J, Storey M-A, Damian D. Selecting empirical methods for software engineering research. In: Shull F, Singer J, Sjöberg DIK, eds. *Guide to Advanced Empirical Software Engineering*. Springer; 2008:285-311.
6. Klint P, Van der Storm T, Vinju J. RASCAL: a domain specific language for source code analysis and manipulation. Proceedings of the 2009 9th IEEE International Working Conference on Source Code Analysis and Manipulation; 2009:168-177; IEEE.
7. Klint P, Van der Storm T, Vinju J. EASY meta-programming with rascal. leveraging the extract-analyze-synthesize paradigm for meta-programming. Proceedings of the 3rd International Summer School on Generative and Transformational Techniques in Software Engineering; 2010:222-289; Springer.
8. Aarssen R. cwi-swat/clair: v0.1.0. 2017. 10.5281/zenodo.891122
9. Cheng P, Chulani S, Dang YB, et al. Jazz as a research platform: experience from the software development governance group at IBM research. Proceedings of the 1st International Workshop on Infrastructure for Research in Collaborative Software Engineering; 2008; ACM, New York, NY.
10. Waddington DG, Yao B. High-fidelity C/C++ code transformation. *Electron Notes Theor Comput Sci*. 2005;141(4):35-56. doi:10.1016/j.entcs.2005.04.037
11. Aarssen RTA, Van der Storm T. High-fidelity metaprogramming with separator syntax trees. In: Bach PC, Hu Z, eds. *Proceedings of the 2020 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation* , ACM; 2020:27–37.
12. Aarssen RTA, Vinju JJ, Van der Storm T. Concrete syntax with black box parsers. *Art Sci Eng Program*. 2019;3(3). 10.22152/programming-journal.org/2019/3/15
13. Fowler P, Levine L. A conceptual framework for software technology transition. CMU/SEI-93-TR-031: Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA; 1993.

14. Van den Brand M, Van Deursen A, Heering J, et al. The ASF+SDF meta-environment: a component-based language development environment. *Electron Notes Theor Comput Sci*. 2001;44(1):3-8. doi:10.1016/S1571-0661(04)80917-4
15. Visser E. Program transformation with stratego/XT. In: Lengauer C, Batory D, Consel C, Odersky M, eds. *Domain-Specific Program Generation: International Seminar*. Springer; 2004:216-238.
16. Cordy JR, Dean TR, Malton AJ, Schneider KA. Source transformation in software engineering using the TXL transformation system. *Inf Softw Technol*. 2002;44(13):827-837. doi:10.1016/S0950-5849(02)00104-0
17. Baxter ID, Pidgeon C, Mehlich M. DMS®: program transformations for practical scalable software evolution. Proceedings of the 26th International Conference on Software Engineering; 2004:625-634.
18. Bagge OS, Kalleberg KT, Haveraaen M., Visser E. Design of the CodeBoost transformation system for domain-specific optimisation of C++ programs. Proceedings of the 3rd IEEE International Workshop on Source Code Analysis and Manipulation; 2003:65-74; IEEE.
19. Aigner G, Hölzle U. Eliminating virtual function calls in C++ programs. Proceedings of the 10th European Conference on Object-Oriented Programming; 1996:142-166; Springer, New York, NY.
20. Van den Brand M, Sellink A, Verhoef C. Generation of components for software renovation factories from context-free grammars. *Sci Comput Program*. 2000;36(2):209-266. doi:10.1016/S0167-6423(99)00037-4
21. Veerman NP. Revitalizing modifiability of legacy assets. *J Softw Maint Evol Res Pract*. 2004;16(4-5):219-254. doi:10.1109/CSMR.2003.1192407
22. Van den Brand MGJ, Van Deursen A, Heering J, et al. The ASF+SDF meta-environment: a component-based language development environment. In: Wilhelm R, ed. *Compiler Construction*. Lecture Notes in Computer Science. Springer; 2001:365-370.
23. Wright HK, Jasper D, Klimek M, Carruth C, Wan Z. Large-scale automated refactoring using ClangMR. Proceedings of the 2013 IEEE International Conference on Software Maintenance; 2013:548-551; IEEE.
24. Van den Brand MGJ, Vinju JJ. Rewriting with layout. Proceedings of the 1st International Workshop on Rule-Based Programming; 2000; ACM, New York, NY.
25. Vinju JJ. Type-driven automatic quotation of concrete object code in meta programs. In: Guelfi N, Savidis A, eds. *Proceedings of the 2nd International Workshop on Rapid Integration of Software Engineering Techniques*. Springer; 2005:97-112.
26. Li H, Reinke C, Thompson S. Tool support for refactoring functional programs. Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell; 2003:27-38; ACM, New York, NY.
27. Fuhrer R, Keller M, Kiezun A. Advanced refactoring in the eclipse JDT: past, present, and future. Proceedings of the 2007 1st Workshop on Refactoring Tools; 2007:30-31; ACM, New York, NY.
28. Tip F, Fuhrer RM, Kiezun A, Ernst MD, Balaban I, De Sutter B. Refactoring using type constraints. *ACM Trans Program Lang Syst*. 2011;33(3):9:1-9:47. doi:10.1145/1961204.1961205
29. Steimann F. Constraint-based refactoring. *ACM Trans Program Lang Syst*. 2018;40(1):1-40. doi:10.1145/3156016
30. Borghi A, David V, Demaille A. C-transformers: a framework to write C program transformations. *XRDS Crossroads ACM Mag Stud*. 2006;12(3):3-3. doi:10.1145/1144366.1144369
31. Collard ML, Decker MJ, Maletic JI. Lightweight transformation and fact extraction with the srcML toolkit. Proceedings of the 2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation; 2011:173-184; IEEE.
32. Collard ML, Maletic JI, Robinson BP. A lightweight transformational approach to support large scale adaptive changes. Proceedings of the 2010 IEEE International Conference on Software Maintenance; 2010:1-10; IEEE.
33. Heering J, Klint P. Towards monolingual programming environments. *ACM Trans Program Lang Syst*. 1985;7(2):183-213. doi:10.1145/3318.3321

How to cite this article: Schuts MTW, Aarssen RTA, Tielemans PM, Vinju JJ. Large-scale semi-automated migration of legacy C/C++ test code. *Softw Pract Exper*. 2022;1-38. doi: 10.1002/spe.3082