Qualify First! A Large Scale Modernisation Report

Leszek Włodarski
mBank, Warsaw, Poland
Leszek.Wlodarski@mbank.pl

Boris Pereira, Ivan Povazan, Johan Fabry, Vadim Zaytsev
Raincode Labs, Brussels, Belgium
firstname@raincodelabs.com

***Abstract*—Typically in modernisation projects any concerns for code quality are silenced until the end of the migration, to simplify an already complex process. Yet, we claim from experience that prioritising quality above many other issues has many benefits. In this experience report, we discuss a modernisation project of *mBank*, a big Polish bank, where bad smell detection and elimination, automated testing and refactoring played a crucial rule, provided pay-offs early in the project, increased buy-in, and ensured maintainability of the end result.**

I. Introduction

It is always tempting to migrate the legacy system first, and then perform quality improvements with modern technology. *Raincode Labs* routinely takes on such projects, and observes how the wishes of the customers are mostly to migrate "as is" and *plan* to improve the quality later. There are mostly two linked risks here: (1) migrating bad code makes the migration trickier, and (2) the plans never realise themselves.

The project we report on in this paper, concerns migrating the central banking system of *mBank* (CBS from now on) from an old technological stack to the .NET Framework. We show that prioritising quality resulted in significant benefits up front and incomparably more maintainable code. Quality as a migration enabler is visible in three ways:

1) Showing early pay-offs at different levels, gaining support for the migration project.
2) Minimising migration-induced disruption since documentation and testing are deployed before migration.
3) Reducing maintenance effort of the legacy system, freeing resources for the migration as such.

We found that bad smell detection and removal (§ III) was met positively by the developers who acquired incentives to create better code to prevent future problems, and thus also saw the migration activities positively. ∼3600 tests were written by developers and 680 by analysts, to aid maturity and comprehension of the legacy system (§ IV). Finally, refactoring out over 99% of 37405 `GOTO` statements (§ V), dramatically increased readability of the code.

II. Context

mBank is one of the top banks in Poland with 5.3M customers, 6.3% market share in loans and 5.9% market share in deposits. It was founded in 1986 and was the first fully-digital bank in Poland in 2000. Unlike many banks, *mBank* does not extensively rely on customising existing software packages. Instead, as a way of developing and maintaining competitive advantage, since 2005 *mBank* software is internally developed by several departments (300 FTE total). The codebase contains both banking-specific packages and common packages (*e.g.,* a CRM system) with a banking flavour. In this report we focus on the modernisation of the CBS of the corporate branch (∼22'000 clients), which is separate from the retail branch.

Client products come with *hundreds* of degrees of freedom, stemming from different customer profiles: some only perform end-of-month wages payments (a large set of simple low-amount transactions, where a single failure is undesirable but acceptable); others perform transactions with much larger amounts, at unpredictable times, and failure or even delay in processing any of them, means losing the client. Such different profiles, the sheer number of features and the combinatorially explosive number of possible interactions among them has required extensive program management and manual testing.

Beyond day-to-day operations, the focus on customisability also has an impact on long-term survivability of the software. The major source of this impact is that 3MLOC of the core banking software is written in a fourth generation language (4GL) that we will call $\mathcal{B}$, detailed to some extent below.

The platform developed at *mBank* is based on a MultiValue database system [1], which is a data model predating relational databases. In short, MultiValue databases can be viewed as relational multidimensional databases where columns can hold lists of values instead of only atomic values. There have been multiple MultiValue databases available on the market, each with its own set of tools and/or 4GLs to aid in development of applications that use the database. For legal reasons, we cannot disclose which product is in use at *mBank*, and call it $\mathcal{M}$. The following two elements of this system are relevant:

1) $\mathcal{B}$ is an imperative, dynamically typed 4GL with automatic memory management. It has structured programming features like `LOOP` and `FOR` statements, plus simpler control flow statements like `GOTO` and `GOSUB`. It has strings, numbers and dynamically sized arrays.
2) $\mathcal{U}$ is a UI and application server that allows $\mathcal{B}$ programs to communicate with a GUI written in a more modern language through a stateful conversational telnet session.

The flexibility of $\mathcal{M}$ and simplicity of $\mathcal{B}$ allowed continual product development for 20+ years by people of varying skill sets, for different banks. As a consequence, the production code is inconsistent, written by people who no longer work at *mBank*, and is of varying quality.

Quality issues were made evident by an incident in production on a vast scale, which caused the bank to stop processing orders for several hours. There were multiple failures in the

system that compounded the original issue: inconsistencies in the database. A postmortem analysis revealed that the root cause was the multivalue nature of the database combined with the lack of strong typing in the database and in $\mathcal{B}$. This allowed inconsistencies to be created by production $\mathcal{B}$ code of dubious quality, causing a chain reaction down the line. This incident led to the decision to migrate the complete system, *i.e.,* database and production code, to new technologies and consider code quality [2] and technical debt [3] in the process.

Two major factors constrained possible migration routes: (1) business and product development may not be frozen during the migration; (2) experience present in other teams within *mBank* needs to be put to good use. Thus, four different options were considered to renovate the $\mathcal{M}$ system:

1) **Migrate to another CBS.** Lessons learnt by *mBank* from the previous four migrations are that this would not solve the current problem at hand, given the unique nature of the software of the corporate branch.
2) **Migrate to the CBS of the retail branch.** It is very desirable to migrate from two CBSs to just one, but estimations showed this option to be so resource intensive that it would block product development.
3) **Upgrade to the newest version of $\mathcal{M}$.** While it would somewhat modernise the CBS, again, the effort that it entails would block product development.
4) **Customised migration to a modern environment.** This option would be able to maintain the inherent value of the system by keeping the resulting outside behaviour identical. This process would then also consider addressing the technical debt that has been accumulated.

The decision made was to take the last option, with .NET and C# as the target technologies since *mBank* has considerable expertise there. It was also known from previous migration projects how modern tools boost productivity and integrability of new developers. It was made an explicit requirement for all project parts to deliver benefits early on.

III. Bad Smell Detection and Remediation

A source code analysis and transformation infrastructure for $\mathcal{B}$ was developed by *Raincode Labs* at the start of the project. Its first use was to perform bad smell detection, as an initial way to locate and decrease technical debt. *mBank* developed several bad smell detection components and integrated them to the SonarQube [4] code quality dashboard, including:

- Simple dead code detection.
- Use of the `GOTO` statement.
- Variables that should be read-only are being assigned to.
- Variables are written but never read.
- Variable increment using the ++ postfix operator.

The last item begs further explanation. During the construction of unit tests (§ IV), it was discovered that the ++ operator in $\mathcal{B}$ causes a loss of precision because it changes the underlying integer value of the variable to a floating point number. Thus, after a large number of mathematical operations on these variables, their actual value drifts away from their expected value. Hence, the use of ++ is declared a bad smell in $\mathcal{B}$, and *mBank* strives to remove it from its codebase.

After each commit, the bad smell detectors are run automatically and visualised in the dashboard. Smell removal happened on an opportunistic basis in between more urgent activities, and included adding unit tests and functional tests before refactoring the code. The following lessons were learnt:

1) Bad smell detection forces developers to change their approach on how to develop and maintain code by adding the objective of producing good quality code (as measured by a low number of bad smells).
2) Removing bad smells causes the developers to gain more knowledge of the source code, since they need to inspect code that they would otherwise not look at. This knowledge was useful later in the project.
3) As bad smells indicate possible issues in the code, removing them reduces possible issues in production. This would reduce tedious maintenance tasks and free up more resources for the migration project itself.
4) Improving the code quality before the transformation leads to better code quality of the transformed code. The prime example is `GOTO` (see § V–VI for details).

Developers are positive about the bad smell detection because they see it as a way to ensure their code has fewer possible bugs. It potentially lowers the amount of tedious maintenance work to be done in the future, and hence has increased their project buy-in.

IV. Adding Unit and Functional Tests

Due to its architecture, $\mathcal{U}$ had scaling issues before this project was started: UI apps could not execute queries concurrently, disconnected sessions were freezing the app, etc. To address this, a new UI and application server $\mathcal{W}$ was developed in C# that is stateless, focuses on scalability, and allows for basic monitoring and diagnostics. $\mathcal{W}$ is now in production and internal client GUIs are rewritten to take advantage of it.

Because $\mathcal{W}$ no longer uses a conversational-style stateful interaction model between $\mathcal{B}$ programs and the UI, it was demonstrated early on in the project that $\mathcal{B}$ code needed to be refactored to correctly interface with $\mathcal{W}$. This need for refactorings then placed an emphasis on the issue of the absence of any (automated) testing framework or rigorous testing culture. To address this need, two different testing frameworks have been developed at *mBank*: one for unit tests and one for functional tests:

1) The unit testing framework of *mBank* was developed in C#: developers add structured comments to their $\mathcal{B}$ code, which serve as annotations, specifying inputs, outputs, assertions, etc. The *mBank* developers were happy to use the framework once it was introduced, and the number of tests grows constantly (reached 3600 green unit tests after one year). Originally planned as a refactoring aid to introduce $\mathcal{W}$, these tests have added extra value on their own. Issues in production now almost exclusively occur in code not yet covered by unit tests.

```
FOR goto IN Tools.GetGotoStatements(ROOT) {
  loop := FIRST s IN goto.Ancestors() WHERE s IS LoopStatement OR s IS ForStatement;
  IF loop != VOID {
    label := goto.Target.Ref();
    IF label == loop.Statements[LAST]
      OR ( label IS LoopStatement AND label == loop.Statements[0] )
      OR ( label IS LoopStatement AND label.NextStatement == loop ) {
    Tools.ReplaceStatement(goto, "CONTINUE");
} } }
```

Fig. 1. Core of the specification of the GotoInLoopToContinue refactoring.

2) The functional test tool was targeted at analysts who can specify workflows with inputs and outputs expected at different stages. Automating this process removed the natural limitations on the number of tests, and in the first year they wrote 680 different functional tests, with a constantly raising rate of creation. A typical example of a problem that only was solved with functional tests were several kinds of deadlocks we found and removed thanks to them — deadlocked programs frozen in production get restarted by system administrators too promptly to analyse the causes. Beyond enabling bugfixes, another obvious consequence was knowing exactly which bugs are present before migration and hence are not introduced by the migration process.

This work has had an immediate pay-off: code in production became better tested and documented, new bugs were discovered, isolated and fixed. Both developers and analysts have reaped the benefits from these frameworks which has increased their support for the migration project. Now, after the successful end of the project, developers are still actively adding unit tests, while analysts are adding functional tests—it has become a part of their process.

V. $\mathcal{B}$ to $\mathcal{B}$ Transformation

It is known from both the experience of *Raincode Labs* and academic literature [5] that any language translation adds an overhead to the generated code in terms of infrastructure needed to emulate the functionality of the original language. Hence, applying $\mathcal{B}$ to C# transformations directly would have caused a code quality drop. To address this, we preceded them with $\mathcal{B}$ to $\mathcal{B}$ refactorings that improve the initial code quality. The framework used to implement them is a proprietary technology developed at *Raincode Labs* to support $\mathcal{B}$. It works similarly to existing open-source alternatives such as ASF [6], TXL [7], Stratego [8] or Rascal [9].

One of the often used constructs in $\mathcal{B}$ is unstructured `GOTO`, which does not exist in C# and has to be crudely emulated, piling the emulating infrastructure on top of an already condemned practice [10]. Yet, $\mathcal{B}$ **does** support several structured programming constructs: `FOR...REPEAT` and `LOOP...REPEAT` with both `BREAK` and `CONTINUE`, as well as `GOSUB`/`RETURN` for visiting a subroutine and returning from it.

Refactoring 99.46% of all 37405 `GOTO` statements took writing 40 transformations for different patterns of `GOTO` usage—an example of one can be seen on Figure 1. A single transformation took 20–340 LOC to specify, including comments. Transformations were made to work collaboratively in small steps: *e.g.,* one could introduce a flag variable in order to separate the point of decision making from the actual jump, so that the actual `GOTO` statement can be picked up by another transformation to be turned into a `CONTINUE`.

VI. $\mathcal{B}$ to C# Transformation

C# does have a `goto` statement, but it is missing support for $\mathcal{B}$ features of `GOTO` that are essential to us (*e.g.,* it is not possible to `goto` out of a loop in C#). Hence, we had to develop our own emulation of $\mathcal{B}$-style `GOTO` in C# to handle the 203 remaining `GOTO` statements in the $\mathcal{B}$ code. Hopefully this section can give an indication of the complexity of the resulting code if we had *not* performed the cleanup of `GOTO`, and how it would have impacted the code after migration.

In the spirit of making transformation steps as simple as possible, the overall transformation strategy of $\mathcal{B}$ to C# is to perform a one-to-one statement translation, also known as the *syntax swap* [5]. The architecture of the transformation is a depth-first traversal over the abstract syntax tree, where for each kind of node in the tree there is a translation to their equivalent C# statement. This architecture allows for easy testing and debugging of the translation since a statement in the produced C# code is easily traceable to a statement in the $\mathcal{B}$ code and a specific part of the translation.

Each transformation from $\mathcal{B}$ to C# takes from one line to almost a thousand lines. The simplest ones are arithmetic operations since they are conceptually identical in $\mathcal{B}$ and C#. Built-in DSL features of $\mathcal{B}$, like opening a file, are translated to appropriate library calls, which are collected into a separate runtime library shipped together with the compiled code. Essentially this runtime represents the language difference: features native to $\mathcal{B}$ but not included in C#.

The only exception from the one-to-one paradigm is the treatment of the remaining `GOTO`s, since they cannot be deconstructed to mere library calls. We address them in a two-part process: (1) moving labels outside nested control flow constructs, and (2) splitting code in methods and adding a control flow dispatch manager. For the former step, we make sure labels only live on the outmost level of the AST: for example, if there is a label inside an `IF`, we move it and all the statements following it, outside of the `IF`, add another

```
IF [x] THEN                 IF [x] THEN
  [a]                         [a]
                              GOTO L1
  L1:
    [b]
ELSE                        ELSE
  [c]                         [c]
END                         END
                            NewLabel:
[d]                           [d]
                            L1:
                              [b]
                              GOTO NewLabel
```

Fig. 2. Illustration of code changes for bubbling a label out of an IF Left: before, right: after. Note the additional two GOTOs and label.

```
                            NewForLabel:
FOR i = x TO y              i = x
                            IF i > y THEN
                              GOTO NewEndFor
                            END
  [a]                       [a]
  L1:                       L1:
    [b]                       [b]
                              i++
END                           GOTO NewForLabel
                            NewEndFor:
```

Fig. 3. Illustration of code changes for bubbling a label out of an FOR Left: before, right: after. Note how the FOR is removed.

label after the END and add two GOTO statements to preserve the control flow (Figure 2). FOR loops are destroyed by this transformation, and become IFs with GOTOs (Figure 3).

Bubbling labels up to the root of the AST has an **negative** impact to code quality: it adds more GOTO statements and destroys some structured loops. However, it is **unavoidable** since C# is an structured OO language that does not have anything even remotely resembling unstructured jumps to arbitrary locations.

After all the labels are propagated out of code blocks, they end up on the top level of the AST. Essentially, each program becomes segmented by labels. To strengthen this segregation, we traverse all such segments that do not end with a GOTO, and add an explicit GOTO to the next segment (this refactoring obviously preserves the semantics and degrades the code quality further). Then, each segment is transformed to a C# method which performs as usual for continuation-passing style, returning the name of the label it used to transfer control to. Lastly, we add a special dispatch method that calls the "first" method, analyses its return value and keeps calling other methods until the end label of the program is reached.

Handling of local subroutines with GOSUB conforms to the same scheme: each GOSUB is translated to a call of the dispatch method with the name of the subroutine label as an argument. Thus, the $\mathcal{B}$ GOSUB call stack is recreated in C# through dispatch method calls, and GOSUB and GOTO statements can peacefully coexist (*e.g.,* a GOTO into a subroutine will result in eventually returning from it). A RETURN from a subroutine is translated to returning a special label to the dispatch method, to signal that the execution of the dispatch method itself needs to terminate, ending the subroutine call.

Clearly, supporting this framework of GOTO dispatches adds significant technical debt to the resulting C# code, since we are essentially working against the language, bending it to do things it was designed to help avoiding. Jumping outside the loop results in introducing artificial control variables, segmenting the code in methods according to labels is not idiomatic—these things decrease readability and thus maintainability of the code. Besides possible maintenance problems, we may experience problems at runtime as well, since using a dispatch method yields in runtime overhead and subroutine calls now require two stack frames instead of one. We have not performed any profiling or benchmarking yet to see whether this problems will be of any significance in practice.

Let us recall that the initial codebase contained 37405 GOTO statements and 94477 labels in 5921 programs (out of 14929). Applying the $\mathcal{B}$ to C# transformations directly, increases the numbers to 49422 GOTOs and 103348 labels, breaking the code in correspondingly many methods. Instead, because of our quality-first focus, we went down to 203 GOTOs in 24 programs, which became 499 GOTO statements after the label lifting process (still a mere 1.3% of the original numbers). The numbers show that without the quality-first focus, we end up with **99 times more boilerplate LOC** to handle the infrastructure (returning to the dispatch method and calling the segment method), and the code would have been split into 47% more sections.

VII. Related Work

Given the context of this work being an industrial experience report, we restrict ourselves to merely listing related areas of prior research and providing starting pointers to the readers interested in the context of this work, instead of diving into comparison of details.

One obviously related area with a substantial body of research results accumulated over the last decades, is software migration: of code, data [11], tools [12], tests [13], API [14], etc. From our point of view, the most realistic industry-aware overview of the field was provided by Terekhov and Verhoef [5], and the most recent case study in migrating banking software reported by Khadka et al. [15]. The software quality aspect of migration of legacy software systems was discussed, among others, by Matthes et al. [16]. Many challenges we have faced in translating $\mathcal{B}$ to C#, were exactly the same as the issues of translating COBOL to Ada [17], COBOL to Java [18], [19] or PL/I to Java [20]. 4GL-related challenges were recently listed by Zaytsev [21].

For bad smell detection we refer to their original definition [22] and to a recent survey [23]. The concept is broader than any particular smell such as dead code, clones or GOTOs, and narrower than technical debt [3] which is a

metaphor for any hasted shipping of imperfect code with the intention of eventually fixing it. Detecting and eliminating smells for us was considerable and tedious work, but 4GLs are nowhere nearly as challenging in that aspect as modern dynamic languages like PHP [24] or JavaScript [25].

The "small step" iterative transformation of the source code that we have explained in § V, is formalized as term rewriting [26]. Our in-house developed DSL functions similarly to other term rewriting implementations like mentioned before [6], [7], [8], [9].

VIII. Conclusion

In this paper we have reported on the modernisation effort of the central banking system (CBS) of the corporate branch of *mBank*. In contrast to other projects treated at *Raincode Labs*, in this effort the decision was made to *first* improve the quality of the codebase, and after that engage in modernisation work. We have shown how this upfront quality improvement work has yielded positive results from the beginning of the project on, and how it has enabled the modernised code to be of orders of magnitude better quality than it would have been without the quality improvement. By now the modernisation project has successfully finished: the application server $\mathcal{U}$ is in production, with both performance and scalability on levels beyond satisfactory, the smell detectors and the testing frameworks are in continuous daily use. The $\mathcal{B}$ code translated to C# was gradually put in production, after some more work on addressing the database and other issues we have no space to discuss (it was related to preventing issues like the one explained in the introduction, in the future).

Let us recall the three parts of this large project that we were able to fit into the space for this text. First, we presented language-specific bad smell detectors that are now run at each commit, giving an incentive to the developers to improve code quality continuously. The early pay-off here was that code in production is cleaned up through routine everyday development actions. Second, we talked about the unit testing and functional testing frameworks that were needed to put the new UI and application server in production. Measurable early pay-offs of this part was the removal of existing bugs in production software, such as program deadlocks. Third, we talked about the code transformation itself, focusing on the automatic removal of spaghetti code with `GOTO` statements. We gave an outline of the code transformations that removed 99% of the 37405 statements, have shown how this code is translated to a modern programming language, and argued about the impact of choosing a not quality-first alternative.

Treating quality as a tool to enable migration has also increased support for the project throughout the organisation. Developers support the project because higher code quality means less tedious maintenance work, and the new test framework has significantly eases writing of tests. Analysts support the project because the functional test framework gives them enough freedom in the kind and amount of tests they write. End users support the project because they have benefited early on from new GUIs that were built as part of the project.

The impact of unresolved 499 `GOTO`s on code maintainability was assessed to be acceptable. The resulting code is seen as having much higher quality than the original system had, all thanks to $\mathcal{B}$ to $\mathcal{B}$ refactorings developed within this project. Had we **not** have improved the quality of the $\mathcal{B}$ code before translating it to C#, the quality of the resulting C# code would have been so low that it would be impossible to maintain in practice (issues reported in this paper are not the only ones, they are just used as examples). Improving code quality by removing spaghetti code, among other issues, is what made it possible for the code after translation to be maintainable. The focus on first improving the quality of the code before migrating it to another language was therefore crucial to a successful migration effort.

References

[1] D. B. Nelson, "Generalized Information Retrieval Language and System (GIRLS)," Mar 1965.

[2] B. W. Boehm, J. R. Brown, and M. Lipow, "Quantitative Evaluation of Software Quality," in *ICSE*. IEEE, 1976, pp. 592–605.

[3] W. Cunningham, "The WyCash Portfolio Management System," *ACM SIGPLAN OOPS Messenger*, vol. 4, no. 2, pp. 29–30, 1993.

[4] SonarSource, "SonarQube," https://www.sonarqube.org, 2017.

[5] A. A. Terekhov and C. Verhoef, "The Realities of Language Conversions," *IEEE Software*, vol. 17, no. 6, pp. 111–124, Nov./Dec. 2000.

[6] J. A. Bergstra, J. Heering, and P. Klint, "The Algebraic Specification Formalism ASF," in *Algebraic Specification*. ACM Press, 1989.

[7] T. R. Dean, J. R. Cordy, A. J. Malton, and K. A. Schneider, "Grammar Programming in TXL," in *SCAM*. IEEE, 2002, pp. 93–104.

[8] E. Visser, "Stratego: A Language for Program Transformation Based on Rewriting Strategies," in *RTA*, ser. LNCS, vol. 2051. Springer, 2001.

[9] P. Klint, T. van der Storm, and J. J. Vinju, "EASY Meta-programming with Rascal," in *GTTSE*, ser. LNCS, vol. 6491. Springer, 2009.

[10] E. W. Dijkstra, "Go To Statement Considered Harmful," *CACM*, vol. 11, pp. 147–148, 1968.

[11] A. Meier, "Providing Database Migration Tools A Practicioners Approach," in *VLDB*. Morgan Kaufmann, 1995, pp. 635–641.

[12] A. Ketata, C. Moreno, S. Fischmeister, J. H. Liang, and K. Czarnecki, "Performance Prediction upon Toolchain Migration in Model-Based Software," in *MoDELS*. IEEE, 2015, pp. 302–311.

[13] A. Stocco, M. Leotta, F. Ricca, and P. Tonella, "PESTO: A Tool for Migrating DOM-Based to Visual Web Tests," in *SCAM*. IEEE, 2014.

[14] A. Hora and M. T. Valente, "Apiwave: Keeping Track of API Popularity and Migration," in *ICSME*. IEEE, 2015, pp. 321–323.

[15] R. Khadka, A. Saeidi, S. Jansen, J. Hage, and G. P. Haas, "Migrating a Large Scale Legacy Application to SOA," in *WCRE*. IEEE, 2013.

[16] F. Matthes, C. Schulz, and K. Haller, "Testing & Quality Assurance in Data Migration Projects," in *ICSM*. IEEE, 2011, pp. 438–447.

[17] R. Gray, T. Bickmore, and S. Williams, "Reengineering COBOL systems to Ada," in *7th Software Technology Conference*, 1995.

[18] H. M. Sneed, "Migrating from COBOL to Java," in *ICSM*, 2010.

[19] H. M. Sneed and K. Erdös, "Migrating AS400-COBOL to Java: A Report from the Field," in *CSMR*. IEEE, 2013, pp. 231–240.

[20] H. M. Sneed, "Migrating PL/I Code to Java," in *CSMR*, 2011.

[21] V. Zaytsev, "Open Challenges in Incremental Coverage of Legacy Software Languages," in *PX/17.2*. ACM, 2017, pp. 1–6.

[22] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[23] T. Sharma and D. Spinellis, "A Survey on Software Smells," *Journal of Systems and Software*, vol. 138, pp. 158–173, 2018.

[24] H. Boomsma, B. V. Hostnet, and H.-G. Gro, "Dead Code Elimination for Web Systems Written in PHP," in *ICSM*. IEEE, 2012, pp. 511–515.

[25] N. G. Obbink, I. Malavolta, G. L. Scoccia, and P. Lago, "An Extensible Approach for Taming the Challenges of JavaScript Dead Code Elimination," in *SANER*. IEEE, 2018, pp. 391–401.

[26] F. Baader and T. Nipkow, *Term Rewriting and All That*. Cambridge University Press, 1998.