

If You Build It, Will They Come? Lessons from 20 Years of Industrial Use of a Code Transformation DSL

Johan Fabry
Raincode Labs
Brussels, Belgium
johan@raincode.com

Vadim Zaytsev
University of Twente
Enschede, Netherlands
vadim@grammarware.net

Abstract

Legacy modernisation at industrial scale requires automated source-to-source transformation rather than manual refactoring. This paper reports an experience study of RcScript, a proprietary code-transformation DSL used in production at Raincode and Raincode Labs for more than 20 years, and of the long-lived transformation platform built with it to modernise PACBASE-generated COBOL. The platform has been applied to more than 500 million lines of legacy source code. We highlight one engineering question within that industrial ASE solution: which optional language features of the implementation DSL are actually adopted in practice, which are avoided, and why. The study combines static analysis of two production configurations with an interview with the lead maintainer. The observed feature use varies, yet the results show a consistent pattern: features that improve local readability are used, whereas features that hide errors, are hard to read, or depend on weak editor support are avoided or used only sparingly. From that, we distil practical lessons for the design and evolution of industrial DSLs and long-lived automated transformation platforms.

CCS Concepts

• Software and its engineering → Compilers; Domain specific languages.

Keywords

software legacy, code transformation, domain-specific language

ACM Reference Format:

Johan Fabry and Vadim Zaytsev. 2026. If You Build It, Will They Come? Lessons from 20 Years of Industrial Use of a Code Transformation DSL. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26)*, October 12–16, 2026, Munich, Germany. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3832783.3834520>

1 Introduction

Industrial legacy modernisation often involves repetitive, large-scale source code transformations that are infeasible to perform manually with acceptable cost, risk, or consistency. In such settings, automated software engineering solutions are needed to recognise

recurring code patterns, rewrite them systematically, and do so repeatedly across many customer codebases.

This paper therefore focuses on a practical question inside one such production transformation platform: given a transformation DSL that implements a long-lived industrial automation pipeline, which optional language features are actually used, which are avoided, and why? Rather than arguing for universally good or bad language constructs, we report what happened in one automation-heavy industrial setting and extract lessons that may matter when evolving similar DSLs and automated transformation systems.

Our case is RcScript, a proprietary code-transformation DSL used at Raincode and Raincode Labs in the scope of software re-platforming — more specifically, providing off the shelf as well as bespoke replacement compilers to migrate mixed 2GL/3GL/4GL software assets from the mainframe to the .NET ecosystem and make them cloud native. We analyse a production source-to-source transformation system that has been in active daily use since 2002. It is maintained by developers with long-term RcScript experience, and has been applied to transform more than 500 MLOC of legacy software. The study combines static analysis of two near-identical production configurations with an interview with the lead maintainer of both the system and the RcScript interpreters.

The value of this showcase lies more in its depth than in its breadth. We do not claim that RcScript is representative of programming languages in general, and the transformation codebase is modest in size compared with massive legacy systems it processes. Instead, its distinctive property is the combination of more than two decades of continuous production maintenance with repeated use on industrial migration projects at very large source-code scale. This gives us an unusually long view of which optional constructs remained part of working practice, which were used only selectively, and which saw almost no uptake. The transformed MLOC therefore establish the production exposure of the automation platform; they should not be interpreted as the size of the RcScript corpus analysed here.

The contributions of this paper are:

- an industrial experience report on a long-lived automated source-to-source transformation solution for legacy modernisation;
- an analysis of how the DSL used to implement that solution is actually used in production, including which features are used frequently, sparingly, or almost never;
- interview-based explanations for these usage patterns, centred on readability, error visibility, and tool support;
- practical lessons for teams engineering proprietary DSLs and transformation-based ASE solutions in industrial settings.

ASE '26, Munich, Germany

© 2026 Copyright held by the owner/author(s).

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26)*, October 12–16, 2026, Munich, Germany, <https://doi.org/10.1145/3832783.3834520>.

The goal is analytic rather than statistical generalisation. We do not claim that one proprietary codebase can settle questions of language design in general; nor is the purpose of this paper to argue for new general-purpose programming-language constructs. Instead, we offer a detailed industrial case of how the design of the DSL used inside a production ASE pipeline affected the long-term engineering of that automation platform, from which similar organisations may judge what is likely to transfer to their own settings.

The rest of the paper is structured as follows. Next we discuss related work, and then follow with an overview of the RcScript language. Section 4 then describes the different language features we investigate in the software under study, explained in section 5 together with the analysed corpus. The next two sections perform and report on a quantitative and qualitative analysis, respectively. Section 8 distils lessons learned, followed by threats to validity in section 9 and our conclusion in section 10.

2 Related Work

The oldest paper on software language features was written by Zeiger [59]: four language features (mapreduce, recursion, parameters and declarations) were investigated and formalised with automata and mathematics. Such foundational work is still relevant, exemplified by Staton and Levy’s formalisation of type constructions of imperative or impure programming languages [48].

Modern programming languages support many different concepts and constructs, ranging from basic ones like having types and variables, to more advanced ones like polymorphism, exceptions and concurrency [55]. Modelling languages cover other constructs, atypical to primarily algorithmic languages, like multiplicity or explicit associations [15]. Markup languages or exchange formats include constructs for linking to external entities or detailing layout of imported images. Software languages in general cover all of these kinds of languages [35], as well as anticipate many more yet to be created domain-specific languages [51]. Thus, the potential arsenal for a software language designer is already huge, and there is always an option of creating entirely new constructs.

Some language features are traditionally considered harmful or dangerous and therefore naturally attract researchers. Callaú et al. investigated how developers use dynamic language features of Smalltalk [12], similar work exists for JavaScript [45], PHP [24, 25] and Python [3, 18, 53]. Some of these projects have continuations aimed at using the results of their analyses as refactoring opportunities [38]. Breuel coupled a family of dynamic language features to an implementation strategy for them [9]. It is speculated that languages with a known abundance of harmful features, are stigmatised up to the point of causing mental harm to their users [30].

Prior empirical studies have examined language-feature use with substantially broader datasets and, in some cases, stronger behavioural or evolution-oriented measurements. Callaú et al. [12] combine large-scale analysis of Smalltalk projects with qualitative inspection of dynamic-feature uses; Richards et al. [45] instrument JavaScript execution across a large population of websites; Åkerblom et al. [3] trace uses of dynamic features in Python programs; Hills [25] and Farooq et al. [18] study feature-usage patterns across multiple systems in PHP and Python, respectively; and Wang

et al. [53] relate Python dynamic-feature usage to change-proneness. These designs favour breadth, runtime evidence, or associations with software-evolution outcomes. Our study instead sacrifices such breadth for longitudinal and organisational depth: it examines one proprietary transformation platform with more than two decades of production history, observes it in its production context, and combines static occurrence counts with the retrospective interpretation of its long-term maintainer. Consequently, our counts support descriptive within-case observations, while claims about preference, causality, or maintenance impact require the additional evidence available in some of these prior designs but absent here. Our resulting evidence is, however, substantially richer in the industrial and historical context surrounding the observed uses. This difference in study design motivates the inference boundaries that will be stated explicitly in section 6.1.

Beyond work organised around particular language features, our study also relates to industrial experience reports on language engineering and model-driven automation. Mohagheghi et al. [40] synthesise experiences from three industrial applications of model-driven engineering, highlighting practical concerns such as reuse, tool integration, and usability. Hubaux et al. [28] follow four different companies in their adoption endeavours of the same DSL for feature modelling. Völter et al. [52] report lessons from developing the mbeddr family of embedded-software languages with JetBrains MPS, focusing on language modularity, management of complexity, scalability, and the development process. Hermans et al. [22] look into about 30 projects of using DSLs in industry with the goal of identifying conditions for successful DSL use. More recently, Bunte et al. [11] evaluate the Spoofox language workbench through OIL, an industrial DSL at Canon Production Printing, comparing its Spoofox implementation with an existing Python implementation and deriving lessons for industrial language engineering.

Those studies primarily evaluate the adoption, engineering, deployment, or (co)evolution of a modelling or language-engineering technology in an industrial setting. Our perspective is complementary: both RcScript and the transformation platform built with it were already mature technologies when the observations reported here were made. We therefore look inside an established industrial automation platform and retrospectively examine its implementation language itself: which optional features became part of routine production code, which remained exceptional, and how the long-term maintainer explains those usage patterns. In this sense, the contribution is less an evaluation of whether an organisation should adopt a DSL or language workbench, and more an account of how one proprietary language has aged under sustained industrial use.

Sadowski et al. try to apply methods of usability testing to comparing software languages for parallel programming based on their features [46]. Lamm attempts to separate software language features like type systems, object orientation, genericness and reflection to see how they respect or overcome abstraction limitations [34]. Lämmel et al. proposes a general method to compare software languages by their features [36].

Caromel et al. demonstrate the classic approach to researching language features: state the problem, think hard, present a list of language features that solve this problem, conclude that since they are all present in Eiffel, everyone should be using it [13]. An even older example by Banâtre et al.: “actions are a must-have language

feature to model/program communicating processes” [5]. Same holds for Andrews et al. [4]. Similarly, Krishnamurthy et al. make a wish list of language features motivated by the contemporary problems *they* are facing [33].

The reverse of that approach is exemplified by Heymans et al. [23] who perform an exhaustive contemporary investigation of all feature diagram languages¹, in order to draw conclusions about their commonalities. In particular, they retarget an older result around UML to evaluate “language appropriateness” and thus quality of a domain-specific language, including:

- domain appropriateness (the language is powerful enough to express domain concepts);
- participant language knowledge appropriateness (statements of the language as used by the developers, are aligned with their knowledge);
- knowledge externalisability appropriateness (all explicit knowledge of domain experts can be expressed in the language);
- comprehensibility appropriateness (language users understand the language);
- technical actor interpretation appropriateness (the language is automatically executable and analysable);
- organisational appropriateness (the language is aligned with standards and other contexts of its use).

Language features that we investigate, were intentionally put into the language by its designers. Even though it is well-known that “language design is largely an art, not a science” [41, p.67], there are many books containing advice and methodologies on how to design general-purpose, domain-specific and modelling software languages [2, 21, 27, 37, 42], as well as papers [15, 17, 19, 26, 39, 47, 51, 56–58].

Since RcScript is a software language to program code transformations, it falls into the same category as other metaprogramming languages like ASF [6], FermaT [54], Rascal [32], SmPL [31], Stratego [50] and TXL [16]. However, some of these are strongly typed, and most are functional. RcScript features will be explained in the next section.

3 The RcScript Language

The RcScript language is a metaprogramming language built at Raincode over 20 years ago. It is extensively used in almost all its products. Notably, multiple Raincode products perform source-to-source code transformation, either keeping source and target language the same, or transforming to a different language, and these transformations are implemented completely in RcScript.

RcScript is an interpreted, dynamically typed, procedural language designed with the explicit purpose to perform metaprogramming tasks. Programs are defined in files, and each file declares top-level procedures and variables. Programs can be considered as modules as well: procedures and variables are read-only public: a script can read another script’s variables and call its procedures. There are various versions of the interpreter, each version is bundled with a parser for a different target language, i.e., allowing to

write a metaprogram that processes programs in that target language. The interpreter takes as argument the list of target files to parse and the RcScript script to run. The file(s) are parsed before the script is run, and their Abstract Syntax Trees (ASTs) are contained in the ROOT or ROOTS implicit variables.

The parser and data structures for the AST are implemented in different languages (YAFL [7] and DURA [8]), which are outside of the scope of this paper. However, they do impact RcScript: while it is a procedural language, RcScript programs see the ASTs as objects of a class-based, single-inheritance style language. (Nonetheless, since the RcScript language itself is procedural, there are no means in RcScript programs to declare classes, methods or inheritance.) For instance, all nonterminals in the AST inherit from `NonTerminal`, and all expressions typically inherit of `Expression`, which in its turn would be a (grand)child of `NonTerminal`. This allows each nonterminal to know its begin line and column number as well as its end line and column number because it is defined in the `NonTerminal` class. In RcScript, the methods and attributes of these objects are accessed using the typical dot notation.

In RcScript the underlying source code of the programs in the target language cannot be modified. Instead, modifications to the AST are performed by means of calls to the so-called patch API. The interpreter exposes an API that allows modifications to the program to be requested by the script. The ASTs are immutable and are never manipulated directly. Instead patches are created: a patch can be a string to be inserted at specific line and column number in the text, it can be a nonterminal to be deleted, or it can be a nonterminal to be replaced (by another nonterminal or by a string of text). All patches are queued, applying them causes a new file to be written to disk that incorporates all these changes.

The fact that arguably typical metaprogramming activities are performed through APIs, raises the question of what we should consider as a language feature to be studied in this project. Firstly, if the interpreter provides some API to obtain e.g., the line number of a nonterminal, this is arguably a feature of the standard library instead of a language feature. Secondly, we also need to consider whether a specific language feature may be re-implemented by the programmer in an ad hoc manner. For example, suppose that modifications of the underlying source code were indeed directly possible in RcScript. Then a programmer could create an alternative target language manipulation infrastructure and use it instead of the existing patch API. But such an infrastructure would be so large that programmers would be hard-pressed to re-implement it. Hence this is not a realistic scenario. In effect, the programmers have no choice: they are forced to use the feature as provided.

Hence the language features that we consider in this paper:

- **are not** in the standard built-in library;
- **could be** implemented by a programmer with little code.

4 Studied RcScript Features

Having given an overview of RcScript, we now focus on the language features we study in this text. By a language feature we see any language-rooted functionality typically listed in books on programming language concepts, and by nonstandard ones we mean ones outside the intersection of mainstream languages (this eliminates looping constructs, OO basics, etc). Arguably, nonstandard

¹The possible confusion is not intentional: they investigate *feature diagram* languages while we talk about *language features*. The two have nothing in common, except that it is possible to represent a set of language features in any feature diagram language.

language features represent language complexity on some level because they are the ones that need to be learnt by programmers experienced in other languages, when learning a new one. Thus, we study features that are nonstandard for a procedural (dynamically typed) scripting language, yet included in RcScript. As noted previously, these furthermore could be re-implemented in a small amount of code and hence we will describe them showing their alternative implementation.

4.1 The Safe Navigation Operator !

The ! operator is a variant of the dot (.) operator that handles the empty VOID value on the left hand side of the dot, as well as the presence or absence of a field or method on the right hand side. For example, take the expression `nt!SubNodes`. If the left hand side `nt` is VOID, the entire expression will evaluate to VOID. If `nt` does **not** define a property `SubNodes`, the result is VOID. If it does, for example when `nt` is a nonterminal, the result is the value of `nt.SubNodes`, in the example the collection of all subnodes of `nt`.

In general, an expression `l!r` can be implemented in RcScript using the following code:

```
TEST 1 <> VOID
  THEN (TEST 1 CAN r THEN l.r ELSE VOID)
  ELSE VOID;
```

The `TEST a THEN b ELSE c` is the equivalent of the CPL-style ternary operator expression `a ? b : c`. The `CAN` operator is a reflexive operator that returns TRUE if the left hand side defines the property or method given as the right hand side.

Note that because ! with a VOID left hand side is VOID, an expression like `l!r!s` will evaluate to VOID if either `l`, or `l!r`, or `l!r!s` is VOID.

4.2 List Operations

There are two native data structures in RcScript: lists and dictionaries. Dictionaries do not have any standout features so we shall not discuss them in this text. Lists have the arguably well-known `map()` and `filter()` operations as well as membership testing expressions `IN` and `NOT IN`. Given their prevalence, we do not study these features in this paper.

The interesting language features are the five quantifiers over lists: `COUNT`, `FIRST`, `LAST`, `FOR_ALL`, `THERE_IS`. These take three arguments: a list, a parameter name and a filter expression where that parameter will be bound. Conceptually, the expression filters the list to produce a result list. The quantifiers then return, respectively, the number of items in the result list, the first element, the last element, TRUE if the result is as large as the original (FALSE if not), TRUE if the result is not empty (FALSE if not).

For example, take the following `THERE_IS` that establishes whether in the list of literals `{1, 2, 3}` there is a value larger than 2:

```
THERE_IS X IN {1, 2, 3} :- X > 2;
```

We can implement this as shown below, using the list count operation `#` and the filter operation `|`. The latter takes a list as the left hand side and a lambda function definition as the right hand side, with a parameter `[X]` below) and a filter expression.

```
#( { 1, 2, 3 } | [X] X > 2; ) > 0
```

4.3 The X Implicit Variable

RcScript has both a `FOR` and a `WHILE` loop construct. What is notable about them is that naming the iterator is optional: skipping the definition of the iterator will cause it to be named `X`. Put differently, instead of writing all line numbers of the subnodes of `nt` to the standard output as follows:

```
FOR X IN nt!SubNodes DO
  OUT.WriteLine(X.LineNr);
END;
```

an alternative is (note the missing `X`):

```
FOR IN nt!SubNodes DO
  OUT.WriteLine(X.LineNr);
END;
```

This implicit variable can also be used in the list quantifiers and operations we presented in section 4.2. Similar to the code above, the definition of the parameter (`X` or `[X]`) is simply omitted.

4.4 Reading from RESULT

`RESULT` is a second implicit variable of note. It arises from the fact that RcScript does not have a return statement. Instead, all procedures implicitly declare the variable `RESULT`, initialised to VOID. All procedures return the value of `RESULT` when they end.

Hence, to return a result, each procedure's code assigns the return value to `RESULT`. However, since it is a variable, it is also possible to read `RESULT`. This allows the procedural code to immediately accumulate variables in `RESULT`, instead of needing to define an extra variable for the accumulation.

For example, the code below finds the highest end column number of the subnodes of `nt`:

```
PROCEDURE HighestEnd(nt)
VAR temp;
BEGIN
  temp := 0;
  FOR IN nt!SubNodes DO
    temp := TEST temp > X.EndColNr
      THEN temp
      ELSE X.EndColNr;
  END;
  RESULT := temp;
END;
```

It defines a procedure `HighestEnd` that has one variable `temp`, iterates over all subnodes, keeping the highest end column number in `temp`. It can also be defined without the use of this variable:

```
PROCEDURE HighestEnd(nt)
BEGIN
  RESULT := 0;
  FOR IN nt!SubNodes DO
    RESULT := TEST RESULT > X.EndColNr
      THEN RESULT
      ELSE X.EndColNr;
  END;
END;
```

4.5 Non-intuitive Features

There are two keywords in RcScript that provide features whose effects are not immediately intuitive.

Firstly, `IMPLIES` is the following Boolean operator:

`a IMPLIES B  $\iff$  (NOT A) OR B`

Secondly, `|||` computes the transitive closure of a function over a set. It takes a start set, a parameter declaration and an expression. The expression is computed for all elements of the start set and its results are accumulated in a result set. The function is then computed for all elements in the result set, and its results are accumulated in the same result set. This is repeated as long as the result set grows. The RcScript code for this is somewhat extensive, so we do not include it here.

4.6 String Handling

It is to be expected that a metaprogramming language provides features that facilitate the handling of strings. This is especially so if producing or modifying code is performed using strings and not through the direct construction of AST nodes, as is in RcScript.

There are only two string handling operations in the language: string concatenation with `&` and string templates with `{`}``. Templates allow the construction of (multi-line) strings where variables of the program are referred, and when the string is constructed the variables are replaced by their current value (as a string).

In other words, if `nt` is a nonterminal, a string like the following:

```
"The nonterminal " & nt & ":
- starts at line " & nt.LineNr & "
- ends at line " & nt.EndLineNr
```

can also be written using a template expression with three template lines:

```
{
`The nonterminal ` nt `:
` - starts at line ` nt.LineNr
` - ends at line ` nt.EndLineNr
}
```

5 Industrial Transformation System

For the purpose of analysing the use of the language features we described above, we studied one specific solution of Raincode: the migration of PACBASE software. PACBASE is a fourth-generation language whose compiler essentially consists of a generator of COBOL code. This means that compiling PACBASE code to executable code is fundamentally a two-step operation: first the COBOL code is generated and second this code is compiled using the systems' COBOL compiler.

The migration path offered by Raincode allows a client to remove the dependency on the PACBASE compiler and infrastructure. This is done by taking the COBOL code, as generated by the compiler, and applying a wide variety of refactorings to it to make it readable and maintainable. This is needed because, as a byproduct of being generated code, the COBOL code is highly verbose (even for COBOL), sub-optimal in terms of using expressive language features, difficult to read, and in general unfit for human consumption. The refactoring solution by Raincode transforms the COBOL code

to a palatable form that can be further developed and maintained by the client's development team. It is both the most popular and the most thoroughly tested solution on the market for this niche [43].

5.1 Automated Software Engineering Context

The industrial problem addressed in this setting is the large-scale modernisation of legacy PACBASE systems. Because PACBASE generates COBOL that is correct but difficult to read, maintain, and evolve, a migration project must systematically recognise patterns recurring in generated code, and rewrite them into forms that are more maintainable for downstream developers. At the scale of hundreds of millions of lines of code, this work is not realistically achievable by manual inspection and editing alone.

The applied automated software engineering solution is a configurable source-to-source transformation pipeline implemented in RcScript. Individual transformation steps detect and rewrite specific code patterns, while an orchestration layer manages dependencies, ordering constraints, iteration, and customer-specific configuration. In other words, the migration capability is not a one-off script but a long-lived automation platform used in production.

This paper focuses on the maintainability of that automation platform itself. The transformation pipeline is implemented in RcScript, so the language features that developers actually adopt, avoid, or work around, directly affect the evolvability, readability, and reliability of the industrial automated solution. For an automation platform that is maintained over decades and customised per customer, feature choices in the implementation DSL therefore have operational consequences: they influence how easily transformation rules can be extended, reviewed, debugged, and safely adapted during migration projects.

5.2 Solution Details

The PACBASE migration is implemented as a sequence of many steps. Each step is responsible for a single refactoring: finding pieces of code that can be refactored and then applying the refactoring where possible. To illustrate typical code for a (tiny) refactoring step, an example is shown in [fig. 1](#). This step takes generated code that has a spurious IF test and reduces it to its essential MOVE operation², as shown in its initial comment (lines 1–4).

The code in [fig. 1](#) iterates over all statement nonterminals `nt` (line 9) in the AST and first establishes if its structure corresponds to the kind of IF that needs to be refactored (lines 10 to 21). When this is the case, a patch (see [section 3](#)) is made that deletes the characters from the beginning of the IF up to the end of the ELSE keyword (line 23). Afterwards, the `ENDIF` keyword is removed (line 24), and the source code change is applied (line 25). The code uses an external module called `RefoLib` (lines 9, 23–25) that implements utility methods that are used throughout the PACBASE refactoring code. Note also the use of `!` (discussed in [section 4.1](#)) in lines 10 and 11.

With each step performing one specific refactoring where possible, a crucial part of the migration of PACBASE code lies in how these steps are orchestrated. Due to a step being capable of producing code that matches in another step, such producer-consumer relations may actually be (transitively) circular. Some steps even

²A MOVE in COBOL is actually a copy operation.

```

1 COMMENT
2 match:                                     -> produce:
3 IF x = y THEN MOVE y TO A ELSE MOVE x TO A END -> MOVE x TO A
4 ENDCOM
5
6 PROCEDURE Run;
7 VAR thenmove, elsemove, cond;
8 BEGIN
9   FOR nt IN RefoLib.GetStatements(ROOT) DO
10    IF nt IS IfStatement AND #(nt.ElseStatementList!List)=1
11    AND #(nt.StatementList!List)=1 AND nt.Condition IS AtomicComparison
12    AND nt.Condition.Right IS SingleRightSide AND nt.Condition.Right.Op=Equal
13    THEN
14      thenmove := nt.StatementList.List[1];
15      elsemove := nt.ElseStatementList.List[1];
16      cond := nt.Condition;
17      IF thenmove IS MoveStatement
18      AND elsemove IS MoveStatement AND thenmove.Targets == elsemove.Targets
19      AND (thenmove.Expr == cond.Left AND elsemove.Expr == cond.Right.Arg
20      OR elsemove.Expr == cond.Left AND thenmove.Expr == cond.Right.Arg)
21      THEN
22        PATCH.Delete( nt, RefoLib.GetElseKeyword(nt) );
23        RefoLib.RemoveEndIf( nt, FALSE );
24        RefoLib.CommitAtomicPatch("IfMoveToMove", nt);
25      END;
26    END;
27  END;
28 END;
29 END;

```

Figure 1: The IfMoveToMove refactoring step. Lines 1 to 4 document the intuition of the refactoring.

produce “dirty” code intentionally, assuming that a subsequent step will apply a cleanup refactoring. Orchestration is the responsibility of a workflow-like infrastructure that decides which refactoring steps are applied (by calling their Run procedure) in what order, as well as when and how to loop.

Migration of PACBASE code can also be configured per client: out of the library of steps, not all of them need to be applied and some have parameters (to their Run procedure) that can be fine-tuned. Raincode provides a “default” configuration of steps and parameters that is used in a first proof of concept migration of a small subset of client code. After discussing the results of this proof of concept, the configuration is then subsequently customised together with the client. The purpose for this customisation step is to have the transformations yield COBOL code that is better suited to the customers’ wishes.

The code for a given migration hence corresponds to the collection of selected steps that will be run, the orchestration code and all utility code like the RefoLib module used in [fig. 1](#). For this study, we have investigated two such instances: the default configuration and the configuration for a project with Bankia. The Bankia project [44] was a one-year effort that modernised over 250 million LOC

of COBOL, the largest such project yet undertaken at Raincode (possibly, anywhere). While this configuration is representative of a customisation that is typically performed by the client it however cannot be representative of all configurations. To act as a proxy for the “median” configuration we hence also investigated the default configuration. We can consider it as to give an estimate of the code that has been run in all previous PACBASE projects, the combined LOC of which is also around 250 million.

This software is in use since 2002 and around ten persons have made significant contributions to it. However, the large majority is developed by one person, who now is fully responsible for maintaining this codebase. He has experience with RcScript since 2001 and has been the factual maintainer of the RcScript interpreters since 2003.

To form the corpus of code to analyse, we have manually inspected the Bankia configuration as well as the default configuration and determined which steps are executed in each. For each of these configurations we have then determined which modules are called by the different steps, again through manual code inspection. Adding the orchestration code to each configuration then completes the corpus. An overview of basic metrics for each configuration are

given in [table 1](#). The numbers of this table already give an indication that the code for both configurations is not substantially different, which also will become apparent in the next section.

Table 1: Basic metrics for the corpus being analysed. “Configuration-specific steps” counts only refactoring steps present in one configuration and absent from the other; it is not the total number of refactoring steps executed in a migration. The configurations overlap by 22173 LOC (97–98%), so they share nearly all analysed code.

Amount of	In Bankia	Default
Files	76	79
Configuration-specific steps	3	6
K on disk	860	870
Procedures	700	703
LOC	22605	22875

This means that the 3 and 6 configuration-specific steps in [table 1](#) do *not* mean that the Bankia migration or the default configuration use only three or six refactorings. They identify only the small set of steps that separates one analysed configuration from the other. The large common core is more important for the study design than this minor variability: comparing the two configurations can reveal whether this small configuration delta substantially changes the descriptive feature counts, but it is not meant to demonstrate that the same patterns replicate in an independent project or codebase.

6 Feature Usage in the Industrial Codebase

Our quantitative question concerns source-level *feature use* rather than *runtime performance*. We are not interested in how often the *interpreter* executes particular constructs, but in how often those constructs occur in the transformation source code written and maintained by *developers*. To establish this, we statically analysed the source code for the two different production configurations and counted the extent of use of the language features. Naturally, we have written a program in RcScript that analyses the corpus written in RcScript. This is made possible because there is an interpreter that allows us to metaprogram RcScript code (i.e., use RcScript as both the target and the metaprogramming language).

The results differ only slightly between the Bankia corpus and the default corpus. As seen in [table 1](#), the configurations overlap by 97–98% of their LOC and are therefore just not independent samples. Note that we use the second configuration only as a sensitivity check: it shows that the small configuration-specific delta does not lead to significant changes in the descriptive usage counts. This comparison does not provide evidence of cross-project robustness or independent replication.

6.1 Measurement and Inference Boundaries

Our static analysis measures syntactic occurrences in the maintained RcScript source code. We use these counts as evidence that a feature is present in, and to varying degrees incorporated into, this industrial codebase. While we might sometimes be tempted to see them as proxies for something else, occurrence counts alone do *not* measure feature value, developer preference, runtime importance,

or causal effects on defects, productivity, or maintainability. Where the set of opportunities to use a feature can be identified from the source, an opportunity-normalised interpretation is stronger; where it cannot, we report the counts descriptively.

The implicit X (cf. [section 4.3](#)) is the clearest example of the former case. The corpus contains roughly 660 constructs in which an iterator *could* be left implicit, and it *is* made explicit only eight times. Here, the denominator is observable and the counts directly characterise uptake at eligible sites. For other features the choice set is not observable. In particular, the counts of `.` and `!` (cf. [section 4.1](#)) must not be read as a preference ratio: many dot accesses occur where safe navigation is truly unnecessary, while `!` is relevant when absence or VOID must be tolerated. Likewise, the RESULT (cf. [section 4.4](#)) counts establish that reading the implicit result variable is a recurring idiom, but they do not tell us how many procedures could sensibly have used an explicit accumulator instead.

The interview in [section 7](#) supplies practitioner interpretations of these source-level observations. We use those explanations to make sense of the counts and to formulate case-grounded lessons, not as independent evidence of causality. The conclusions in [section 8](#) therefore distinguish observed usage from the maintainer’s explanation of why that usage arose.

6.2 The `!` Operation

There are two interesting questions about the use of the `!` operation. Firstly, does it completely replace the `.` operation? Secondly, in expressions that are qualified names, how frequent do they use multiple `!` chained together?

In [table 2](#) we can see that there are over 9000 dot operations in the corpus and over 400 uses of the exclamation mark. This means that the exclamation mark is indeed well used, around 1.6 times per procedure, or 5.6 times per file, but by no means does it replace the dot operation.

Table 2: Number of consecutive dot or exclamations in one expression, e.g., `a!b!c` corresponds to 2 times `!`

.	In Bankia	Default	!	In Bankia	Default
1	7499	7622	1	344	350
2	1397	1405	2	30	30
3	98	105	3	10	11
4	16	21	4	1	1
5	1	1	5	0	0

The semantics of `!` allows for a compact manner to follow a chain of references, while allowing any of the constituent parts to be VOID. As noted in [section 4.1](#), `!` applied to VOID is VOID. Hence an expression with `!` can be written as a qualified name, e.g., `p!q!r!s`. This allows for the compact navigation of a path to a specific property while being resilient with regard to absent parts of the path. We can see that this quite often used, with more than 40 multi-exclamation mark expressions in the code, i.e., once every two source code files.

When looking for qualified names with the dot, we also find them in the corpus. There is however no straightforward relationship between the uses of `.` and `!`. For a single use the raw occurrence ratio is about 21 to 1, double use is 47 to 1, triple use is 10 to 1 and

quadruple use is 20 to 1. These ratios show that both forms occur in qualified names, but, as discussed above, they do not quantify a developer preference because the two operators are not semantically interchangeable at every site.

Lastly, it is of course also possible to mix the use of dots and exclamation marks to denote a qualified name. We see that this is done in the code, in the Bankia corpus this is done 155 times, and in the default corpus it is present 165 times.

6.3 Implicit Variables

There are two implicit variables in RcScript that are of interest to us in this work: the `X` in loops and list comprehensions, and the `RESULT` variable used to set the return value of a procedure. We discuss them here in this order.

Table 3: Implicit variables `X` and `RESULT`

	In Bankia	Default
<code>X</code> is implicit	652	658
Implicit <code>not</code> used	8	8
<code>X</code> is read	1517	1521
Procedure without <code>RESULT</code>	190	191
Procedure assigns to <code>RESULT</code>	510	512
Procedure reads <code>RESULT</code>	153	152
<code>RESULT</code> is assigned	1037	1040
<code>RESULT</code> is read	452	450

Recall that loops can omit to declare the variable for the iterator in the loop, and list quantifiers can omit to declare the argument to the filter function (see [section 4.3](#)). The first question hence is: in how many of these occasions the definition of this variable is effectively omitted? [Table 3](#) shows that of around 660 possible implicit `X`s, the vast majority are indeed implicit: such a variable is only declared 8 times.

Other variable names are: `df`, `t`, `Y`, `tok`, `regexp`, `st`, `sub`. The case of the `Y` variable is telling however. It shows a situation where an implicit `X` cannot be used since there is a quantifier nested inside a filter function and both the filter as the quantifiers' arguments are needed:

```
fromparag!ThruBlock2 |
  NOT (THERE_IS [Y] IN parag!ThruBlock2
    :- X == Y);
```

Inspection of the code reveals that of the 8 cases where such a variable is declared, 5 are cases where they are indeed needed. Only 3 cases remain (`tok`, `regexp`, `st`) where the implicit definition of `X` is not used.

Looking at the codebase we also see that there is **no** definition of `X` as a (local or global) variable. There are however many single-character variable names: over half the letters of the alphabet are used and the biggest culprit is `a`, declared 16 times. There is however **no** uppercase single-character variable name and no lowercase `x` (although `xx` is present). We verified, and the language does indeed permit defining uppercase single-character variable names (including `X`), yet it is clear that in this code the role for `X` is reserved as the implicit variable for loops and list quantifiers.

Looking at the numbers for `RESULT` (also in [table 3](#)), of the approximately 510 procedures that assign it (and hence return a value), 29% also read it (around 150) and the average number of reads is around 3. Assignment to `RESULT` is more than double the amount of reads.

6.4 List Operations

Looking at the typical map and filter operations ([table 4](#)), we see that they are used in average about twice per file, with map being used a bit more than filter. In total, membership tests are performed about one hundred times, so 0.8 times per file.

Table 4: List Operations

Feature	In Bankia	Default
Filter	159	160
Map	189	189
IN	83	80
NOT IN	23	21
FOR_ALL	100	105
THERE_IS	107	108
FIRST	69	69
LAST	17	17
COUNT	22	22

Considering the list quantifiers, in [table 4](#) we can see that `FOR_ALL` and `THERE_IS` are the most popular quantifiers, comparing to map and filter, they are used more than half the time. Looking at their absolute numbers, we can say that they are used quite often, on average 1.3 times per file or once every 7 procedures. `FIRST` is third in the list, on average 1.1 times per file or once every 10 procedures. `COUNT` and `LAST` come last, about once every 4 files or 35 procedures.

6.5 Miscellaneous

[Table 5](#) shows that string-template use is highly concentrated rather than evenly distributed. Across each configuration we find 64 templates comprising 1000 template lines, but there is one program that generates a large `Makefile` and accounts for 34 of those templates and 891 of the lines. Excluding this specialised program leaves 30 templates and 109 template lines, or 3.6 lines per template, compared with 15.6 lines per template in the aggregate. We therefore treat the aggregate template-line average as an outlier-sensitive descriptive statistic, not as a typical template size.

Considering the deep thought features (presented in [section 4.5](#)), [table 5](#) reveals that their usage is quite low. The `IMPLIES` feature is used once every 4 files, on par with `LAST` and `COUNT` detailed above. The transitive closure is only used once in the entire codebase.

7 Interview-Based Lessons from the Lead Maintainer

Because RcScript is used only within Raincode and Raincode Labs, a broad user study was infeasible. We therefore conducted a focused semi-structured interview [14] with the lead maintainer of the PACBASE migration software, who has worked with RcScript since 2001 and maintained the RcScript interpreters since 2003. During

Table 5: Miscellaneous language-feature counts. The outlier-excluded rows make explicit how strongly one specialised Makefile generator dominates the string-template line count.

Feature	In Bankia	Default
String templates (all)	64	64
Template lines (all)	1000	1000
Templates without the outlier	30	30
Template lines without the outlier	109	109
IMPLIES	18	18
Transitive closure	1	1

the interview, the participant was shown the quantitative tables and asked to interpret the observed usage patterns.

Participant positionality and role in the evidence. The interviewee is not an independent observer of the analysed corpus: he wrote the large majority of the transformation code and has also maintained the RcScript interpreters since 2003. This dual role is valuable because it gives him unusually deep knowledge of both the historical reasons behind the code and the language mechanisms used to express it. At the same time, his explanations are necessarily interpretations of a codebase to which he is a major contributor, and may reflect retrospective rationalisation or his own long-standing development style. We therefore do not treat the interview as independent triangulation of the static analysis. Instead, we use it as expert insider interpretation: the counts establish what is present in the source, while the interview supplies attributed explanations for how and why those patterns arose. This distinction is carried through explicitly in [table 6](#) and [section 9](#).

The aim of the interview was explanatory rather than representative. Instead of reproducing the transcript (which is, after removal of proprietary information, available on request), we distil the recurring themes that help make sense of the industrial usage data.

7.1 Error Visibility Matters More Than Convenience

The lead maintainer regarded the `!` operator as useful but risky. In his view, it can keep execution going after typos or mistaken member accesses and thereby hide defects for longer time that ordinary dot access would reveal earlier. This helps explain why `!` is used regularly but does not replace ordinary dot access. Mixed chains of `.` and `!` were described as an incremental maintenance pattern: code is written first with dots and later hardened only where VOID may occur.

7.2 Brevity is Accepted Only While it Remains Readable

The implicit `X` is strongly preferred in short local contexts and is almost universally used where available, but the interviewee explicitly rejected it in long or deeply nested expressions. The same readability rationale explains the positive view of `FOR_ALL`, `THERE_IS`, and `FIRST`: they were valued less for expressive power than for making common list queries easier to read and write.

7.3 Tool Support Shapes Feature Adoption

String templates were not rejected on semantic grounds. Their limited use was attributed mainly to poor editor support: without syntax highlighting, embedded variables were difficult to distinguish from surrounding text. Once editor support improved, the feature became more acceptable. In other words, tooling was part of the feature’s effective usability in practice.

7.4 Some Expressive Features Remain “Demo” Features

The deep-thought features were described as hard to read, overly specialised, or both. In particular, the sole remembered use of the transitive-closure operator was later optimised away. For this industrial setting, its expressive power did not translate into regular day-to-day value.

7.5 Representativeness Within the Company

The interviewee considered himself more knowledgeable than a typical RcScript developer (Raincode and Raincode Labs currently have about 20 in total), but judged the kind of code in this project to be broadly similar to other large RcScript codebases inside the company. We therefore use the interview to explain the observed behaviour in this codebase, not to claim population-level representativeness.

8 Lessons Learned for Long-Lived Industrial ASE Platforms

The quantitative counts and the maintainer interview jointly motivate five case-grounded propositions for long-lived industrial ASE platforms. We retain the “lesson” terminology because this is an industry showcase, but the propositions are deliberately not presented as equally established or universally causal. Their evidential status follows [table 6](#): some rest on directly observable source-level opportunities, while others are interpretive explanations or our hypotheses prompted by sparse observations.

[Table 6](#) makes it explicit that the extent to which the five lessons below are supported by our experience, is not equally strong across them. They are analytic lessons from this industrial case: strongest where an opportunity denominator is observable, and more tentative where the evidence consists of rare occurrences plus retrospective expert interpretation.

L1: Available features show markedly different levels of use (descriptive support). Within this RcScript codebase, constructs that are all available to the same industrial development setting range from common (`FOR_ALL`, `THERE_IS`, implicit `X`) to marginal (`IMPLIES`, `|||`). This is direct evidence that mere feature availability did not translate into comparable source-level use in this case. For language evolution in similar long-lived platforms, the practical implication is to examine observed production use rather than infer value from the elegance or mere presence of a construct. This particular aspect of the relation between features made available in a (new release of a) language and the timeline of its adoption by the developers, is also the focus on research on *software modernity* [1].

Table 6: From source-level evidence to case-grounded claims. In this table we separate what is directly observed from the maintainer’s interpretation and from the limits of the resulting inference.

Feature(s)	Observed evidence	Case-grounded interpretation	Evidence boundary
Implicit X	About 660 eligible constructs per configuration; only 8 use an explicit iterator, of which 5 were judged necessary on inspection.	Strong within-case evidence that the implicit iterator is the habitual form at eligible sites.	An opportunity denominator is observable, but the result still concerns this codebase and its dominant development style.
! navigation	More than 400 occurrences, alongside over 9000 dot operations and mixed ./! chains.	Safe navigation is part of production practice but does not displace ordinary navigation; the maintainer attributes selective use to error visibility.	Dot and safe-navigation sites have different semantic requirements, so raw ratios are not preference ratios; the explanation is retrospective.
List quantifiers	FOR_ALL, THERE_IS, and FIRST occur regularly; the maintainer describes them as readable and convenient.	These constructs are established idioms in this transformation codebase and are perceived positively by its lead maintainer.	We do not know the number of sites where an equivalent formulation was possible, so counts do not measure relative value or saved effort.
RESULT	Roughly 510 procedures assign RESULT; about 150 also read it.	Reading the implicit result variable is a recurring accumulation/default-value idiom rather than an exceptional use.	No denominator exists for procedures where an explicit accumulator would be a meaningful alternative; superiority is not evaluated.
String templates	64 templates and 1000 template lines; one makefile generator accounts for 34 templates and 891 lines.	Use is concentrated; the maintainer links acceptance of the syntax to editor support.	The tooling explanation is retrospective, and aggregate template-line counts are dominated by one specialised program.
IMPLIES,	18 uses of IMPLIES and one use of transitive closure in each near-identical configuration.	These constructs have little visible presence in this codebase; the maintainer also reports readability or specialisation concerns.	Especially for , one occurrence supports only an illustrative case and a hypothesis, not a general claim about expressive features.

L2: Readability appears to favour routine use (moderate, interpretive support). The list quantifiers and FIRST occur regularly, and implicit X is used at nearly every eligible site. The maintainer consistently interpreted these constructs in readability terms: they compress common patterns while the code remains easy to understand, with implicit X becoming undesirable when the surrounding context grows long or nested. We therefore treat *readability* as a plausible explanation for their established use in this codebase, not as a quantitatively isolated causal factor. In a long-lived automation platform, that interpretation is operationally relevant because transformation logic must remain reviewable, debuggable, and safely extensible over time.

L3: Error visibility constrains safe-navigation use (moderate, interpretive support). The ! operator is clearly part of production practice, yet ordinary dot access remains much more prevalent. Because the two operators are not interchangeable at every site, that imbalance is not itself a preference measure. The maintainer nevertheless gave a consistent explanation for selective ! use: tolerance of missing values or members can hide mistakes that ordinary access would surface earlier. We therefore formulate the lesson narrowly: in this platform, error visibility was an explicit maintainer concern governing where a convenience feature was considered appropriate. Similar teams may wish to evaluate error-masking behaviour as a maintenance trade-off rather than assuming that a safer-looking navigation operator should replace ordinary access everywhere. This finding is confirmed in other settings, explaining adoption of languages like Rust which are harder to write code in, but this sacrifice is welcomed by many practitioners in exchange for early bug/smell detection.

L4: Tooling can determine whether a usable feature is practically attractive (moderate, interpretive support). String template use is concentrated, and the maintainer did not primarily criticise the feature’s semantics. Instead, he recalled that weak syntax highlighting made embedded variables difficult to distinguish from surrounding text, and that the feature became more acceptable after editor support improved. This is retrospective evidence rather than an experimental estimate of tooling effects, but it identifies a concrete mechanism by which language and editor design interacted in this industrial codebase. For an external DSL maintained over decades, feature design and editor support should therefore be considered together as part of the deployed ASE solution. Tool support or the lack thereof have both been prominently cited as reasons for aiding or limiting the adoption of a language or framework [10, 20, 29, 49].

L5: Specialised expressiveness may fail to become a routine idiom (tentative hypothesis). The transitive-closure operator is the weakest-evidence case in our study: it appears only once in each near-identical configuration. Given the 97–98% corpus overlap, the matching counts cannot be treated as independent evidence of recurring use. The maintainer also recalled an occasion where such a use was later replaced for optimisation reasons. This is not enough evidence to conclude that highly expressive or specialised features are generally poor investments. Instead, it motivates a narrower hypothesis: features that elegantly collapse rare operations into a compact construct may remain exceptional in long-lived industrial code when their use cases are too specialised to recur. Testing that hypothesis would require additional codebases or longitudinal

evolution data; here, `|||` serves only as an illustrative boundary case.

Taken together, these case-grounded propositions suggest a practical evaluation rule for industrial DSL evolution inside ASE platforms: prospective features should be assessed not only for expressiveness, but also for their anticipated interaction with readability, debuggability, tooling, and recurring production use. The evidence here does not rank those factors causally; it shows why they were prominent enough in this long-lived platform to deserve explicit attention when language features are introduced or retained.

9 Threats to Validity

Construct validity. Our principal quantitative measure is the number of syntactic occurrences of selected RcScript features in the maintained transformation source. Such counts capture visible source-level use, but they are merely a proxy for broader constructs such as adoption, usefulness, preference, or maintenance value. In particular, an occurrence count is informative about preference only when the set of eligible opportunities is known. We can identify such a denominator for the implicit `X`, but not generally for `!`, list quantifiers, `RESULT`, or the other features. The interview therefore interprets usage patterns rather than validating the latent constructs independently.

Internal validity. The analysed system has one dominant contributor, and the interviewee is that same lead maintainer as well as a long-time maintainer of the RcScript interpreters. His explanations consequently constitute expert insider interpretation of his own and his colleagues' code, not independent triangulation, and may be affected by retrospective rationalisation or an individual coding style. In addition, membership of the analysed corpus was determined through manual inspection of executed steps and their called modules before the feature counts were automated. This procedure could omit a relevant module, although it was applied consistently to both configurations.

External validity. The study concerns one proprietary DSL and one family of industrial transformation configurations. Moreover, the Bankia and default configurations overlap by 97–98% of their LOC; they are therefore a sensitivity check on a small configuration delta, not an independent replication across projects. The interviewee judged this code broadly similar to other large RcScript codebases within the company, but we did not independently verify that assertion. The results should consequently be transferred analytically to settings with similar long-lived transformation DSLs, rather than statistically generalised to programming languages or industrial developers at large.

Conclusion validity. The study is descriptive and does not perform statistical hypothesis testing. It does not measure runtime frequency, defect density, productivity, maintainability cost, or causal benefit. The quantitative and interview evidence therefore supports statements about what appears in this codebase and how its lead maintainer explains those patterns, but not real causal claims that having the feature itself has produced a maintenance outcome. Accordingly, the lessons in [section 8](#) are case-grounded analytic propositions. They are strongest for patterns with an observable

opportunity denominator, and deliberately more tentative for rare constructs such as `|||`.

10 Conclusion

This showcase reported an industrial experience study of one engineering dimension of a long-lived automated software engineering solution: a proprietary RcScript-based transformation platform used in a PACBASE modernisation pipeline [43]. Across two near-identical production configurations and an interview with the lead maintainer, we observed markedly different levels of feature use; the maintainer associated these differences with readability, error visibility, tooling, and the specialisation of particular constructs.

For practitioners designing and maintaining proprietary DSLs and industrial transformation platforms, the main implication is pragmatic: feature design should be evaluated as much through day-to-day maintainability concerns as through language expressiveness. Our experience suggests that the long-term viability of such a transformation platform concerns not only the availability of expressive language features, but on whether those features support readable, debuggable, and toolable transformation code over many years of industrial use. For future work, similar analysis could be repeated on other RcScript systems and complemented with defect, maintenance, or evolution data to test which of the lessons reported here persist across teams and projects.

Data Availability Statement

The aggregate counts reported in the tables and the edited qualitative observations are already fully included in this paper. A DOI-backed public artefact is not provided because the underlying transformation code, customer-specific configurations, and raw interview materials cannot be released without exposing proprietary implementation details and client-confidential information.

References

- [1] Chris Admiraal, Wouter van den Brink, Marcus Gerhold, Vadim Zaytsev, and Christian Zubcu. 2024. Deriving Modernity Signatures of Codebases with Static Analysis. *Special Issue in the Journal of Systems and Software: Open Science in Software Engineering Research (JSS)* 211 (May 2024). doi:10.1016/j.jss.2024.111973
- [2] Alfred Vaino Aho and Jeffrey David Ullman. 1977. *Principles of Compiler Design*. Addison-Wesley.
- [3] Beatrice Åkerblom, Jonathan Stendahl, Mattias Tumlin, and Tobias Wrigstad. 2014. Tracing Dynamic Features in Python Programs. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR)*. ACM, 292–295. doi:10.1145/2597073.2597103
- [4] Gregory R. Andrews and James R. McGraw. 1976. Language Features for Parallel Processing and Resource Control. In *Proceeding of the DoD Sponsored Workshop on Design and Implementation of Programming Languages (LNCS, Vol. 54)*. Springer, 243–287. doi:10.1007/BFb0021426
- [5] Jean-Pierre Banâtre and Michel Banâtre. 1979. Language Features for Description of Cooperating Processes. In *Proceedings of the Fourth International Conference on Software Engineering*, Friedrich L. Bauer, Leon G. Stucki, and Meir M. Lehman (Eds.). IEEE Computer Society, 308–314. <https://dl.acm.org/doi/abs/10.5555/800091.802952>
- [6] Jan A. Bergstra, Jan Heering, and Paul Klint. 1989. The Algebraic Specification Formalism ASF. In *Algebraic Specification*, Jan Heering and Paul Klint (Eds.). ACM Press, Addison-Wesley, 1–66. <http://hdl.handle.net/1874/13976>
- [7] Darius Blasband. 1995. The YAFL Programming Language. *Journal of Object-Oriented Programming* 8, 7 (1995), 42–49.
- [8] Darius Blasband. 2001. Parsing in a Hostile World. In *Proceedings of the 8th Working Conference on Reverse Engineering (WCORE)*, Elizabeth Burd, Peter Aiken, and Rainer Koschke (Eds.). IEEE Computer Society, 291–300. doi:10.1109/WCORE.2001.957834
- [9] Thomas M. Breuel. 2001. Implementing Dynamic Language Features in Java Using Dynamic Code Generation. In *Proceedings of the 39th International Conference on*

- Technology of Object-Oriented Languages and Systems (TOOLS)*. IEEE Computer Society, 143–152. doi:10.1109/TOOLS.2001.941668
- [10] Antonio Bucchiarone, Federico Cicciozzi, Leen Lambers, Alfonso Pierantonio, Matthias Tichy, Massimo Tisi, Andreas Wortmann, and Vadim Zaytsev. 2021. What is the Future of Modelling? *IEEE Software Insights (IEEE Software)* 38 (2021), 119–127. Issue 2. doi:10.1109/MS.2020.3041522
 - [11] Olav Bunte, Jasper Denkers, Louis C. M. van Gool, Jurgen J. Vinju, Eelco Visser, Tim A. C. Willemse, and Andy Zaidman. 2025. OIL: An Industrial Case Study in Language Engineering with Spoofax. *Software and Systems Modeling* 24, 1 (2025), 139–182. doi:10.1007/s10270-024-01185-x
 - [12] Oscar Callaú, Romain Robbes, Éric Tanter, and David Röthlisberger. 2013. How (and Why) Developers Use the Dynamic Features of Programming Languages: The Case of Smalltalk. *Journal of Empirical Software Engineering (EMSE)* 18 (2013), 1156–1194. Issue 6. doi:10.1007/s10664-012-9203-2
 - [13] Denis Caromel and Manuel Rebuffel. 1993. Object-Based Concurrency: Ten Language Features to Achieve Reuse. In *Proceedings of the 11th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS)*. Prentice Hall, 205–213. <https://dl.acm.org/doi/abs/10.5555/189644.189667>
 - [14] D. Jean Clandinin and Vera Caine. 2013. Narrative Inquiry. In *Reviewing Qualitative Research in the Social Sciences*. Routledge, 178–191. doi:10.4135/9781412963909.n275
 - [15] Benoît Combemale, Robert W. France, Jean-Marc Jézéquel, Bernhard Rumpe, Jim Steel, and Didier Vojtisek. 2017. *Engineering Modeling Languages: Turning Domain Knowledge into Tools*. CRC Press. doi:10.1201/b21841
 - [16] James R. Cordy, Thomas Roy Dean, Andrew J. Malton, and Kevin A. Schneider. 2001. Software Engineering by Source Transformation-Experience with TXL. In *Proceedings of the First International Workshop on Source Code Analysis and Manipulation*. IEEE Computer Society, 170–180. doi:10.1109/SCAM.2001.972678
 - [17] Martin Erwig and Eric Walkingshaw. 2011. Semantics First! — Rethinking the Language Design Process. In *Revised Selected Papers of the Fourth International Conference on Software Language Engineering (SLE) (LNCS, Vol. 6940)*, Anthony M. Sloane and Uwe Almann (Eds.). Springer, 243–262. doi:10.1007/978-3-642-28830-2_14
 - [18] Amir Farooq and Vadim Zaytsev. 2021. There Is More Than One Way to Zen Your Python. In *Proceedings of the 14th International Conference on Software Language Engineering (SLE)*, Eelco Visser, Dimitris Kolovos, and Emma Söderberg (Eds.). ACM, 68–82. doi:10.1145/3486608.3486909
 - [19] Marcus Gerhold, Aliaksei Kouzel, Haroun Mangal, Selin Mehmed, and Vadim Zaytsev. 2024. Modelling of Cyber-Physical Systems through Domain-Specific Languages: Decision, Analysis, Design. In *MoDELS Companion: Proceedings of the Sixth International Workshop on Modelling Language Engineering (MLE)*, Manuel Wimmer, Alexander Egyed, Ed Seidewitz, Antonio Bucchiarone, and Dorian Leroy (Eds.). 1170–1179. doi:10.1145/3652620.3688348
 - [20] Jeff Gray, Kathleen Fisher, Charles Consel, Gabor Karsai, Marjan Mernik, and Juha-Pekka Tolvanen. 2008. DSLs: The Good, the Bad, and the Ugly. In *Companion to the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA)*. ACM, New York, NY, USA, 791–794. doi:10.1145/1449814.1449863
 - [21] Dick Grune, Kees van Reeuwijk, Henri E. Bal, Cerial J. H. Jacobs, and Koen G. Langendoen. 2012. *Modern Compiler Design* (second ed.). Addison-Wesley. https://dickgrune.com/Books/MCD_2nd_Edition/
 - [22] Felienne Hermans, Martin Pinzger, and Arie van Deursen. 2009. Domain-Specific Languages in Practice: A User Study on the Success Factors. In *Model Driven Engineering Languages and Systems*, Andy Schürr and Bran Selic (Eds.). Springer, Berlin, Heidelberg, 423–437. doi:10.1007/978-3-642-04425-0_33
 - [23] Patrick Heymans, Pierre-Yves Schobbens, Jean-Christophe Trigaux, Yves Bontemp, Raimundas Matulevičius, and Andreas Claßen. 2008. Evaluating Formal Properties of Feature Diagram Languages. *IET Software* 2, 3 (2008), 281–302. doi:10.1049/iet-sen:20070055
 - [24] Mark Hills. 2015. Evolution of Dynamic Feature Usage in PHP. In *Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Yann-Gaël Guéhéneuc, Bram Adams, and Alexander Serebrenik (Eds.). IEEE Computer Society, 525–529. doi:10.1109/SANER.2015.7081870
 - [25] Mark Hills. 2015. Variable Feature Usage Patterns in PHP. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Myra B. Cohen, Lars Grunske, and Michael Whalen (Eds.). IEEE Computer Society, 563–573. doi:10.1109/ASE.2015.72
 - [26] C. A. R. Hoare. 1973. *Hints on Programming Language Design*. Technical Report. Stanford University, Stanford, CA, USA.
 - [27] Allen I. Holub. 1990. *Compiler Design in C*. Prentice-Hall.
 - [28] Arnaud Hubaux, Quentin Boucher, Herman Hartmann, Raphaël Michel, and Patrick Heymans. 2011. Evaluating a Textual Feature Modelling Language: Four Industrial Case Studies. In *Revised Selected Papers of the Third International Conference on Software Language Engineering (SLE 2010) (LNCS, Vol. 6563)*, Brian A. Malloy, Steffen Staab, and Mark van den Brand (Eds.). Springer, 337–356. doi:10.1007/978-3-642-19440-5_23
 - [29] John Edward Hutchinson, Jon Whittle, and Mark Rouncefield. 2014. Model-Driven Engineering Practices in Industry: Social, Organizational and Managerial Factors that Lead to Success or Failure. *Science of Computer Programming* 89 (2014), 144–161. doi:10.1016/J.SCICO.2013.03.017
 - [30] Sabine Janssens, Ulrik Pagh Schultz, and Vadim Zaytsev. 2017. Can Some Programming Languages Be Considered Harmful?. In *Proceedings of the Eighth Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU)*, Sarah E. Chasins, Gustavo Soares, and Joshua Sunshine (Eds.). <https://grammarware.net/text/2017/harmful.pdf>
 - [31] Neil D. Jones and René Rydhof Hansen. 2007. The Semantics of “Semantic Patches” in Coccinelle: Program Transformation for the Working Programmer. In *Proceedings of the 5th Asian Symposium on Programming Languages and Systems (APLAS) (LNCS, Vol. 4807)*, Zhong Shao (Ed.). Springer, 303–318. doi:10.1007/978-3-540-76637-7_21
 - [32] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. 2009. Rascal: A Domain Specific Language for Source Code Analysis and Manipulation. In *Proceedings of the Ninth International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE Computer Society, 168–177. doi:10.1109/SCAM.2009.28
 - [33] Ravi Krishnamurthy, Witold Litwin, and William Kent. 1991. Language Features for Interoperability of Databases with Schematic Discrepancies. In *SIGMOD Conference*. 40–49. doi:10.1145/115790.115795
 - [34] Ehud Lamm. 2001. Component Libraries and Language Features. In *Proceedings of the Sixth Ada-Europe International Conference on Reliable Software Technologies (LNCS, Vol. 2043)*. Springer, 215–228. doi:10.1007/3-540-45136-6_17
 - [35] Ralf Lämmel. 2018. *Software Languages: Syntax, Semantics, and Metaprogramming*. Springer. doi:10.1007/978-3-319-90800-7
 - [36] Ralf Lämmel, Martin Leinberger, Thomas Schmorleiz, and Andrei Varanovich. 2014. Comparison of Feature Implementations across Languages, Technologies, and Styles. In *Proceedings of the Software Evolution Week: Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, Serge Demeyer, Dave Binkley, and Filippo Ricca (Eds.). IEEE Computer Society, 333–337. doi:10.1109/CSMR-WCRE.2014.6747188
 - [37] Bruce J. MacLennan. 1983. *Principles of Programming Languages: Design, Evaluation and Implementation*. Holt-Saunders.
 - [38] Fadi Meawad, Gregor Richards, Floréal Morandat, and Jan Vitek. 2012. Eval Begone!: Semi-automated Removal of eval from JavaScript Programs. In *Proceedings of the 27th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Gary T. Leavens and Matthew B. Dwyer (Eds.). ACM, 607–620. doi:10.1145/2384616.2384660
 - [39] Marjan Mernik, Jan Heering, and Anthony M. Sloane. 2005. When and How to Develop Domain-Specific Languages. *Comput. Surveys* 37, 4 (2005), 316–344. doi:10.1145/1118890.1118892
 - [40] Parastoo Moghagheghi, Wasif Gilani, Alin Stefanescu, Miguel A. Fernandez, Bjørn Nordmoen, and Mathias Fritzsche. 2013. Where Does Model-Driven Engineering Help? Experiences from Three Industrial Cases. *Software and Systems Modeling* 12, 3 (2013), 619–639. doi:10.1007/s10270-011-0219-7
 - [41] Steven Muchnick. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann.
 - [42] Terrence W. Pratt and Marvin V. Zelkowitz. 2001. *Programming Languages: Design and Implementation*. Prentice-Hall.
 - [43] Raincode Labs. 2018. PACBASE Migration: More than 200 Million Lines Migrated. <https://www.raincode.com/pacbase/>
 - [44] Raincode Labs. 2020. Bankia Spain Migration: Raincode Labs Completes Successful Migration From PACBASE. <https://www.raincode.com/blog/successful-pacbase-migration/>
 - [45] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. 2011. The Eval That Men Do — A Large-Scale Study of the Use of Eval in JavaScript Applications. In *Proceedings of the 25th European Conference on Object-Oriented Programming (ECOOP) (LNCS, Vol. 6813)*, Mira Mezini (Ed.). Springer, 52–78. doi:10.1007/978-3-642-22655-7_4
 - [46] Caitlin Sadowski and Sri Kurniawan. 2011. Heuristic Evaluation of Programming Language Features: Two Parallel Programming Case Studies. In *Proceedings of the Third ACM SIGPLAN workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU)*. ACM, 9–14. doi:10.1145/2089155.2089160
 - [47] Diomidis Spinellis. 2001. Notable Design Patterns for Domain-Specific Languages. *The Journal of Systems and Software* 56, 1 (2001), 91–99. doi:10.1016/S0164-1212(00)00089-3
 - [48] Sam Staton and Paul Blain Levy. 2013. Universal Properties of Impure Programming Languages. In *Proceedings of the 40th Annual Symposium on Principles of Programming Languages (POPL)*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, 179–192. doi:10.1145/2429069.2429091
 - [49] Federico Tomassetti and Vadim Zaytsev. 2020. Reflections on the Lack of Adoption of Domain Specific Languages. In *STAF Workshop Proceedings (STAF) (CEUR Workshop Proceedings, Vol. 2707)*, Loli Burgueño and Lars Michael Kristensen (Eds.). CEUR-WS.org, 85–94. <http://ceur-ws.org/Vol-2707/oopslepaper5.pdf>
 - [50] Eelco Visser. 2001. Stratego: A Language for Program Transformation Based on Rewriting Strategies. In *Proceedings of the 12th International Conference on Rewriting Techniques and Applications (LNCS, Vol. 2051)*, Aart Middeldorp (Ed.). Springer-Verlag, 357–362. doi:10.1007/3-540-45127-7_27

- [51] Markus Völter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats He-lander, Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. 2013. *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*. <https://dslbook.org>.
- [52] Markus Völter, Bernd Kolb, Tamás Szabó, Daniel Ratiu, and Arie van Deursen. 2019. Lessons Learned from Developing mbeddr: A Case Study in Language Engineering with MPS. *Software and Systems Modeling* 18, 1 (2019), 585–630. doi:10.1007/s10270-016-0575-4
- [53] Beibei Wang, Lin Chen, Wanwangying Ma, Zhifei Chen, and Baowen Xu. 2015. An Empirical Study on the Impact of Python Dynamic Features on Change-proneness. In *Proceedings of the 27th International Conference on Software Engineering and Knowledge Engineering*. KSI, 134–139. doi:10.18293/SEKE2015-97
- [54] Martin P. Ward. 2001. The FermaT Assembler Re-engineering Workbench. In *Proceedings of the 17th International Conference on Software Maintenance (ICSM)*. IEEE Computer Society, 659–662. <https://doi.org/10.1109/ICSM.2001.972783>
- [55] David A. Watt and William Findlay. 2004. *Programming Language Design Concepts*. Wiley.
- [56] David S. Wile. 2001. Supporting the DSL Spectrum. *Journal of Computing and Information Technology* 9, 4 (2001), 263–287. <https://doi.org/10.2498/cit.2001.04.01>
- [57] Niklaus Wirth. 1974. On the Design of Programming Languages. In *IFIP Congress*. 386–393.
- [58] Vadim Zaytsev. 2017. Language Design with Intent. In *Proceedings of the ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, Don Batory, Jeff Gray, and Vinay Kulkarni (Eds.). IEEE, 45–52. doi:10.1109/MODELS.2017.16
- [59] H. Paul Zeiger. 1969. Formal Models for Some Features of Programming Languages. In *Proceedings of the First Annual ACM Symposium on Theory of Computing*. ACM, 211–215. doi:10.1145/800169.805435

Received 2026-04-23; accepted 2026-07-01