The Journal of Systems and Software 218 (2024) 112188

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Software modernization powered by dynamic language product lines☆,☆☆

Walter Cazzola*, Luca Favalli

Università degli Studi di Milano, Computer Science Department, Milan, Italy

ARTICLE INFO

Keywords:
Dynamic language product lines
Language workbenches
Software modernization

ABSTRACT

Legacy software poses a critical challenge for organizations due to the costs of maintaining and modernizing outdated systems, as well as the scarcity of experts in aging programming languages. The issue extends beyond commercial applications, affecting public administration, as exemplified by the urgent need for COBOL programmers during the COVID-19 pandemic. In response, this work introduces a modernization approach based on dynamic language product lines, a subset of dynamic software product lines. This approach leverages open language implementations and dynamically generated micro-languages for the incremental migration of legacy systems to modern technologies. The language can be reconfigured at runtime to adapt to the execution of either legacy or modern code, and to generate a compatibility layer between the data types handled by the two languages. Through this process, the costs of modernizing legacy systems can be spread across several iterations, as developers can replace legacy code incrementally, with legacy and modern code coexisting until a complete refactoring is possible. By moving the overhead of making legacy and modern features work together in a hybrid system from the system implementation to the language implementation, the quality of the system itself does not degrade due to the introduction of glue code. To demonstrate the practical applicability of this approach, we present a case study on a COBOL system migration to Java. Using the Neverlang language workbench to create modular and reconfigurable language implementations, both the COBOL interpreter and the application evolve to spread the development effort across several iterations. Through this study, this work presents a viable solution for organizations dealing with the complexity of modernizing legacy software to contemporary technologies. The contributions of this work are (i) a language-oriented, incremental refactoring process for legacy systems, (ii) a concrete application of open language implementations, and (iii) a general template for the implementation of interoperability between languages in hybrid systems.

1. Introduction

Legacy software is an inherent component of enterprise software systems. Today, organizations confront a daunting dilemma: either they depend on legacy software, crafted with unsuitable abstractions, or they face the considerable costs associated with migrating their codebase to new technological spaces (Ali et al., 2020). This problem is exacerbated when migration becomes a necessity due to the costs of maintaining old hardware—such as mainframes—and hiring programmers with expertise in increasingly less popular programming languages, which become rarer as coders move away from aging languages (Lee et al., 2022). The problem of maintaining legacy software transcends commercial applications and affects public administration as well (Uddin et al., 2022). In 2020, the United States had to *«add COBOL programmers on top of ventilators, face masks and healthcare workers to the list of what several states urgently need as they battle the corona virus pandemic»*.[1] The issue stemmed from the surge in unemployment claims overwhelming a 40-year-old COBOL-based system, necessitating the return of retired COBOL experts to address the problem.[2]

Among modern techniques to improve long term maintainability and reusability of software artifacts, some organizations started adopting *software product line* (SPL) engineering for building variable software systems with an high degree of quality at a lower cost (Capilla et al., 2014). The shift towards more dynamic architectures, capable of

☆ Editor: Laurence Duchien.

☆☆ This work was partly supported by the MUR project "T-LADIES" (PRIN 2020TL3X8X).

* Corresponding author.
E-mail addresses: cazzola@di.unimi.it (W. Cazzola), favalli@di.unimi.it (L. Favalli).

[1] Wanted urgently: People who know a half century-old computer language so states can process unemployment claims https://edition.cnn.com/2020/04/08/business/coronavirus-cobol-programmers-new-jersey-trnd/index.html

[2] COBOL Programmers Answer Call to Shore Up Unemployment Benefits Systems https://spectrum.ieee.org/cobol-programmers-answer-call-unemployment-benefits-systems

https://doi.org/10.1016/j.jss.2024.112188
Received 30 January 2024; Received in revised form 11 June 2024; Accepted 17 August 2024
Available online 28 August 2024

adapting to varying conditions through autonomous decision-making, led to the creation of development approaches such as *dynamic software product lines* (DSPL) (Hallsteinsen et al., 2008). These approaches bind variation points at runtime by monitoring the situation and controlling the adaptation process.

In this work, we try to solve some of the issues behind the refactoring of legacy systems by using SPL concepts. We introduce *dynamic language product lines* (DLPL)—*i.e.*, DSPLs in which the adaptive system is a programming language interpreter or a compiler—and a modernization process based on DLPLs. The objective is to manage the costs of migrating outdated systems to modern technologies by spreading them across several iterations. To accomplish this objective, the core interpreter for the legacy programming language is reconfigured at runtime with different, and potentially dynamically generated, micro-languages (Cazzola et al., 2018). The orchestration among all language variants throughout the system's life cycle enables developers to extend and replace algorithms and data structures originally written in the legacy language with code that adheres to the new syntax. The process is incremental, as it allows for the coexistence and interoperation of old and new code until a complete refactoring becomes feasible. Meanwhile, the availability of the system must not be suspended, and new features can already be developed using the modern language. Eventually, the whole system is modernized and the support for the legacy programming language—as well as the need for programmers experienced in its usage—can be finally abandoned. By choosing a dynamic approach over a static one, developers do not need any form of preprocessing and can disregard any incompatibilities between the legacy features and modern features, both on a syntactical and on a semantical level.

We present this process and demonstrate its practical application with the exemplary case of COBOL. By considering a COBOL interpreter developed with the Neverlang language workbench and a COBOL application, we showcase their evolution towards an hybrid COBOL-Java system. Additionally, we highlight the language variants that enable this transformation and discuss how these variants are not dependent on the chosen language. Through this work, we hope to provide a generic and reproducible approach to plan and perform the incremental modernization of legacy systems. We also detail a viable strategy to create flexible and robust hybrid systems that support interoperability between different languages in a non-verbose manner.

The remainder of this paper is structured as follows. Section 2 introduces preliminary concepts needed to present the contributions of the paper. Section 3 presents some of the known challenges in the context of software modernization and the respective solutions. Section 4 contains our contributions—*i.e.*, the concept of DLPL and their usage within a legacy software modernization process. Section 5 discusses the implementation of a DLPL for COBOL and its interoperability with Java. In Section 6, the COBOL DLPL is used to demonstrate the applicability of the modernization process by showing the transition from a COBOL system to Java through several iterations. In Section 7 we acknowledge any threats to the validity of our process and its demonstration case study. Section 8 contains works related to our contribution. Finally, in Section 9 we draw our conclusions on this research and outline some of our future directions.

2. Background

This section provides an introduction to the research context and presents any necessary terminology for understanding the contribution of this work.

2.1. Dynamic software product lines

Product lines are a staple in industrial production. SPL engineering introduces the concepts of software variants and software families for the creation of variability-rich software systems using concepts derived from traditional product line engineering. A software family comprises related yet distinct software variants that vary based on the features they offer. SPL engineering combines domain engineering concepts for designing and implementing software artifacts with application engineering concepts to create products from a configuration—*i.e.*, a subset of all available features (Apel et al., 2013).

While traditional SPL engineering acknowledges the variability of the requirements, such variability is usually delivered at compile-time and is not supported at runtime. DSPL engineering, building on SPL engineering concepts, retains the binding between variability points and variable code at runtime (Hallsteinsen et al., 2008). DSPLs contain an explicit representation of the configuration space, as well as the constraints that describe permissible runtime reconfigurations of the system (Hinchey et al., 2012). The DSPL can be either self-adaptive or adapted externally by a human.

2.2. Language product lines

The concept of applying SPL concepts to the creation of language variant families has gained popularity among researchers and practitioners (Ghosh, 2011; Kühn et al., 2014; Cazzola and Poletti, 2010), leading to the emergence of *language product lines* (LPLs) (Vacchi et al., 2013; Kühn et al., 2015; Kühn and Cazzola, 2016). The LPL approach proves useful for creating variants of domain-specific (Crane and Dingel, 2005; Tratt, 2008; Vacchi et al., 2014) and general-purpose languages (Cazzola and Olivares, 2016). LPL engineering benefits from the development of *modular compilers*, supporting the separate development of language syntax, semantics, and meta-data for reusable IDE specifications. Modern *language workbenches* (Erdweg et al., 2015; van den Brand, 2023) embrace this philosophy to enhance the reusability and maintainability of language assets.

Fowler (2005) introduced the term *language workbench* as a tool suitable for supporting the *language-oriented programming* paradigm (Ward, 1994), where complex software systems are built around a set of domain-specific languages (Fowler and Parsons, 2010) to express domain problems and their solutions effectively. While the original definition focused on projectional editing (Fowler, 2005; Pech, 2021), current research on language workbenches centers on their promise to support the efficient definition, reuse, and composition of language implementations (Leduc et al., 2020; Cazzola and Favalli, 2022; Pfeiffer, 2023) and any related tools (Barros et al., 2022) beyond the desktop environment (Warmer and Kleppe, 2022). Although language workbenches have evolved based on various design philosophies, they share the capability to reuse existing implementations across several variants of a base language (Erdweg et al., 2012; Leduc et al., 2020; Bertolotti et al., 2023a). In the context of LPL engineering, a reusable part of a language specification is termed a *language feature* (Liebig et al., 2013). Languages and their features can be composed to create new language variants through five forms of language composition: language extension, language restriction, language unification, self-extension, and extension composition (Erdweg et al., 2012). Language workbenches have the goal of supporting all five forms of language composition through dedicated abstractions. We dub the collection of all language features involved in the implementation of a specific application feature as *micro-language* (Cazzola et al., 2018). Micro-languages are not necessarily self-contained; that is, their grammar can include open nonterminals and unreachable nonterminals. Moreover, they can potentially overlap, meaning different micro-languages may share some of their features.

2.3. Neverlang language workbench

Neverlang (Cazzola, 2012; Cazzola and Vacchi, 2013; Vacchi and Cazzola, 2015) is a language workbench for the modular development of programming languages. Listing 1 illustrates an excerpt of the implementation of LogLang, a family of languages for the definition

```
 1 module Backup {
 2   reference syntax {
 3     Backup ← "backup" String String;
 4     Cmd    ← Backup;
 5     categories  : Keyword = { "backup" };
 6     in-buckets  : $1 ← { Files }, $2 ← { Files };
 7     out-buckets : $1 → { Files }, $2 → { Files };
 8   }
 9   role(execution) {
10     0 .{
11       String src  = $1.string, dest = $2.string;
12       $$FileOp.backup(src, dest);
13     }.
14   }
15 }
16 slice BackupSlice {
17   concrete syntax from Backup
18   module Backup with role execution
19   module BackupPermCheck with role permissions
20 }
22 language LogLang {
23   slices BackupSlice RemoveSlice RenameSlice
24          MergeSlice Task Main LogLangTypes
25   endemic slices FileOpEndemic PermEndemic
26   roles syntax < terminal-evaluation < permissions : run
27 }
```

Listing 1 Syntax and semantics for the LogLang backup task.

of log rotating tasks, similar to the logrotate UNIX utility. The basic elements (modules, lines 1–15) define the syntactic and semantic assets that can be composed into language features. Slices (lines 16–20), embody the concept of language features that can compiled, tested, and distributed separately, so that the same artifacts can be shared among different language implementations without encompassing recompilation, in accordance to the constraints set by the language extension problem (Leduc et al., 2020).

The Backup module shown in Listing 1 declares a reference syntax for the backup task (lines 2–8), containing two grammar productions. Each role, introduced by the keyword **role**, defines a compilation phase by declaring semantic actions that should be executed when some syntax is recognized, as prescribed by the *syntax-directed translation* technique (Aho et al., 1986). Semantic actions are attached to nonterminals of the productions (lines 10–13) by referring to their position in the grammar: numbering starts with 0 and grows from the top left to the bottom right.[3] Thus, the Backup nonterminal on line 3 is referred to as `$0` and the two String nonterminals on the right-hand side of the production as `$1` and `$2`, respectively. Attributes are accessed from nonterminals using the same criterion by dot notation as in line 11. In contrast, the BackupSlice slice (lines 16–20) composes syntactic assets (the reference syntax from the Backup module, line 17) and semantics assets (from two separate roles of two different modules, lines 18–19). Finally, the **language** descriptor (lines 22–27) composes several language features into a complete language interpreter or a compiler (lines 23–24). Therefore, composition in Neverlang is twofold: (1) between modules, which yields slices, and (2) between slices, which yields a language implementation. Language features can also be composed into bundles—*i.e.*, languages but that can be embedded into other languages. In this work, bundles embody the concept micro-languages.

Semantic actions are performed with respect to the parse tree of the input program; roles are executed in sequence and all visits of the tree are listed within the **roles** clause (line 26) of the **language** descriptor, e.g., permission is executed after parsing and terminal-evaluation. Besides, the **language** clause can declare **endemic slices** whose instances are shared across multiple compilation phases (line 25).[4] Neverlang supports LPL engineering through the AiDE (Vacchi et al., 2013, 2014) variability management environment. LPLs based on Neverlang and AiDE can be developed in a asynchronous and agile manner (Cazzola and Favalli, 2022, 2023a).

3. Known challenges and solutions

Continuous refactoring and modernization of legacy software to modern technologies is a well-known problem (Bélády and Lehman, 1976). The pursuit of powerful abstractions and domain-oriented solutions drives developers to hybridize programming languages with richer features, aiming to bridge the gap between human and machine. Languages initially conceived as purely object-oriented now incorporate functional constructs, such as the functional interface introduced in Java 8, and vice versa. When a paradigm is not supported by the base language, the community often develops its own language extensions, as happened with AspectJ (Kiczales et al., 2001) for aspect-oriented programming and LambdaJ[5] for functional programming in Java *ante litteram*. As a result, modern programming languages have evolved into complex software systems that tend to influence each other. Programs written in these languages require constant updates in accordance to the latest changes, ensuring they can fully leverage the functionalities offered by the languages.

Yet, legacy software continues to exist due to its sheer volume. COBOL alone is estimated to have at least 220 billion lines of code worldwide and it constituted 43% of all banking systems in the USA in 2017 (Deknop et al., 2020). Migrating entire COBOL codebase towards a modern language may be unfeasible or even undesirable: migration is challenging and error prone, especially when done automatically and when dialects of the same language are involved (Terekhov and Verhoef, 2000). However, legacy languages are eventually decommissioned and reliance on modern technologies becomes a necessity. Companies are then faced with a choice (Zaytsev, 2017):

– *language conversion*—the entire codebase is converted to conform to the new language;
– *automated refactoring*—the legacy code is refactored to eliminate harmful patterns and to incorporate more modern language features;
– *complete reengineering*—the existing codebase serves as a reference framework for the logic, but the system is reengineered using modern tools and technologies;
– *compiler reimplementation*—the codebase is kept intact, but the legacy compiler is updated to include modern features.

Each of the four options discussed above has its limitations. First and foremost, they involve suspending the availability of the system. Large-scale businesses cannot afford offline periods, and they must maintain a consistent pace of updates (Vu et al., 2014). Hence, while the new version of the system is under development, the legacy code must remain operational and regularly updated. With each new update, the new version of the system suffers a shift in the requirements, which in the worst case may render the new version already obsolete upon completion. A clear solution in this regard is language interoperability: if the execution of old code and new code can be interleaved, and if data structures can be shared between the two languages, then updates can be delivered using the new language while the legacy code is modernized incrementally. Some modern technologies, like GraalVM, support a polyglot approach involving an expanding pool of diverse programming languages within the same environment (Šipek et al., 2019). However, their solution may not be applicable to specific modernization scenarios among proprietary domain-specific languages or other fourth-generation programming languages (Tharp, 1984). For

[3] Neverlang also provides a labeling mechanism for productions, so that nonterminals are referred via an offset from such a label, e.g., `$BKP[1]` is the first nonterminal from the right-hand side of the BKP production.

[4] Please see Vacchi and Cazzola (2015) for further details.

[5] https://github.com/mariofusco/lambdaj

the vast majority, constructing interfaces between languages is challenging (Chisnall, 2013), as seen in the verbose, boiler-plate-heavy code required for the usage of the Java Native Interface (Hirzel and Grimm, 2007).

4. DLPL-driven modernization approach

The sentiment driving towards the interoperability of old code and the new code must not come at the cost of degrading software quality due to the introduction of complex compatibility layers. Therefore, the language implementation should be flexible enough to smoothly introduce language features that support the conversion of data types while leaving the developers unaware. Then, it should be possible to remove these features when the compatibility layer is no longer needed. The introduction and removal of these features should not impact the code written by developers but rather only the two language implementations. This section presents a modernization approach compliant with this sentiment, thanks to a family of variants of the legacy programming languages, for which we support dynamic adaptation capabilities by introducing the concept of dynamic language product lines (DLPL). This approach is better suited to the development of new programming languages from scratch since it is rare for legacy programming languages to be written in a feature-oriented manner and therefore to be framed as a DLPL. In such cases, the code developed according to the language can remain relevant across several language revisions, and even if the language is eventually decommissioned. However, the same approach is applicable to already existing legacy programming languages, albeit with an initial overhead tied to the creation of a (possibly incomplete) implementation for the legacy programming language.

4.1. Dynamic language reconfiguration

Improving language development requires dealing with usual concerns such as software evolution and component reusability. Just as any other piece of software, programming languages evolve over time. As new features are added to the language, so do the syntax and the semantics of the language. It is impossible to foresee all the future requirements, and due to the inherent nature of parsing algorithms, new additions may introduce inconsistencies and incompatibilities. For instance, a purely LR(1) grammar may become non-LR(1) by adding a new production as a result of the introduction of a new language feature.

A similar consideration can be made with regards to the semantics. For instance, due to the evolution of the requirements, the same language construct may have different semantics depending on the context in which it is being used. Consider something as simple as a variable declaration. Each programming language handles memory in its own way, even though several languages share an identical syntax for variable declaration. In contexts in which several languages must coexist and interoperate in an hybrid system as we do in this work, the same linguistic construct may operate differently depending on the context—*e.g.*, whether the part of the code that is currently being executed is either COBOL or Java.

Both problems can be solved statically. Inconsistencies in the grammar can be solved by manually rewriting the grammar or changing parsing algorithm, at the cost of reducing the long-term reusability of existing software artifacts and possibly affecting its performance. The semantics can instead be changed by modifying the software artifacts and adding additional `if` or `switch` statements to their implementation, thus degrading their readability and maintainability over time. Modifying existing artifacts to accommodate new features is especially daunting when the grammar is large and complex or when the features added to the language were never intended to be used within the same grammar, as we will see with regards to the COBOL-Java interaction throughout this work. Moreover, once the refactoring is complete and the interoperability features are no longer needed, these changes must later be reverted, introducing further overhead to the process.

Conversely, using a dynamic language reconfiguration approach, while not mandatory, can improve the maintainability and separation of concerns of the system. Updating the grammar *on-the-fly* by adding and removing productions depending on the parsing context—*e.g.*, the part of source code being currently parsed is written in COBOL—can ensure that the parser may never encounter parsing conflicts without having to change the original grammar. From an implementation standpoint, grammar reconfiguration can be achieved through dynamic shrinkage and growth of LR goto-graphs (Cazzola and Vacchi, 2014), which involves the addition of new states and transitions, as well as merging and splitting states whose kernel may be changed in the process, without affecting the remaining structure of the graph. Semantics can also be dynamically updated. By changing the semantics associated to specific constructs—*i.e.*, specific nodes within the parse tree—based on the execution context it is possible to keep the separation of concerns, so that different implementations of the same constructs across several languages are kept separated and loaded on demand. This capability must be exposed by the language API through methods of inspecting and changing the internal state of the compiler/interpreter. A more sophisticated implementation may leverage aspect-oriented programming (Kiczales et al., 1997) concepts adapted to language development (Cazzola and Shaqiri, 2017), such as weaving different semantics to parse tree nodes depending on their structure and/or other conditions—*e.g.*, parse tree attribute values and their types—and even external factors—*e.g.*, energy consumption constraints (Cazzola et al., 2018). By keeping the alternative implementations of syntax and semantics separate, once the refactoring process is complete, it suffices to disable the reconfiguration capabilities and to remove any legacy features from the language, without needing to perform further changes to the compiler.

4.2. Dynamic language product lines

LPLs and DSPLs are distinct evolutions of the traditional SPL engineering process. LPLs treat programming languages as SPLs, where features correspond to language constructs comprised of a syntactical and a semantical implementation, whose composition results in the creation of a language variant. DSPLs, on the other hand, focus on the need of producing software capable of adapting to fluctuations in the user needs and the environment. These two techniques can be combined so that languages are treated as dynamic language product lines (DLPL). In a DLPL, the language can reconfigure itself into a different variant of the family by disabling some of its features and enabling others. The result is a language implementation with syntax and semantics adaptable to the program being executed. The degree to which the language implementation can be adapted to the program is tied to the variability space and the available language features. In Section 4.3 we discuss how a DLPL suits the needs of a legacy system modernization scenario.

4.3. Process

Fig. 1 summarizes the proposed modernization process.

Running example. For the sake of discussing this process, let us introduce a legacy application to be used as a running example throughout this section. The application serves as a minimal handler for banking accounts, offering features such as authentication, deposit, withdrawal, balance, and transaction history. For simplicity, this section only focuses on any abstract details that are useful to present the process, especially on the authentication and deposit application features, as well as the corresponding micro-languages, both legacy and modern. Instead, the actual implementation of such an application will be presented in Section 6. All references to this running example will be highlighted as follows:

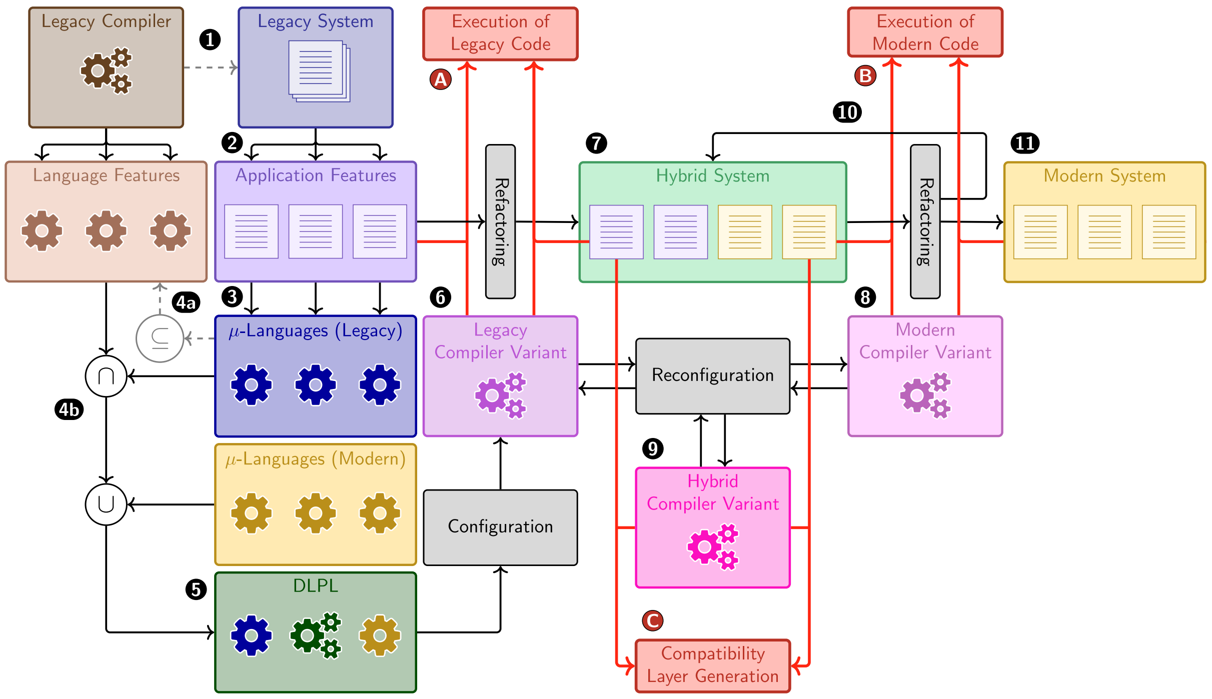

Figure 1. Main actors involved in the DLPL-driven modernization approach and their interaction.

Example: Description.

Initial system. The initial system comprises a set of source files compliant with the syntax of the legacy programming language. The system is traditionally executed by feeding the source files to a compatible language implementation (Fig. 1-❶). This process is applicable to both compilers and interpreters, but for brevity, the rest of this work will specifically refer to compilers. Any traditional modernization process involves rewriting a portion of the system or a section of the compiler, with varying extent over the scope of the entire system. Generally, as symbolized by the dashed arrow, changes to the system do not affect the compiler, whereas the opposite is not true: any modification to the compiler may cause an update to the software system. The proposed process deviates from this norm as it involves several coexisting language variants, in a polyglot and incremental modernization approach. This approach was inspired by a prior work, wherein micro-languages were employed to provide technical sustainability support for new and existing concerns through flexible language-level programming (Chitchyan et al., 2015).

Legacy compiler modularization. The initial step involves identifying the application features present within the legacy system (Fig. 1-❷). In SPL fashion, each application feature represents a coherent subset of the functionalities offered by the entire system.

Example: We already mentioned some application features of our running example: authentication, deposit, withdrawal, balance, and transaction history. Many others may be present, such as logging and user assistance/help.

It is rather common for each of these application features to be implemented using several language features. Similarly, the same language feature may be used across several application features. Therefore, the micro-languages needed by the system may share some of their features. Each application feature is associated to the micro-language that supports its execution (Fig. 1-❸).

Example: The authentication logic application feature is developed using several language features and libraries, that together form a micro-language. Consider a naive two-step authentication in which (1) a database is queried to retrieve the user data according to the provided username and (2) the provided password is matched against that retrieved from the database. Implementing such an application feature requires a variety of language features, such as user input, database querying, strings comparison, Booleans, and if statements, whereas many other language features, such as arithmetic expressions and loops are not needed. Between the two, only the former set of features will form the authentication micro-language.

The same process is repeated for all the application features, each time deriving a new micro-language comprised of different (but possibly overlapping) language features.

Example: The deposit feature will need both language features that overlap with the authentication (such as, input/output to read the amount to be deposited) and language features that do not overlap (such as, arithmetic expressions).

The collection of all language features derived in this way—*i.e.*, they are part of at least one micro-language—is then reified into an executable language artifact using a language workbench. In this phase, all micro-languages and language features represent a fragment of the legacy compiler; thus, the system has not undergone any modernization steps yet. Instead, these features form a subset of the features offered by the original legacy compiler (Fig. 1-4a). This aspect is crucial to the process and its usability, as there is no need to rewrite the entire compiler before being able to modernize the system. Only language features that are actually part of a micro-language for—*i.e.*, the intersection between the set of features of the whole language and those of forming the micro-languages (Fig. 1-4b)—any of the application features of the legacy system need to be developed.

DLPL creation. Now, the modernization process can commence. Employing the same language workbench selected in the preceding step, we create any needed modern micro-languages. In other terms, the development team identifies the application features essential for inclusion in the modern system and subsequently creates any micro-languages whose language features are needed in their realization.

Example: If the only needed application features (and corresponding micro-languages) are authentication and deposit, the team will create the following language features: user input, database querying, strings comparison, Booleans, if statements, and arithmetic expressions.

The union between the legacy micro-languages and the modern micro-languages results in the creation of the DLPL, which will be employed throughout the subsequent stages of this process (Fig. 1-❺). It is important to note that the DLPL is not set in stone; should the requirements change in the future, new language features can be seamlessly incorporated into the product line in an agile manner (Cazzola and Favalli, 2023a).

Legacy code execution. During this phase, it is possible to run the legacy system through the use of the modularized compiler. All the steps discussed so far represent the initial overhead of this process and can be bypassed if the legacy compiler was initially conceived as a DLPL, or if it was already refactored into a DLPL during a prior modernization. In a sense, this process is future-proof, as implementing the compiler in terms of a DLPL makes it easier to incorporate additional language features compared to creating a new language implementation from scratch. To begin compiling and running the legacy system, an initial configuration must be derived. This initial configuration can be carried out by a domain expert, possibly with a configuration editor (Cazzola and Favalli, 2022). We refer to the resulting variant as the Legacy Compiler Variant, as it is semantically equivalent to the legacy compiler, excluding any features that are not part of any micro-language, which are omitted. Assuming the language features were implemented correctly, feeding the legacy code to the Legacy Compiler Variant will result in its compilation and execution with no semantic differences (Fig. 1-Ⓐ).

System refactoring. Traditionally, incremental modernization is performed at service level and may cause computational overhead—*e.g.*, due to use of serialization and deserialization—or degrade software quality due to the introduction of compatibility layers between different languages. In the worst case scenario, incremental migration may not be possible, requiring the whole system to be migrated at once. Conversely, this process aims at allowing incremental migration at any level of granularity by entailing the creation of a backlog of legacy application features slated for modernization. These features do not necessarily have to be implemented concurrently; instead, they can be iteratively selected and updated. The result of each iteration is a hybrid system that incorporates both legacy application features and modern application features (Fig. 1-❼).

Example: The development team can start migrating the authentication feature to the modern language, while still using legacy language features to implement the deposit feature.

The execution of such a system is driven by the DLPL and requires the presence of at least three language variants: the aforementioned Legacy Compiler Variant, as well as the later discussed Modern Compiler Variant (Fig. 1-❽) and Hybrid Compiler Variant (Fig. 1-❾). In some cases—*i.e.*, when the language workbench and the target of the modernization process share their runtime environment—the implementation of the Modern Compiler Variant can be omitted, as exemplified in Section 5. Thanks to the DLPL, the configuration can be updated based on the specific application feature under consideration. Similar to a prior study in which the reconfiguration capabilities of a base language were employed to generate a family of mutants (Cazzola and Favalli, 2023b), the language implementation provides reflective capabilities. When providing each application feature to the DLPL, introspection (Tanter, 2009) is used to detect the need for reconfiguration, while intercession (Tanter, 2009) is used to perform the reconfiguration seamlessly—*i.e.*, without interrupting the execution and without re-compilation. As source code is fed to the compiler, the compiler dynamically switches between variants: the processing of legacy code is managed by the Legacy Compiler Variant, and the modern code, written with the new syntax, is managed by the Modern Compiler Variant (Fig. 1-Ⓑ).

Example: Upon performing the authentication, the system will reconfigure into the Modern Compiler Variant, switching back to the Legacy Compiler Variant upon performing a deposit.

In most cases, these two variants are not sufficient. Achieving interoperability between legacy and modern languages necessitates the preservation of data—*e.g.*, the user obtained by performing the authentication with the Modern Compiler Variant is the same user that will perform the deposit using the Legacy Compiler Variant. Furthermore, both languages must manipulate such data according to their respective syntax: the coexistence of the two languages should not cause the introduction of a hybrid syntax. Such a solution would be undesirable, as the system would undergo refactoring twice: first from legacy to hybrid, and then from hybrid to modern. Instead, the two syntaxes are kept distinct at all times, while the data is shared. To address this requirement, the transition from the Legacy Compiler Variant to the Modern Compiler Variant (and vice versa) is facilitated through a reconfiguration towards a third variant: the Hybrid Compiler Variant. The Hybrid Compiler Variant does not execute or compile any code but orchestrates the migration of data between the other two variants by creating a compatibility layer that developers are entirely unaware of (Fig. 1-Ⓒ). Admittedly, the introduction of this compatibility layer incurs a performance overhead upon reconfiguration, but we consider such overhead acceptable based on three reasonable assumptions. Firstly, not all data requires adaptation, as one of the two languages may lack the syntactic complexity to handle certain data types. For instance, adapting a COBOL data structure to Java makes sense, as they can be accessed as arrays and classes, while the reverse may not be true since COBOL lacks constructs for invoking methods on objects. Moreover, all COBOL variables must be declared within the `DATA DIVISION`, making it illogical for a Java fragment to introduce additional variables. The preferred approach is for the Java code to perform computations and then modify the value of an existing COBOL variable according to the results, before resuming control to the Legacy Compiler Variant. Secondly, we can assume that reconfigurations are infrequent. If the execution of two modern application features is interspersed by a legacy application feature, there are two possibilities: either the legacy application feature is very complex and long-running, or it would be more logical to modernize it as well to have modern code running contiguously. Lastly, given that we are dealing with a modernization scenario, all legacy application features are expected to be eventually replaced by their modern counterparts. Therefore, this compatibility layer is temporary and will no longer be necessary once the refactoring is complete.

Discontinuing legacy language features. As mentioned earlier, language features may be reused across several micro-languages or application features; modernizing a specific application feature renders the corresponding micro-language obsolete, but it does not guarantee the obsolescence of all related language features. Conversely, all language features remain essential as long as they are part of at least one micro-language.

Example: After the refactoring, the legacy authentication micro-language became obsolete; by extension the legacy implementation of the Booleans language feature may also become obsolete: this feature is discontinued if and only it is not used across any other non-obsolete micro-language.

This could pose a challenge, because language features from the legacy and modern language sharing the same concern may not coexist within the same language. This requirement is incompatible with the needs of an incremental modernization process, because modern language features may require the removal of all incompatible legacy language features before being introduced. The approach proposed in this work does not face this limitation and can accommodate conflicting language variants within the same DLPL. The motivation is intuitive: since the two language features are part of different language variants (Legacy Compiler Variant and Modern Compiler Variant respectively), they never coexist. Therefore, any mutual exclusivity between the two is irrelevant, as an invalid variant can never be generated. While the legacy language feature is necessary for compiling/executing legacy code, it can be bug-fixed and updated without affecting already modernized application features. However, once a legacy language feature is no longer part of any micro-language, it becomes unnecessary, and development support can be dropped. This streamlines future development by saving time that would have been invested in updating language features with no impact on any application feature within the system.

Adapting to language-specific requirements. The presented process illustrates a DLPL consisting of three language variants. While this represents the minimum number of variants necessary for implementing the process, it is important to note that the process itself is applicable to DLPLs with more variants. As we will discuss in Section 6, the COBOL DLPL consists in several variants, encompassing platform-based variants (GNU COBOL and IBM COBOL) and dynamically-generated variants (custom syntax for separators and currency symbols). In this context, the DLPL adheres to the base process, but also uses reflection to dynamically adapt the configuration according to the platform and the specific syntax used within the source code of the legacy application features.

Process completion and iteration. The modernization process can span several iterations, each involving selection of legacy application features for modernization. Each iteration yields a new version of the hybrid system, consisting of different legacy and modern application features (Fig. 1-❿).

Example: The authentication application feature may be refactored as part of the first iteration, deriving both a modern micro-language and the corresponding modern application feature, whereas deposit may be refactored during the second iteration.

This process also entails iteratively adjusting the DLPL by incorporating modern language features and removing legacy ones that are no longer needed, as previously detailed. Eventually, all application features are modernized, marking the completion of the modernization process. The system now exclusively comprises modern application features, thus making it a fully modern system (Fig. 1-⓫). While the evolution of a software system is never truly complete, the same process can be repeated for future modernization needs, treating the modern system as the new legacy system and the Modern Compiler Variant as the new Legacy Compiler Variant. Performing the modernization will require the creation of all relevant language features, but with considerably less effort, as the Legacy Compiler Variant will already be available from the preceding modernization.

4.4. Limitations and other considerations

The modernization process has limitations that warrant consideration. Language workbenches are typically used for implementing domain-specific languages. Therefore, being based on a DLPL written using a language workbench, this process may be better suited to the modernization of domain-specific languages rather than general-purpose ones. Notably, fourth-generation programming languages—i.e., domain-specific languages with a high abstraction level compiled towards a general-purpose third-generation programming language—often have a shorter lifespan compared to third-generation programming languages (Voelter, 2013) and can even cause technical debt (Zaytsev and Fabry, 2019). Eventually, they are decommissioned in favor of newer domain-specific languages or general-purpose programming languages. Modernizing according to the process presented in this work potentially represents its most effective application, as the process can be implemented without the initial overhead associated with creating a modular version of the legacy language.

Table 1
COBOL DLPL language decomposition.

Concern	# Language Features
COBOL Statements	51
IBM COBOL Features	1
GNU COBOL Features	1
Shared COBOL Features	7
Identifiers	11
Literals	8
Error codes	15
Identification Division	8
Environment Division	11
Procedure Division	8
Arithmetic Operations	7
Files	11
Utils	8
Java Statements	6
Others	164

Conversely, large-scale modernization attempts often target a legacy general-purpose programming language, such as COBOL. This presents two challenges. Firstly, general-purpose programming languages, particularly older ones, are typically implemented monolithically. Therefore, reusing existing implementations to facilitate the creation of a DLPL is unlikely. Secondly, re-writing a compiler for a general-purpose programming language is significantly more complex than doing so for a domain-specific one. Developing a high-quality COBOL parser, for instance, has been reported to take up to three years according to professionals.[6]

Nevertheless, adopting an incremental modernization approach, as proposed, does not suspend the availability of the legacy system. It enables continuous updates to the original legacy code while simultaneously developing modern application features. Furthermore, due to the modular nature of this approach, developers may consider omitting complex language features to facilitate the process. Language features can be omitted from the DLPL whenever they are used by any application feature or when by modernizing the application features that require them upfront. In this regard, each development team may want to find the preferred trade-off based on the system under consideration.

Another limitation is tied to the dynamic nature of the language implementation. To the best of our knowledge, dynamic adaptation of the syntax and/or the semantics of a compiler is typically not supported by language workbenches. Consequently, depending on the chosen workbench, the same approach may necessitate the hybrid system to be re-compiled/executed multiple times, introducing additional overhead on performance.

5. COBOL language product line

To test the applicability of the process presented in Section 4, we implemented a partial yet feature-rich LPL of COBOL variants using Neverlang. Available language features include most important aspects of COBOL, including (but not limited to):

- data division (with file, working storage, local storage, and linkage sections);
- environment division (with configuration section);
- identification division (with program name, type, author and other information);
- procedure division (callable as a subprogram through the `CALL` statement, with the ability of using and returning variables);
- arithmetic expressions;

[6] https://compilers.iecc.com/comparch/article/98-05-108

– display;
– file control (sequential, random, and dynamic);

The COBOL LPL includes a standalone variant capable of executing COBOL programs that comply with the available language subset. We do not delve into the complete implementation details as they are not pertinent to presenting the modernization process. Instead, we offer basic information on the overall LPL and concentrate on the peculiarities and the key language features provided by each variant used throughout the modernization process. The LPL is structured as an hybrid Neverlang-Java project. The Java source comprises 118 classes, totaling 11,276 lines of code. The Neverlang part of the project consists of 9,086 lines of code. While the subdivision of Java code into classes does not affect the variability space of the COBOL LPL, Neverlang's modularization aspects are more significant. Each Neverlang component represents a language fragment that can be either included or removed from the configuration. Some modules can be further customized at runtime to dynamically generate a variant for handling user-defined syntax or incorporating additional semantic behavior. The COBOL LPL contains the following Neverlang units:

– 310 modules;
– 7 slices;
– 4 endemic slices;
– 25 bundles;
– 5 languages.

With regards to this language decomposition, the language features (modules and slices) were logically divided according to the concern they implement, with a bundle for each concern. The most relevant concerns are listed in Table 1, with less relevant ones grouped under the "Others" concern for brevity. As a reminder from Section 2, it is important to note that both modules and slices can embody the concept of a language feature. Both modules and slices can be included in a language or bundle. However, slices are specifically designed for reusing and combining syntax and semantics from different modules to create a novel language feature while avoiding code duplication.

It is important to note that, despite incorporating traditional product line concepts like features and variants, the COBOL LPL is not modeled in terms of a feature model. Therefore, there is no model describing the dependencies between all available language features and a configuration cannot be derived in a traditional fashion—*i.e.*, by selecting features from a feature model within a configuration editor. In our context, this would not be feasible, as some language features are dynamically generated and are not statically available. This is not a concern, since all reconfigurations are pre-determined and therefore the DLPL cannot incur into an invalid configuration despite it lacking a feature model. All variants are thus either hand-crafted or generated on-the-fly: the execution always begins with a default Configuration Variant (refer to Section 5.1) that is later refined based on environmental conditions. The aforementioned Neverlang language with the units are not executable, with the exception of the Configuration Variant; instead, they all serve as templates to be completed at runtime, as detailed later.

5.1. Configuration variant

The Configuration Variant is one of only two variants that exist at startup, alongside the Java variant. The Configuration Variant does not execute any code; instead, it performs a broad parsing of the source code and inspects the system. This allows for the subsequent refinement of the configuration. The initial parser uses a coarse tokenizer to speed up the analysis of program parts whose grammar is currently unknown or that are irrelevant from a configuration standpoint. For instance, the execution environment is inspected to determine the platform, so that the compiler can be reconfigured into a GNU COBOL or a IBM COBOL variant, discussed later. Moreover, the `CONFIGURATION SECTION` of the `ENVIRONMENT DIVISION` holds information that change the syntax of the language, including separators and currencies. We will delve into these aspects in more detail in Section 5.3. Once the configuration is determined, the language is reconfigured into the Dynamically Generated Variant (Section 5.3).

```
1  DATA DIVISION.
2    WORKING-STORAGE SECTION.
3      01 MY-TABLE OCCURS 5 TIMES INDEXED BY TABLE-INDEX.
4        05 TABLE-VALUE PIC X(10).
5  PROCEDURE DIVISION.
6    MOVE 3 TO TABLE-INDEX.
7    DISPLAY TABLE-VALUE(TABLE-INDEX).
```

Listing 2 Table indexing not compliant to the IBM COBOL variant.

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. HelloWorld.
ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
    SPECIAL-NAMES.
      DECIMAL-POINT IS COMMA.
DATA DIVISION.
  WORKING-STORAGE SECTION.
    01 WS-PRICE PIC 9(4)V99 VALUE 1234,56.
PROCEDURE DIVISION.
  DISPLAY WS-PRICE.
```

Listing 3 Using comma as a decimal separator in COBOL.

5.2. GNU COBOL & IBM COBOL Variants

These two variants are not complete by themselves; instead, they are employed by the Configuration Variant as templates to be filled. While we did not focus on creating a comprehensive collection of COBOL dialects in this work, these two variants are crafted to showcase how a DLPL is particularly well-suited for developing dialects of the same language. On one hand, the modularity offered by language workbenches facilitates code reuse, as we can anticipate that two dialects will share most of their code and differ only in a few specific language features. On the other hand, combining this modular approach with dynamic reconfiguration capabilities allows the variant not to be manually determined by the user. Instead, the actual configuration can be automatically determined based on factors such as the platform or the syntax of the source code. Take Listing 2 as an example. In IBM COBOL, the `INDEXED BY` (Listing 2, line 3) clause is used to define an index variable that can only be used for indexing a table or in specific statements such as the `SET` statement. Conversely, in GNU COBOL, the `INDEXED BY` clause declares a numeric variable that can be used in any statement that is compatible with numeric variables, such as the `MOVE` statement (Listing 2, line 6). Therefore, running Listing 2 on IBM COBOL causes an error on line 6 due to it trying to execute a `MOVE` on an index. Conversely running Listing 2 on GNU COBOL yields the correct output. Based on this knowledge, the program can be analyzed prior to its execution to determine if it needs features specific of either dialect, such as performing a `MOVE` on an index, therefore dynamically adapting the configuration accordingly. This option was not implemented in this work for simplicity. Instead, while the language does not run on mainframe, the Configuration Variant use a configuration file serving as a mock to optionally simulate the execution on a mainframe. While we present only these two variants of COBOL, the same approach can be used to create any number of dialects.

5.3. Dynamically generated variant (COBOL variant)

As mentioned earlier, both the GNU COBOL and IBM COBOL variants are initially incomplete and only become usable after additional configuration is performed at runtime. COBOL programs may include the

```
 1 IDENTIFICATION DIVISION.
 2   PROGRAM-ID. HelloWorld.
 3 ENVIRONMENT DIVISION.
 4   CONFIGURATION SECTION.
 5     SPECIAL-NAMES.
 6       CURRENCY IS "EUR" WITH PICTURE SYMBOL '€'.
 7 DATA DIVISION.
 8   WORKING-STORAGE SECTION.
 9     01 WS-PRICE PIC €9(4)V99 VALUE 1234.56.
10 PROCEDURE DIVISION.
11   DISPLAY WS-PRICE.
```

Listing 4 Using € as a currency in COBOL.

```
 1 IDENTIFICATION DIVISION.
 2   PROGRAM-ID. nestedJavaStatement.
 3 DATA DIVISION.
 4   WORKING-STORAGE SECTION.
 5     01 record.
 6       02 var PIC IS 9(5) COMP-5 OCCURS 5 TIMES VALUE IS 1.
 7 PROCEDURE DIVISION.
 8   DISPLAY record.
 9   JAVA
10     System.out.println(Arrays.toString(record.var));
11     Arrays.fill(record.var, 2);
12   END-JAVA.
13   DISPLAY record.
14   JAVA
15     record = new Record(new int[]{3,3,3,3,3});
16   END-JAVA.
17   DISPLAY record.
```

Listing 5 COBOL program partially modernized into Java.

`ENVIRONMENT DIVISION`, which defines aspects of the program execution, particularly within the `CONFIGURATION SECTION`, as illustrated in Listing 3. On line 6, the user declares that for the rest of the program, a comma will be used instead of the dot as a decimal separator. This change can be observed taking effect on line 9, where the `WS-PRICE` variable is declared and initialized to the value `1234,56`. This feature has two effects on the language implementation. Firstly, the parser remains incomplete until it evaluates the `CONFIGURATION SECTION`, as it is impossible to foresee whether numerical tokens in this specific program will contain a dot or a comma. Secondly, the `ENVIRONMENT DIVISION` becomes obsolete after the initial configuration is performed. Therefore, we can save execution time by performing a dynamic reconfiguration and entirely disregarding this division when the program is actually executed.

Granted, the latter aspect is an implementation detail that could be ignored. Regarding the former, it is arguable that the parser could be written to accept both the dot and the comma as separators, subsequently raising an error if the separator does not coincide with the `ENVIRONMENT DIVISION` declaration. However, we consider such an error to be a syntactic error rather than a semantic one. Moreover, there are more complex situations that require a similar approach. Take Listing 4 as an example. On line 6, the user declares that for the rest of the program, the currency will be `"EUR"`, and the corresponding symbol within picture strings—*i.e.*, in variable declarations—will be €. This change can be observed taking effect on line 9, where the picture string for the `WS-PRICE` variable is declared as `€(4)V99`, indicating a currency comprised of four integer digits, a separator, and finally, two decimal digits. While the decimal separator only allows for two options (dot and comma), the currency can be any symbol, albeit with a few limitations. Therefore, the language implementation is dynamically reconfigured with the correct syntax, avoiding the need for a very complex regular expression to recognize any valid picture string with a currency, followed by a semantic phase to detect errors.

Following this dynamic reconfiguration phase, the final COBOL Variant, corresponding to the Legacy Compiler Variant of Fig. 1, is now complete and capable of executing valid COBOL programs.

Table 2
Match between COBOL and Java primitive data types.

COBOL picture string	Java type
PIC X(m), $m > 1$	String
S9(n) COMP-5, $1 \le n \le 4$	short
S9(n) COMP-5, $5 \le n \le 9$	int
S9(n) COMP-5, $10 \le n \le 18$	long
COMP-1	float
COMP-2	double
COMP-3, PACKED-DECIMAL	java.math.BigDecimal
DISPLAY numeric	java.math.BigDecimal

5.4. *Java variant*

Following the process outlined in Section 4 in a diligent way necessitates the implementation of language features from both the Legacy Compiler Variant and the Modern Compiler Variant. While this can pose a significant overhead, it can be avoided in certain contexts, such as when the Modern Compiler Variant coincides with the execution environment of the language workbench. This comes with the trade-off that these language features will eventually need to be implemented if the system requires further modernization towards a third language. As Neverlang runs on the JVM, we chose to execute modernized (Java) system code on a stock JVM for simplicity. Given this decision, the DLPL is not reconfigured into a Java variant. Instead, when transitioning between COBOL and Java code, the language is initially reconfigured into the Compatibility Variant (see Section 5.5), and then temporarily suspended, giving control to the JVM. After the Java fragment completes its execution, the Compatibility Variant is resumed, updating the COBOL environment to an executable state before reconfiguring back into the COBOL Variant. This reconfiguration is triggered by specific statements added to the language, as discussed in Section 6. For instance, Listing 5 showcases a COBOL program partially modernized into Java: lines 9–12 contain a sequence of Java statements that cannot be executed by the COBOL Variant. Instead, when running the program, these statements are executed by the JVM. However, Java statements within the program have contextual information regarding COBOL variables and their values: they can update the `record` variable declared on line 5, changing the content of each entry from 1 to 2. This change is reflected within COBOL when printing the array on line 13. Similarly, the Java statement on line 15 directly replaces the COBOL record with a Java object, filled with a 3 in each entry this time. Thanks to the data update performed by the Compatibility Variant, both behaviors are valid and do not require awareness of the underlying reconfiguration by either the COBOL Variant, the JVM, or the programmer.

5.5. *Compatibility variant*

The Compatibility Variant corresponds to the Hybrid Compiler Variant in Fig. 1 and acts as a bridge between the COBOL Variant and Java. This outcome is achieved in two phases.

Cobol to java translation. During the first phase, COBOL programs are mapped to Java classes, translating each COBOL variable and record into its Java counterpart. For primitive data types, this information can be derived from the picture string, as summarized in Table 2. However, COBOL records are structures containing additional variables, potentially more records. The Compatibility Variant generates a class for each COBOL record, with the class name matching the record's variable name and its attributes corresponding to the variables within the record. This process is recursively repeated in the case of nested records. As a result, Java statements can access the content of COBOL records simply by using dot notation, as demonstrated on lines 10 and 11 of Listing 5 to retrieve the `var` variable within the record. Meanwhile, each Java fragment within a COBOL program is turned into

```
 1 public interface CobolJavaProgram {
 2   MethodHandles.Lookup lp = MethodHandles.publicLookup();
 3   MethodType mt = MethodType.methodType(void.class);
 4   void updateJava();
 5   void updateCobol();
 6   default Optional<Throwable> invokeJavaStatement(
 7       String methodName) {
 8     try {
 9       MethodHandle javaStatement =
10         lp.findVirtual(this.getClass(), methodName, mt);
11       updateJava();
12       javaStatement.invoke(this);
13       updateCobol();
14       return Optional.empty();
15     } catch (Throwable e) {
16       updateJava();
17       return Optional.of(e);
18     }
19   }
20 }
```

Listing 6 Code handling the reconfiguration from COBOL to Java.

a method that can be executed by the JVM. This initial phase occurs before the program's execution and follows an inspection of its *DATA* DIVISION, leveraging the Java compiler instead of relying on runtime bytecode manipulation.

Runtime data adaptation. The second phase is executed at runtime whenever the system switches between executing COBOL code and Java code. During these transitions, the DLPL is reconfigured into the Compatibility Variant, which updates Java instances when switching from COBOL to Java and vice versa. Moreover, the Compatibility Variant calls the method that was generated for the Java fragment. Additionally, the Compatibility Variant calls the method that was generated for the Java fragment. The interface implemented by the Compatibility Variant to support this behavior is shown in Listing 6. The default method invokeJavaStatement (Listing 6, line 6) handles the transition from COBOL code to Java code and vice versa by invoking the updateJava (line 11) and updateCobol (line 13) methods, respectively. The updateJava (line 4) and updateCobol (line 5) methods do not provide a default implementations and are instead generated during the previous translation step to support the interoperability of all variables contained within the *DATA* DIVISION. Both update methods operate the conversion by representing data contained within arrays and records as a stream of bytes, according to the conversions in Table 2. The actual Java statement (or sequence of statements) is retrieved using reflection (Listing 6, line 9) and then invoked (line 12). Java statements have access to an instance of the calling CobolJavaProgram, from which COBOL variables and their values can be retrieved.

Other compatibility features. In addition to invoking Java statements and blocks, the Compatibility Variant enables the augmentation of the COBOL program with object-oriented elements such as fields, methods, and imports. Most of these features are illustrated in Listing 7. For instance, the *JAVA* DIVISION (lines 7–23) contains code that affects the entire program, comprising *IMPORTS*, *ADD* JAVA blocks, and a *PROGRAM* JAVA block; the *JAVA* DIVISION may also accept parameters through the **USING** keyword (*nameArg*, in example). The *IMPORTS* block (lines 8–10) allows developers to import Java classes for use throughout all Java statements. The *ADD JAVA* block (lines 11–16) is employed to add fields and methods to specific parts of the program. In this case, the user variable declared on line 5 is enhanced with a sayHello method; these changes are then accessible in any Java statement, as demonstrated on line 25. The *PROGRAM* JAVA block (lines 17–22) behaves similarly to the *ADD JAVA* block, but affect the COBOL program itself rather than a variable. Moreover, all COBOL paragraphs—such as *GOODBYE* on line 27—are associated to a Java method. For example, invoking the defaultsection.goodbye method on line 14 causes the DLPL (currently running the Java Variant) to reconfigure first into the Compatibility Variant and then into the COBOL Variant to execute the *GOODBYE* paragraph. Sections and the entire *PROCEDURE* **DIVISION** can be invoked in a similar fashion. Notice how this approach enables parts of the system already rewritten in Java to seamlessly invoke parts not yet modernized using native Java syntax. The implementation supports mutually recursive calls between COBOL code and Java code.

```
 1 IDENTIFICATION DIVISION.
 2   PROGRAM-ID. UserHello.
 3 DATA DIVISION.
 4   WORKING-STORAGE SECTION.
 5     01 user.
 6       05 name PIC X(20).
 7 JAVA DIVISION USING String nameArg.
 8   IMPORTS
 9     import java.util.*;
10   END-IMPORTS.
11   ADD JAVA
12     public void sayHello(){
13       System.out.println("Hello " + name);
14       defaultsection.goodbye();
15     }
16   END-JAVA TO user.
17   PROGRAM JAVA
18     private Person user;
19     public void startProgram(){
20       System.out.println("Start Program");
21       user = new Person(nameArg);
22     }
23   END-JAVA.
24   PROCEDURE DIVISION.
25     JAVA startProgram();  END-JAVA.
26     JAVA user.sayHello(); END-JAVA.
27     GOODBYE.
28       DISPLAY "GOODBYE " user.
```

Listing 7 COBOL program with Java imports, fields, and methods.

Table 3
Versions of the system and evolution steps thereof.

Version	Features	Step	Insertions (LoC)	Deletions (LoC)
1	COBOL system	–	–	–
2	Java main and file access	1	63	34
3	Java user operations	2	49	37
4	Java user class	3	125	58
5	MongoDB server	4	229	136
6	Java system	5	72	91
Total			538	356

6. Demonstration case study

To demonstrate the applicability of the proposed modernization process proposed, we present an evolution scenario from a COBOL program to a mostly Java program, retaining only some data structures from the original system. The application used throughout this section is a concrete implementation of the running example presented in Section 4—*i.e.*, a handler for banking accounts. Although small, the application is complex enough to showcase most of the refactoring aspects of both the COBOL DLPL and the process in general. The complete source code for all system versions of the modernization scenario is available at: https://zenodo.org/doi/10.5281/zenodo.10564325

6.1. Experiment overview

The demonstration case study consists of six system versions and involves five modernization steps, as summarized in Table 3 along with the required changes of each step in terms of lines of code (LoC). The original system is written entirely in COBOL, and each step involves migrating a subset of the application features to Java. Finally, in system version number 6, all relevant application features are rewritten in Java, leaving only some record data structures in COBOL, thus completing the modernization experiment. Most notably, in the original system,

```
 1  Deposit-Withdraw.
 2     DISPLAY "Enter the amount: ".
 3     ACCEPT ins-amount FROM CONSOLE.
 4     <...>
 5     MOVE logged-user-id TO user-id.
 6     MOVE ins-amount TO amount.
 7     <...>
 8     OPEN EXTEND transactions-list.
 9     WRITE transaction.
10     <...>
```

Listing 8 Deposit–Withdraw operations in the original COBOL system.

both login and all other banking operations are handled by reading files from memory. Starting from version 5 onward, the system replaces files with a MongoDB server for persistence. As shown in the last row of Table 3, a total refactoring effort of 538 insertions and 356 deletions was spread across 5 iterations thanks to the approach presented in this work.

6.2. COBOL application

In the original (legacy) system (system version 1 in Table 3), the program reads a list of registered users from a file, searching for a match against the email and password that the user provided on the command line when starting the program. If a match is found, the user is presented with a series of possible operations on their bank account, implemented using an *EVALUATE* block in COBOL. Each operation corresponds to a COBOL paragraph that is executed if the user inputs the corresponding operation ID (1 through 5). In the case of deposit or withdrawal operations, the program preserves the changes by writing the corresponding transaction on a new line in the transactions file, as illustrated in Listing 8 (line 8–9).

6.3. Java main method and file access

The second version of the system (step 1/version 2 in Table 3), resulting from the first modernization step according to the process, replaces the main paragraph, originally written in COBOL, with a Java equivalent among other changes. In this system version, the user operations are still in COBOL and are invoked by the new Java code as if they were Java methods, as previously demonstrated in Listing 7. Only the login operation, shown in Listing 9, is modernized into Java. The original *PERFORM* **UNTIL** block (Listing 6.1, line 4) is replaced by a **while** loop (Listing 6.1, line 4). Similarly, the entire body of the block is rewritten in Java. Notably, the check that the status variable (*users*-list-fs) is set to 10 (Listing 6.1, line 5) to verify that the file has been fully read is replaced by a call to the isUserListEof Java method for a better abstraction.

6.4. Java user operations

In the second modernization step (step 2/version 3 in Table 3), each user operation written in COBOL is refactored into a corresponding Java method. In this version, part of the implementation is still in COBOL, and notably, all the new Java methods are added to the user record of the COBOL itself, rather than creating a dedicated Java class. The migration from the user record written in COBOL to the User class written in Java was divided into two phases: (i) during the second (current) iteration, only methods were created, whereas (ii) the third iteration (further discussed in Section 6.5) led to the creation of a fully-fledged Java class, comprised of both user-related fields and methods.

6.5. Java user class

The third iteration (step 3/version 4 in Table 3) fully replaces the user record written in COBOL with the User Java class, marking the first real progress towards the elimination of legacy COBOL code. As depicted in Listing 10, which showcases a fragment of the entire class, the User class retains a reference to parts of the COBOL program—such as the fields declared on lines 2 and 3 of Listing 10. Through these fields, the User class can reuse segments of the COBOL implementation, for instance, delegating to the store_transaction paragraph (Listing 10, line 8), while migrating all logic and data directly related to the user from COBOL to Java. Conversely, the COBOL program can interact with User instances by adding a User field to the program, using the same syntax shown in a prior example (Listing 7, line 18).

6.6. MongoDB server

This step (step 4/version 5 in Table 3) aims to harness modern Java features in a system that still runs fragments of COBOL code. This is arguably the most significant refactoring step within the context of this case study with regards to the offered application features. This step serves as a demonstration that the approach is compatible with external libraries, as exemplified with MongoDB. In fact, we believe that adopting modern libraries could be one of the primary motivations for migrating from a legacy programming language to a modern one. As finding programmers capable of maintaining legacy systems becomes more challenging, transitioning to modern libraries with large communities and readily available user support becomes an increasingly appealing solution. For instance, COBOL does not support connections to NoSQL servers and thus MongoDB serves as an interesting use case. In this version, the main method is enriched with the boilerplate code needed to connect to the MongoDB instance, that is then passed as an argument to the COBOL program. The evolution is evident upon inspecting the changes shown in Listing 11. While Listing 6.4 still predominantly utilizes COBOL code with some calls to Java methods, Listing 6.4 is mostly Java code querying the NoSQL database and employing modern features, such as the functional interface.

6.7. Java system

As a result of the previous step, the system is predominantly running Java code. However, it still retains portions of COBOL code that have become obsolete, such as user information and transaction handling. The logic for these application features is now part of the Java code. Therefore, the last modernization step (step 5/version 6 in Table 3) focused on removing the corresponding variables within the *WORKING*-STORAGE section, as well as translating any remaining COBOL statements to Java. This version of the program is fully refactored and represents the end of the demonstration case study. The only remaining parts of COBOL code are the user and *transaction* records, which were not manually refactored for simplicity, as their implementation would be identical to the code generated by the Compatibility Variant.

7. Threats to validity

Our demonstration case study, as well as our contribution in general, may be affected by internal and external validity issues. We formulated the modernization approach based on our prior work on open language implementations (Cazzola et al., 2018; Cazzola and Favalli, 2023b). Consequently, some choices may have been influenced by our familiarity with the Neverlang language workbench and its capabilities, introducing both internal and external threats to validity. In terms of internal validity, we strived to define all steps of the process in a language-agnostic manner, attempting to avoid Neverlang-specific concepts. However, we acknowledge the possibility of unintentionally overlooking other options that might not be feasible in Neverlang.

In terms of external validity, the applicability of the approach may be limited by other language workbenches not offering the same capabilities. For instance, to the best of our knowledge no other language workbench offers out-of-the-box runtime adaptability features, making

```cobol
Login.
  OPEN INPUT users-list.

  PERFORM UNTIL is-logged = "Y"
      OR users-list-fs = "10"
    READ users-list
    IF email = ins-email
       AND pass = ins-pass THEN
      MOVE "Y" TO is-logged
    END-IF
  END-PERFORM.

  IF is-logged = "N" THEN
    DISPLAY "Incorrect username or password"
    GOBACK.
    DISPLAY "WELCOME " first-name " " last-name.
```

(a) COBOL implementation.

```
Login.
  OPEN INPUT users-list.
  JAVA
    while(is_logged == ((byte) 'N')
        && !isUserListEof()){
      file_utils.read_user();
      if(user.email.equals(ins_email)
         && user.pass.equals(ins_pass)){
        is_logged = (byte) 'Y';
      }
    }
  END-JAVA.
  IF is-logged = "N" THEN
    DISPLAY "Incorrect username or password"
    GOBACK.
  DISPLAY "WELCOME " first-name " " last-name.
```

(b) Java implementation.

Listing 9 Reimplementation of the login behavior during step 1.

```java
public class User {
  private final List<AccountManagement.Transaction> trs;
  private final AccountManagement accountManagement;
  /* More fields and constructors */
  public void deposit() {
    var transaction = /* prompt user */;
    accountManagement.setTransaction(transaction);
    accountManagement.defaultsection.store_transaction();
    addTransaction(transaction);
  }
  public void addTransaction(
        AccountManagement.Transaction transaction) {
    this.trs.add(transaction);
  }
  /* Other user operations */
}
```

Listing 10 Java class for user operations.

them potentially unsuitable for creating a DLPL. However, as previously mentioned, this limitation to external validity can be mitigated by reviewing the modernization approach: by encompassing re-compilation instead of performing runtime adaptation, the development team can trade-off the system performance in favor of applying the same process with a non-dynamic LPL. The demonstration case study might encounter similar validity issues. Acknowledging the relatively small size of the refactored system, its external applicability may be perceived as limited. However, we attempted to mitigate this concern by properly structuring the case study: we prioritized developing a full-fledged DLPL over a large demonstration case study, favoring variety over quantity of refactoring opportunities. This design choice aims to limit the size of the project in favor of presenting a diverse range of modernization scenarios that developers might encounter in a real-world modernization scenario, such as incorporating the MongoDB external library into a COBOL program.

From an internal validity perspective, the demonstration case study is limited by the subset of language features offered by our current implementation of the COBOL DLPL. This limitation prevented the inclusion of certain application features that could be common in real-world COBOL programs but are incompatible with our subset. However, as discussed in Section 5, our implementation, while still improvable, is extensive and encompasses various features commonly found in imperative programs, including delegation to other COBOL programs. Moreover, it is worth noting that the modernization process, the COBOL DLPL, and the demonstration case study were independently designed by different authors. This intentional separation aims to ensure that the process is not tailored solely to the specifics of this particular case study.

8. Related work

It is important to note that this work is not the first attempt at modernizing legacy software by addressing certain technical aspects to facilitate evolution and maintenance. In the literature, service-oriented architectures stand out as one of the most popular methods to achieve this goal. They involve exposing parts of the legacy system as services that can be utilized by modern software through a technique known as method engineering (Erradi et al., 2006; Khadka et al., 2011; Lewis et al., 2005). Our contribution shares some of its core ideas with these works, as they all aim to reuse components of the original system to enhance interoperability. The modernization of legacy systems into service-oriented architectures often depends on service identification approaches (Abdellatif et al., 2021) to identify functionalities that could be transformed into services. In this context, services share similarities with our application features and their corresponding micro-languages. The key distinction is the fact that service-oriented architectures typically involve distributed applications, whereas in our approach, legacy and modern software coexist within the same runtime environment. Moreover, while service-oriented architectures are mostly communication-driven, our approach is language driven, as the language features enabling the execution of legacy application features are linked to specific variants of the language family.

SPLs have been a topic of discussion with regards to both software modernization (Laguna and Crespo, 2013) and refactoring in general (Ribeiro and Borba, 2008). However, SPLs are known to introduce additional maintainability issues, such as product line erosion (Zhang et al., 2013): as software evolves, variability and relationships among software features also evolve, leading to a misalignment between the variability model and software artifacts. Moreover, the goal is typically extracting a SPL from a legacy system (Martinez et al., 2015; Bertolotti et al., 2023b), extending a SPL with more variants (Marques et al., 2019), or coping with the update of test suite as the software family evolves (do Carmo Machado et al., 2014). To the best of our knowledge, SPLs and, more specifically, LPLs were never used as frameworks to facilitate the evolution of another system.

Language workbenches such ad MPS and Spoofax have been used to modernize legacy software generators (Lillack et al., 2016) and to implement refactoring techniques (Misteli, 2020), although the modernization of software developed by their means is not addressed. Modern applications of language workbenches are varied and go beyond the realm of software modernization. Research topics include (among others): development of distributed (Martínez-Lasaca et al., 2023) and web-based (Warmer and Kleppe, 2022) languages, low-code platforms (Pfeiffer and Wortmann, 2023), optimization problems modeling (Wijesundara, 2023), block-based environments (Verano Merino and van Wijk, 2022), and the derivation of programming languages based on examples (Barash, 2020). More in general, transpilers—*i.e.*, software designed to translate a program from one language to another—form an old and complex research topic. Recent developments include SequalsK (Schultes, 2021), a bidirectional transpiler between Swift and Kotlin, aimed at bridging Android and iOS application development. Similarly, a C to Rust source-to-source

```
JAVA DIVISION.
 <...>
Load-transactions.
   OPEN INPUT transactions-list.
   READ transactions-list.
   PERFORM UNTIL transactions-list-fs = "10"
      JAVA
        user.addTransaction(new Transaction(transaction));
      END-JAVA
      READ transactions-list
   END-PERFORM.
   CLOSE transactions-list.
   <...>
```

(a) File-based implementation.

```
JAVA DIVISION USING
  MongoCollection<AccountManagement.User> users
  MongoCollection<AccountManagement.Transaction> transactions.

  <...>

  PROGRAM JAVA
   <...>
   public void load_transactions(){
     transactions.find(Filters.eq("user_id", user.id))
       .forEach(user::addTransaction);
   }
   <...>
```

(b) MongoDB-based implementation.

Listing 11 Reimplementation of the transactions log during the fourth modernization step.

transpiler was used to migrate legacy code (Ling et al., 2022), while a partial Python to Rust transpiler, demonstrating a 12x performance improvement in the transpiled code (Lunnikivi et al., 2020). These transpilers focus on translating between a single source language and a single target language for modernization, reusability or refactoring purposes, however the same approach may not be applicable to further modernization scenarios. In contrast, the ROSE compiler infrastructure (Quinlan, 2000; Quinlan and Liao, 2011) uses an intermediate representation to support multiple languages, such as C, C++, and Java, whereas Basilisk (Bertolotti et al., 2024b), and ★piler (Bertolotti et al., 2024a) are transpilers that aim to improve the life-span of existing libraries by rendering them available across several languages.

COBOL and legacy systems written in COBOL are frequently cited in discussions about software modernization, particularly towards Java. Among others, contributions encompass case study reports for various contexts, including commercial applications (De Marco et al., 2018) and public administration (Smith and Laszewski, 2010), as well as classifications (Barbier and Recoussine, 2015), integration between COBOL and Java applications (Brune, 2019), reverse engineering techniques (Barbier et al., 2010), and defect location (Ciborowska et al., 2021). Some studies investigate the challenges of modernizing legacy software, particularly when the target of the modernization process is not a system written in COBOL itself but rather one developed using a fourth-generation programming language based on COBOL. While COBOL remains relevant due to continuous updates and modern tooling that make maintenance activities manageable, the same cannot be said for independent compilers (Deknop et al., 2020), often existing in experimental and prototypical states (Zaytsev, 2017). These works investigate challenges and propose solutions related to this issue, including incremental coverage and migration log differencing. In this context, programming languages are acknowledged as valuable tools for assessing the generalizability of engineering techniques to legacy software (Zaytsev, 2020). Their design is investigated to address the challenges of interoperability, aiming to overcome differences in data representation across various languages (van Assen et al., 2023), possibly foreseeing future modernization efforts. While the challenges outlined in these studies show similarities to those we encountered in our work during the creation of the COBOL DLPL, the solutions proposed in those studies employ very different techniques. A compiler-based solution involving language variability and runtime adaptation like the one we proposed shifts the complexity of the modernization process towards the creation of a modular language implementation, offering a high degree of flexibility with regards to the requirements of the modernization attempt. By selectively omitting features from the legacy language and the modern language that do not impact any application features, we significantly reduce the required refactoring effort. Moreover, while the efforts put in the refactoring of the legacy system by other approaches may have limited reusability in subsequent modernization attempts, the language features implemented as part of our proposal can be reused to modernize a different legacy system.

9. Conclusions

The modernization of legacy software systems remains an unsolved challenge, particularly in enterprise environments, where a universal solution is highly needed. In this work we explored a novel approach that tries to address the inherent complexity of this challenge by spreading the development effort required to fully modernize a legacy system across several iterations. Leveraging language workbenches, LPLs, and their dynamic reconfiguration capabilities, our approach provides a refactoring process not bound to any specific technological space, meant to support the interoperability between any pair of programming languages. Through our work and the accompanying demonstration case study, we illustrated how the process benefits from achieving interoperability between languages while allowing for the omission of some features to reduce the development effort. Handling the complexity at language level involves creating a modular compiler, but it serves as a flexible and reusable technique, since existing language features can be repurposed in subsequent modernization scenarios. Our future research directions in this area aim to extend the applicability of this approach beyond a demonstration case study by conducting large-scale modernization attempts. This will involve analyzing how professionals perceive legacy systems and their refactoring (Khadka et al., 2014). Supporting this effort with a post-modernization analysis (Khadka et al., 2015) may reveal unintended benefits and detrimental effects, providing valuable insights to further enhance the modernization process for broader applicability.

CRediT authorship contribution statement

Walter Cazzola: Writing – original draft, Supervision, Project administration, Methodology, Investigation, Funding acquisition, Conceptualization. **Luca Favalli:** Writing – original draft, Software, Methodology, Investigation, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

A replication package is available on Zenodo and linked in the paper.

References

Abdellatif, M., Shatnawi, A., Mili, H., Moha, N., El Boussaidi, G., Hecht, G., Privat, J., Guéhéneuc, Y.G., 2021. A taxonomy of service identification approaches for legacy software systems modernization. J. Syst. Softw. 173.

Aho, A.V., Sethi, R., Ullman, J.D., 1986. Compilers: Principles, Techniques, and Tools. Addison Wesley, Reading, Massachusetts.

Ali, M., Hussain, S., Ashraf, M., Paracha, M.K., 2020. Addressing software related issues on legacy systems—A review. Int. J. Scient. Technol. Res. 9, 3738–3742.

Apel, S., von Thein, A., Wendler, P., Größlinger, A., Beyer, F., 2013. Strategies for product-line verification: Case studies and experiments. In: Chang, B.H., Pohl, K. (Eds.), Proceedings of the 35th International Conference on Software Engineering. ICSE'13, IEEE, San Francisco, CA, USA, pp. 482–491.

van Assen, M., Ntagengerwa, M.A., Saylir, O., Zaytsev, V., 2023. Crossover: Towards compiler-enabled COBOL-C interoperability. In: Shaikhha, A. (Ed.), Proceedings of the 22nd International Conference on Generative Programming: Concepts and Experiences. GPCE'23, ACM, Cascais, Portugal, pp. 72–85.

Barash, M., 2020. Example-driven software language engineering. In: Tratt, L., de Lara, J. (Eds.), Proceedings of the 13th International Conference on Software Language Engineering. SLE'20, ACM, Chicago, IL, USA, pp. 246–252.

Barbier, F., Eveillard, S., Youbi, K., Guitton, O., Perrier, A., Cariou, E., 2010. Model-driven reverse engineering of COBOL-based applications. In: Ulrich, W.M., Newcomb, P.H. (Eds.), Information Systems Transformation. The MK/OMG Press, pp. 283–299, chapter 11.

Barbier, F., Recoussine, J.L., 2015. Software modernization: Technical environment. In: COBOL Software Modernization: From Principles to Implementation with Blu Age® Method. John Wiley & Sons, Inc., pp. 21–38, chapter 2.

Barros, D., Peldszus, S., Assunção, W.K.G., Berger, T., 2022. Editing support for software languages: Implementation practices in language server protocols. In: Wimmer, M. (Ed.), Proceedings of the 25th International Conference on Model Driven Engineering Langauges and Systems. MoDELS'22, ACM, Montréal, Canada, pp. 232–243.

Bélády, L.A., Lehman, M.M., 1976. A model of large program development. IBM Syst. J. 15, 225–252.

Bertolotti, F., Cazzola, W., Favalli, L., 2023a. On the granularity of linguistic reuse. J. Syst. Softw. 202, http://dx.doi.org/10.1016/j.jss.2023.111704.

Bertolotti, F., Cazzola, W., Favalli, L., 2023b. SPLLS: Software product lines extraction driven by language server protocol. J. Syst. Softw. 205, http://dx.doi.org/10.1016/j.jss.2023.111809.

Bertolotti, F., Cazzola, W., Favalli, L., 2024a. ★Piler: Compilers in search of compilations. J. Syst. Softw. 212, http://dx.doi.org/10.1016/j.jss.2024.112006.

Bertolotti, F., Cazzola, W., Ostuni, D., Castoldi, C., 2024b. When the dragons defeat the knight: Basilisk an architectural pattern for platform and language independent development. J. Syst. Softw. 215, http://dx.doi.org/10.1016/j.jss.2024.112088.

van den Brand, M., 2023. A personal retrospective on language workbenches. Softw. Syst. Model. 22, 847–850.

Brune, P., 2019. An open source approach for modernizing message-processing and transactional COBOL applications by integration in java EE application servers. In: Escalona, M.J., Mayo, F.D., Majchrzak, T.A., Monfort, V. (Eds.), Proceedings of the 14th International Conference on Web Information Systems and Technologies. WEBIST'18, Springer, Seville, Spain, pp. 244–261.

Capilla, R., Bosch, J., Trinidad, P., Ruiz-Cortéz, A., Hinchey, M., 2014. An overview of dynamic software product line architectures and techniques: Observations from research and industry. J. Syst. Softw. 81, 3–23.

do Carmo Machado, I., McGregor, J.D., Cavalcanti, Y.a.C., de Almeida, Eduardo Santana, 2014. On strategies for testing software product lines: A systematic literature review. Inf. Softw. Technol. 56, 1183–1199.

Cazzola, W., 2012. Domain-specific languages in few steps: The Neverlang approach. In: Gschwind, T., De Paoli, F., Gruhn, V., Book, M. (Eds.), Proceedings of the 11th International Conference on Software Composition. SC'12, Springer, Prague, Czech Republic, pp. 162–177.

Cazzola, W., Chitchyan, R., Rashid, A., Shaqiri, A., 2018. μ-DSU: A micro-language based approach to dynamic software updating. Comput. Lang. Syst. Struct. 51, 71–89. http://dx.doi.org/10.1016/j.cl.2017.07.003.

Cazzola, W., Favalli, L., 2022. Towards a recipe for language decomposition: Quality assessment of language product lines. Empir. Softw. Eng. 27, http://dx.doi.org/10.1007/s10664-021-10074-6.

Cazzola, W., Favalli, L., 2023a. Scrambled features for breakfast: Concepts of agile language development. Commun. ACM 66, 50–60. http://dx.doi.org/10.1145/3596217.

Cazzola, W., Favalli, L., 2023b. The language mutation problem: Leveraging language product lines for mutation testing of interpreters. J. Syst. Softw. 195, http://dx.doi.org/10.1016/j.jss.2022.111533.

Cazzola, W., Olivares, D.M., 2016. Gradually learning programming supported by a growable programming language. IEEE Trans. Emerg. Top. Comput. 4, 404–415. http://dx.doi.org/10.1109/TETC.2015.2446192, special Issue on Emerging Trends in Education.

Cazzola, W., Poletti, D., 2010. DSL evolution through composition. In: Proceedings of the 7th ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution. RAM-SE'10, ACM, Maribor, Slovenia.

Cazzola, W., Shaqiri, A., 2017. Open programming language interpreters. Art. Sci. Eng. Program. J. 1, 5–1–5–34. http://dx.doi.org/10.22152/programming-journal.org/2017/1/5.

Cazzola, W., Vacchi, E., 2013. Neverlang 2: Componentised language development for the JVM. In: Binder, W., Bodden, E., Löwe, W. (Eds.), Proceedings of the 12th International Conference on Software Composition. SC'13, Springer, Budapest, Hungary, pp. 17–32.

Cazzola, W., Vacchi, E., 2014. On the incremental growth and shrinkage of LR goto-graphs. Acta Inform. 51, 419–447. http://dx.doi.org/10.1007/s00236-014-0201-2.

Chisnall, D., 2013. The challenge of cross-language interoperability. Commun. ACM 56, 50–56.

Chitchyan, R., Cazzola, W., Rashid, A., 2015. Engineering sustainability through language. In: Proceedings of the 37th International Conference on Software Engineering. ICSE'15, IEEE, Firenze, Italy, pp. 501–504, Track on Software Engineering in Society.

Ciborowska, A., Chakarov, A., Pandita, R., 2021. Contemporary COBOL: Developers' perspectives on defects and defect location. In: Andersson, B., Almeida, L., Hsieh, J.W. (Eds.), Proceedings of the 37th International Conference on Software Maintenance and Evolution. ICSME'21, IEEE, Luxembourg City, Luxembourg, pp. 227–238.

Crane, M.L., Dingel, J., 2005. UML vs. Classical vs. Rhapsody statecharts: Not all models are created equal. In: Briand, C. (Ed.), Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems. MoDELS'05, Springer, Montego Bay, Jamaica, pp. 97–112.

De Marco, A., Iancu, V., Asinofsky, I., 2018. COBOL to java and newspapers still get delivered. In: Khomh, F., Lo, D. (Eds.), Proceedings of the 34th International Conference on Software Maintenance and Evolution. ICSME'18, IEEE, Madrid, Spain, pp. 583–586.

Deknop, C., Fabry, J., Mens, K., Zaytsev, V., 2020. Improving a software modernisation process by differencing migration logs. In: Morisio, M., Torchiano, M., Jedlitschka, A. (Eds.), Proceedings of the 21st International Conference on Product-Focused Software Process Improvement. PROFES'20, Springer, Turin, Italy, pp. 270–286.

Erdweg, S., Giarrusso, P.G., Rendel, T., 2012. Language composition untangled. In: Sloane, A.M., Andova, S. (Eds.), Proceedings of the 12th Workshop on Language Description, Tools, and Applications. LDTA'12, ACM, Tallinn, Estonia.

Erdweg, S., van der Storm, T., Völter, M., Tratt, L., Bosman, R., Cook, W.R., Gerritsen, A., Hulshout, A., Kelly, A., Konat, G., Molina, P.J., Palatnik, M., Pohjonen, R., Schindler, E., Schindler, K., Solmi, R., Vergu, V., Visser, E., van der Vlist, K., Wachsmuth, G., van der Woning, J., 2015. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. Comput. Lang. Syst. Struct. 44, 24–47.

Erradi, A., Anand, S., Kulkarni, N., 2006. Evaluation of strategies for integrating legacy applications as services in a service oriented architecture. In: Zhao, J.L., Blake, M.B. (Eds.), Proceedings of the 3rd International Conference on Services Computing. SCC'06, IEEE, Chicago, IL, USA, pp. 257–260.

Fowler, M., 2005. Language workbenches: The killer-app for domain specific languages? Martin Fowler's Blog. http://www.martinfowler.com/articles/languageWorkbench.html.

Fowler, M., Parsons, R., 2010. Domain Specific Languages. Addison Wesley.

Ghosh, D., 2011. DSL for the uninitiated. ACM Queue Mag. 9, 1–11.

Hallsteinsen, S., Hinchey, M., Park, S., Schmid, K., 2008. Dynamic software product lines. IEEE Comput. 41, 93–95.

Hinchey, M., Park, S., Schmid, K., 2012. Building dynamic software product lines. IEEE Comput. 45, 22–26.

Hirzel, M., Grimm, R., 2007. Jeannie: Granting java native interface developers their wishes. In: Bacon, D.F., Videira-Lopes, C., Steele, G.L. (Eds.), Proceedings of the 22nd Annual Conference on Object-Oriented Programming Systems, Languages, and Applications. OOPSLA'07, ACM, Montréal, Québec, Canada, pp. 19–38.

Khadka, R., Batlajery, B.V., Saeidi, A.M., Jansen, S., Hage, J., 2014. How do professionals perceive legacy systems and software modernization>. In: Briand, L., van der Hoek, A. (Eds.), Proceedings of the 36th International Conference on Software Engineering. ICSE'14, IEEE, Hyderabad, India, pp. 36–47.

Khadka, R., Reijnders, G., Saeidi, A., Jansen, S., Hage, J., 2011. A method engineering based legacy to SOA migration method. In: Cordy, J.R., Tonella, P. (Eds.), Proceedings of the 27th IEEE International Conference on Software Maintenance. ICSM'11, IEEE, Williamsburg, VA, USA, pp. 163–172.

Khadka, R., Shrestha, P., Klein, B., Saeidi, A., Hage, J., Jansen, S., van Dis, E., Bruntink, M., 2015. Does software modernization deliver what it aimed for? A post modernization analysis of five software modernization case studies. In: Krinke, J., Robillard, M. (Eds.), Proceedings of the 31st International Conference on Software Maintenance and Evolution. ICSME'15, IEEE, Bremen, Germany, pp. 477–486.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, B., 2001. An overview of AspectJ. In: Knudsen, J.L. (Ed.), Proceedings of the 15th European Conference on Object-Oriented Programming. ECOOP'01, Springer-Verlag, Budapest, Hungary, pp. 327–353.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Videir Lopes, C., Loingtier, J.M., Irwin, J., 1997. Aspect-oriented programming. In: Akşit, M., Matsuoka, S. (Eds.), 11th European Conference on Object Oriented Programming. ECOOP'97, Springer-Verlag, Helsinki, Finland, pp. 220–242.

Kühn, T., Cazzola, W., 2016. Apples and oranges: Comparing top-down and bottom-up language product lines. In: Rabiser, R., Xie, B. (Eds.), Proceedings of the 20th International Software Product Line Conference. SPLC'16, ACM, Beijing, China, pp. 50–59.

Kühn, T., Cazzola, W., Olivares, D.M., 2015. Choosy and picky: Configuration of language product lines. In: Botterweck, G., White, J. (Eds.), Proceedings of the 19th International Software Product Line Conference. SPLC'15, ACM, Nashville, TN, USA, pp. 71–80.

Kühn, T., Leuthäuser, M., Götz, S., Seidl, C., Aßmann, U., 2014. A metamodel family for role-based modeling and programming languages. In: Combemale, B., Pearce, D.J., Barais, O., Vinju, J. (Eds.), Proceedings of the 7th International Conference Software Language Engineering. SLE'14, Springer, Västerås, Sweden, pp. 141–160.

Laguna, M.A., Crespo, Y., 2013. A systematic mapping study on software product line evolution: From legacy system reengineering to product line refactoring. Sci. Comput. Program. 78, 1010–1034.

Leduc, M., Degueule, T., Van Wyk, E., Combemale, B., 2020. The software language extension problem. Softw. Syst. Model. 19, 263–267.

Lee, D., Henley, A.Z., Hinshaw, B., Pandita, R., 2022. Opencbs: An open-source COBOL defects benchmark suite. In: Avgeriou, P., Binkley, D. (Eds.), Proceedings of the 38th International Conference on Software Maintenance and Evolution. ICSME'22, IEEE, Limassol, Cyprus, pp. 246–256.

Lewis, G., Morris, E., Smith, D., O'Brien, L., 2005. Service-oriented migration and reuse technique (SMART). In: Zou, Y., Di Penta, M. (Eds.), Proceedings of the 13th IEEE International Workshop on Software Technology and Engineering Practice. STEP'05, IEEE, Budapest, Hungary, pp. 222–229.

Liebig, J., Daniel, R., Apel, S., 2013. Feature-oriented language families: A case study. In: Collet, P., Schmid, K. (Eds.), Proceedings of the 7th International Workshop on Variability Modelling of Software-Intensive Systems. VaMoS'13, ACM, Pisa, Italy.

Lillack, M., Berger, T., Hebig, R., 2016. Experiences from reengineering and modularizing a legacy software generator with a projectional language workbench. In: Rabiser, R., Xie, B. (Eds.), Proceedings of the 20th International Systems and Software Product-Line Conference. SPLC'16, ACM, Beijing, China, pp. 346–353.

Ling, M., Yu, Y., Wu, H., Wang, Y., Cordy, J.R., Hassan, A.E., 2022. In rust we trust: A transpiler from unsafe C to safer rust. In: Companion Proceedings of the 44th International Conference on Software Engineering. ICSE'22-Companion, IEEE, Pittsburgh, PA, USA, pp. 354–355.

Lunnikivi, H., Jylkkä, K., Hämäläinen, T., 2020. Transpiling python to rust for optimized performance. In: Orailoglu, A., Jung, M., Reichenbach, M. (Eds.), Proceedings of the 20th International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation. SAMOS'20, Springer, Samos, Greece, pp. 127–138.

Marques, M., Simmonds, J., Rossel, P.O., Bastarrica, M.C., 2019. Software product line evolution: A systematic literature review. Inf. Softw. Technol. 105, 190–208.

Martinez, J., Ziadi, T., Bissyandé, T.F., Klein, J., le Traon, Y., 2015. Automating the extraction of model-based software product lines from model variants. In: Cohen, M., Grunske, L., Whalen, M. (Eds.), Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering. ASE'15, IEEE, Lincoln, NE, USA, pp. 396–406.

Martínez-Lasaca, F., Díez, P., Guerra, E., de Lara, J., 2023. Dandelion: A scalable, cloud-based graphical language workbench for industrial low-code development. J. Comput. Lang. 76.

Misteli, P.D., 2020. Towards language-parametric refactorings. In: Student Research Competition at the 4th International Conference on Art, Science and Engineering of Programming. SRC@Programming'20, ACM, Porto, Portugal, pp. 213–214.

Pech, V., 2021. JetBrains MPS: Why modern language workbenches matter. In: Bucchiarone, A., Cicchetti, A., Ciccozzi, F., Pierantonio, A. (Eds.), Domain-Specific Languages in Practice. Springer, pp. 1–21, chapter 1.

Pfeiffer, J., 2023. Systematic component-oriented language reuse. In: Proceedings of the Doctoral Symposium at the 26th International Conference on Model Driven Engineering Langauges and Systems. PhD-MoDELS'23, IEEE, Västerås, Sweden, pp. 166–171.

Pfeiffer, J., Wortmann, A., 2023. A low-code platform for systematic component-oriented language composition. In: Degueuele, T., Scott, E. (Eds.), Proceedings of the 16th International Conference on Software Language Engineering. SLE'23, ACM, Cascais, Portugal, pp. 208–213.

Quinlan, D., 2000. ROSE: Compiler support for object-oriented frameworks. Parallel Process. Lett. 10, 215–226.

Quinlan, D., Liao, C., 2011. The rose source-to-source compiler infrastructure. In: Midkiff, S., Eigenmann, R., Bae, H. (Eds.), Proceedings of the Cetus Users and Compiler Infrastructure Workshop. Galveston, TX, USA, pp. 1–3.

Ribeiro, M., Borba, P., 2008. Recommending refactorings when restructuring variabilities in software product lines. In: Dig, D., Fuhrer, R.M., Johnson, R. (Eds.), Proceedings of the 2nd Workshop on Refactoring Tools. WRT'08, ACM, Nashville, TN, USA, pp. 8:1–8:4.

Schultes, D., 2021. Sequalsk—A bidirectional swift-kotlin-transpiler. In: Abreu, R., Fazzini, M. (Eds.), Proceedings of the 8th International Conference on Mobile Software Engineering and Systems. MobileSoft'21, IEEE, Madrid, Spain, pp. 73–83.

Smith, M.K., Laszewski, T., 2010. Modernization case study: Italian ministry of instruction, university, and research. In: Ulrich, W.M., Newcomb, P.H. (Eds.), Information Systems Transformation. The MK/OMG Press, pp. 171–191, chapter 7.

Tanter, E., 2009. Reflection and Open Implementation. Technical Report TR-DCC-20091123-013, DCC, University of Chile.

Terekhov, A.A., Verhoef, C., 2000. The realities of language conversions. IEEE Softw. 17, 111–124.

Tharp, A.L., 1984. The impact of fourth generation programming languages. ACM SIGCSE Bull. 16, 37–44.

Tratt, L., 2008. Domain specific language implementation via compile-time meta-programming. ACM Trans. Program. Lang. Syst. 30, 31:1–31:40.

Uddin, G., Alam, O., Serebrenik, A., 2022. A qualitative study of developers' discussions of their problems and joys during the early COVID-19 months. Empir. Softw. Eng. 27.

Vacchi, E., Cazzola, W., 2015. Neverlang: A framework for feature-oriented language development. Comput. Lang. Syst. Struct. 43, 1–40. http://dx.doi.org/10.1016/j.cl.2015.02.001.

Vacchi, E., Cazzola, W., Combemale, B., Acher, M., 2014. Automating variability model inference for component-based language implementations. In: Heymans, P., Rubin, J. (Eds.), Proceedings of the 18th International Software Product Line Conference. SPLC'14, ACM, Florence, Italy, pp. 167–176.

Vacchi, E., Cazzola, W., Pillay, S., Combemale, B., 2013. Variability support in domain-specific language development. In: Erwig, M., Paige, R.F., Van Wyk, E. (Eds.), Proceedings of 6th International Conference on Software Language Engineering. SLE'13, Springer, Indianapolis, USA, pp. 76–95.

Verano Merino, M., van Wijk, K., 2022. Workbench for creating block-based environments. In: Burgueño, L., Cazzola, W. (Eds.), Proceedings of the 15th International Conference on Software Language Engineering. SLE'22, ACM, Aukland, New Zealand, pp. 61–73.

Voelter, M., 2013. DSL Engineering. CreateSpace Independent Publishing.

Šipek, M., Mihaljević, B., Radovan, A., 2019. Exploring aspects of polyglot high-performance virtual machine graalvm. In: Skala, K. (Ed.), Proceedings of the 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics. MIPRO'19, IEEE, Opatija, Croatia, pp. 1671–1676.

Vu, H.C., Do, P., Barros, A., Bérenguer, C., 2014. Maintenance grouping strategy for multi-component systems with dynamic contexts. Reliab. Eng. Syst. Saf. 132, 233–249.

Ward, M.P., 1994. Language oriented programming. Softw. Concept Tools 15, 147–161.

Warmer, J., Kleppe, A., 2022. Freon: An open web native language workbench. In: Burgueño, W. (Ed.), Proceedings of the 15th International Conference on Software Language Engineering. SLE'22, ACM, Aukland, New Zealand, pp. 30–35a.

Wijesundara, S.S., 2023. Domain specific languages for optimisation modelling. In: Proceedings of the ACM Student Research Competition at the 45th International Conference on Software Engineering. SRC@ICSE'23, IEEE, Melbourne, Australia, pp. 299–301.

Zaytsev, V., 2017. Open challenges in incremental coverage of legacy software languages. In: Church, L., Gabriel, R.P., Hirschfeld, R., Masuhara, H. (Eds.), Proceedings of the 3rd International Workshop on Programming Experience. PX'17, ACM, Vancouver, BC, Canada, pp. 1–6.

Zaytsev, V., 2020. Software language engineers' worst nightmare. In: Tratt, L., de Lara, J. (Eds.), Proceedings of the 13th International Conference on Software Language Engineering. SLE'20, ACM, Virtual, USA, pp. 72–85.

Zaytsev, V., Fabry, J., 2019. Fourth-generation languages are technical debt. In: Avgeriou, P., Schmid, K. (Eds.), Proceedings of the 2nd Technical Debt Conference. TechDebt'19, Montreal, QC, Canada, p. 1.

Zhang, B., Becker, M., Patzke, T., Sierszecki, K., Savolainen, J.E., 2013. Variability evolution and erosion in industrial product lines: A case study. In: Jarzabek, S., Gnesi, S. (Eds.), Proceedings of the 17th International Software Product Line Conference. SPLC'17, ACM, Tokyo, Japan, pp. 168–177.

Walter Cazzola is currently a Full Professor in the Department of Computer Science of the Università degli Studi di Milano, Italy and the Chair of the ADAPT laboratory. Dr. Cazzola designed the mChaRM framework,@Java, [a]C#, Blueprint programming languages and he is currently involved in the designing and development of the Neverlang language workbench. He also designed the JavAdaptor dynamic software updating framework and its front-end FiGA. He has written over 100 scientific papers. His research interests include (but are not limited to) software maintenance, evolution and comprehension, programming methodologies and languages. He served on the program committees or editorial boards of the most important conferences and journals about his research topics. He is associate editor for the Journal of Computer Languages published by Elsevier. More information about Dr. Cazzola and all his publications are available at http://cazzola.di.unimi.it and he can be contacted at cazzola@di.unimi.it for any question.

Luca Favalli is currently a Computer Science Postdoctoral Researcher at Università degli Studi di Milano. He got his Ph.D. in computer science from the Università degli Studi di Milano. He is involved in the research activity of the ADAPT Lab and in the development of the Neverlang language workbench and of JavAdaptor. His main research interests are software design, software (and language) product lines and dynamic software updating with a focus on how they can be used to ease the learning of programming languages. He can be contacted at favalli@di.unimi.it for any question.