On Polyglot Program Testing

Philémon Houdaille, Djamel Eddine Khelladi, Benoît Combemale, Gunter Mussbacher

On Polyglot Program Testing

Philémon Houdaille
Djamel Eddine Khelladi
CNRS - Univ. Rennes - IRISA - INRIA
Rennes, France
{name.surname}@irisa.fr

Benoit Combemale
Univ. Rennes - IRISA - CNRS - INRIA
Rennes, France
benoit.combemale@irisa.fr

Gunter Mussbacher
McGill University / INRIA
Montréal / Rennes, Canada / France
gunter.mussbacher@mcgill.ca

ABSTRACT

In modern applications, it has become increasingly necessary to use multiple languages in a coordinated way to deal with the complexity and diversity of concerns encountered during development. This practice is known as polyglot programming. However, while execution platforms for polyglot programs are increasingly mature, there is a lack of support in how to test polyglot programs. This paper is a first step to increase awareness about polyglot testing efforts. It provides an overview of how polyglot programs are constructed, and an analysis of the impact on test writing at its different steps. More specifically, we focus on dynamic white box testing, and how polyglot programming impacts selection of input data, scenario specification and execution, and oracle expression. We discuss the related challenges in particular with regards to the current state of the practice. We envision in this paper to raise interest in polyglot program testing within the software engineering community, and help in defining directions for future work.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; **Interoperability**; Software development techniques.

KEYWORDS

polyglot programming, white box testing

1 INTRODUCTION

As complexity of software increases, developers have to face more and more challenges when developing applications. One of these challenges is the diversity of concerns encountered: a single program might present a variety of different concerns to achieve its end goal, such as machine learning algorithms, network communications, video encoding, user interface, *etc.*

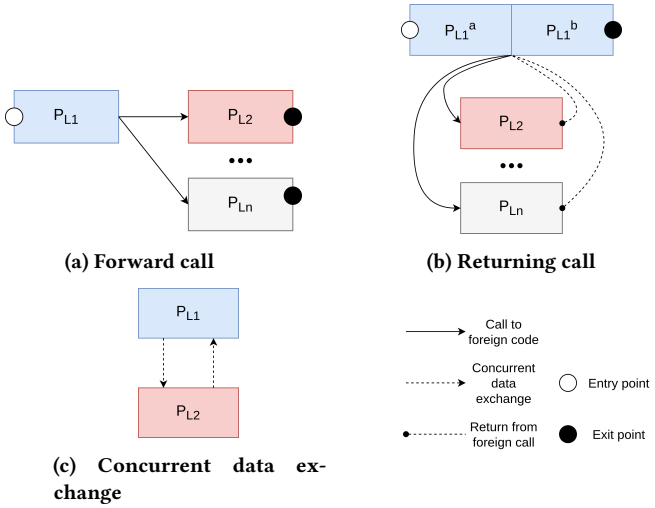
One solution to help developers manage this diversity of concerns is the use of multiple programming languages in conjunction, which has been defined as polyglot programming [27]. Different concerns are not always best answered in the same manner, and one factor determining the ease of implementation is the programming language being used. For example, Python is well suited for data management due to its library ecosystem, while high performance video encoding would be better realized in a language with lower-level control of memory such as C. Polyglot programming exploits this concept with the idea that for the best possible language to be used in answer to a specific concern, the part of the program written in this language needs to be able to freely interact with parts written in other languages that will solve different problems in the same program. In this paper, we use the term polyglot platform to refer to an environment where parts of a program written in multiple languages are able to execute and interoperate, exchange data, or pass messages. We henceforth name these parts written in different languages sub-programs. It is worth noting that sub-programs are not micro-services: the definition of polyglotism [27] we use focuses on the programmer's interaction with multiple languages, which does not happen in a micro-service based architecture where the interaction with another service (presumably written in another language) is separated by a well defined API, and different services are often handled by different development teams. Polyglotism could still occur within a single micro-service.

Examples of polyglot platforms include MetaCall [4], Foreign Function Interfaces [3] (FFI), GraalVM [37] and its Truffle [36] language framework, or WebAssembly [13]. However, the scope of this work focuses on platforms that allow some notion of data exchange between languages, allowing to naturally share variables within the program. For instance, MetaCall and most FFIs focus only on making functions available across languages, and thus, would not be included in our scope. On the other hand, WebAssembly modules let programmers call functions but also directly access memory from programs written in other languages, while GraalVM has dynamic code evaluation and a global memory space shared between sub-programs. We focus on such platforms that support both language mixing, *i.e.*, the ability to call code from a foreign language, and data sharing, *i.e.*, the ability to naturally exchange data between languages. The underlying implementation of the execution platform is irrelevant so long as these operations are supported.

While efforts towards high quality polyglot execution platforms are promising and gaining attraction [10, 28–31, 33], we argue that development tools of the same quality as non-polyglot programming also need to be investigated. In particular, tools to write tests are crucial for any software release and to the best of our knowledge have not yet been envisioned for the specific case of polyglot programs. As such, this paper represents the first step and a vision towards the establishment of tools to better test polyglot programs, and an attempt to anticipate problems that may arise from this practice.

More specifically, we explore polyglot program testing in the context of dynamic testing where *input data*, *scenario* specification and execution, and the *oracle* are the main steps. We focus on white box testing with unit and integration testing, where the test writer has knowledge of the coordination of sub-programs, in contrast to system testing where the system is considered as black box.

Our contributions are as follows. We first propose an appropriate decomposition of polyglot programs in Section 2, describing the base structure cases and their combination. We then provide an analysis of the impact of each polyglot structure case on every step



(a) Forward call **(b) Returning call**

(c) Concurrent data exchange

Figure 1: Polyglot program structure cases

of writing tests in Section 3. Section 4 briefly reviews related work and Section 5 concludes the paper and provides perspectives.

2 STRUCTURE OF POLYGLOT PROGRAMS

This section provides an overview of how polyglot programs are structured. Section 2.1 describes the base cases. Section 2.2 then discusses how they can be composed.

2.1 Base cases

This section will discuss the possible scenarios in which polyglot sub-programs can be mixed and interoperate as a whole.

Fundamentally, programs can be coordinated either through control-flow or data-flow. Control-flow can be expressed with a variety of means, such as function calls, conditionals, loops, jumps. Data-flow is often controlled through variables, and with concepts, such as visibility or scopes. We consider the case where the polyglot platform is able to both mix code and exchange data between sub-programs. In other words, we consider two primitives: execution of foreign code as if calling a function, and exchange of data between parts of the program written in different languages. The former serves as the control-flow polyglot primitive, while the latter is a data-flow polyglot primitive. In our scope, the execution of foreign code is akin to a function call, meaning there is also implicit polyglot data-flow in the form of arguments (from the caller to the callee) and return value (from the callee to the caller).

For instance, WebAssembly uses the concept of modules written in any language. These modules can export functions and data. One can then use the WebAssembly API in another language to access these modules and their exports: this lets the programmer both call code (through exported functions) and directly manipulate exported data. The original WebAssembly module inclusion statement can be seen as a code execution that only declares all of the module's exports. With GraalVM and Truffle, the same two primitives are more clearly split into two constructs: the `polyglot.eval` function, and the `polyglot.export` and `polyglot.import` twin functions. These respectively let programs evaluate code from another language, and manipulate a global polyglot memory space.

Based on these two types of primitives available, we can derive the possible patterns of sub-program coordination. We name these structure cases and show a visual representation in Figure 1.

The first case shown in Figure 1a depicts the basic use of the code execution primitive. A given sub-program P_{L1} ends its execution by calling code written in another language L2, and this sub-program P_{L2} continues the execution. There is no data returned from the foreign called code in this case, meaning P_{L1} never interacts with data from P_{L2} (but P_{L2} may manipulate P_{L1} data given through function parameters). When concurrency is supported, P_{L1} can spawn threads for any number of sub-programs, represented by the P_{Ln} box. On the diagram, a white circle represents the entry point of the structure case, while a black circle represents its exit point.

In the event where the foreign code call does return a value as in Figure 1b, the program written in L1 can be split into two parts: before the call ${P_{L1}}^a$, and after the call ${P_{L1}}^b$ where the return value of the foreign code call may be used. As with the previous structure case, this can appear with any number of threads.

Finally, while all foreign code calls are implicit data exchanges, the data exchange primitives can also be used between concurrent threads. This translates to Figure 1c, where two concurrent threads from different sub-programs have no calls to each other, but still exchange data. While different execution platforms may have different concepts of scope and visibility, we simplify by assuming a single global space, where any thread can both use data from and send data to any other threads. How the threads synchronize these concurrent data exchanges depends on the languages and their available synchronization primitives. This last case does not have entry and exit points, and cannot be instantiated on its own; it requires another structure case that spawns at least two threads as shown in Figure 1a and Figure 1b with the P_{Ln} box.

2.2 Composability

Each structure of Figure 1 is a minimal polyglot program that we can reason on. Any other more complex polyglot program construction is simply a composition of the three identified base cases. We intuit that properties over these three base cases still hold when they are composed to form more complex programs if the following rules are enforced during composition.

The first rule to compose polyglot programs is that if any two threads of different sub-programs are running concurrently, they may exchange data as shown in Figure 1c. The second rule states that any box on the diagram can be replaced with any base structure case or composition of structure cases, provided that the replacement has a single entry point and at least one exit point. Note that ${P_{L1}}^a$ and ${P_{L1}}^b$ are considered as two distinct boxes for this rule.

Figure 2 gives a composition example for a program with two concurrent threads and five different languages. We first start from the base structure case of Figure 1a, where P_{L1} creates two threads in different languages: P_{L2} and P_{L3}. Then, we can transform the P_{L3} box into an instance of the structure case of Figure 1b (by applying the second rule), with ${P_{L3}}^a$, ${P_{L3}}^b$, and a matching new sub-program P_{L4}. We can then specialize the box P_{L4} into another instance of Figure 1a with P_{L4} and P_{L5} (by applying the second rule again). Finally, because P_{L2} and P_{L3} are concurrent threads, we can specify that P_{L2} and ${P_{L3}}^b$ exchange some data (by applying the first rule).

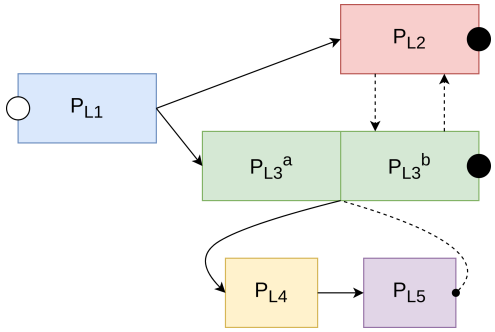


Figure 2: Example of a composition of base structure cases

We argue that while quality unit testing of the sub-programs is a good starting point for a quality test suite covering the whole polyglot program, quality testing of the occurrences of structure cases appearing in the program is also needed. The next section discusses various impacts and challenges that may arise from the presence of these structure cases.

3 IMPACT OF POLYGLOT SITUATIONS ON TEST WRITING

This section describes the impact that polyglot programming usage may have on the task of writing tests. We focus on the three major parts of dynamic, white-box testing, known as 1) *input data* selection, 2) *scenario* specification and execution, and 3) *oracle* expression. We discuss differences with standard practices and the challenges that may arise when testing a polyglot system.

3.1 Input data selection

In dynamic testing, input data selection refers to the problem of selecting inputs for the program under test. In white box testing, input data is often selected with the goal to improve test suite metrics (*e.g.*, code coverage). When considering polyglot programs, there are a few ways input data selection can be impacted.

First, when polyglot code is being tested, there may be multiple levels of sub-programs being nested in calls. This means the test writer needs to select input data that is relevant to all sub-programs of each language (*e.g.*, in case of Figure 1b, testing P_{L1} may imply testing P_{L2}). In the case of manual test writing, this requires the test writer to be proficient in understanding each programming language present in the system. This is different from the implementation of polyglot code, where details of the nested sub-programs may be hidden behind well-defined interfaces. If the input data selection is automated or semi-automated through white box analysis, the analysis tool needs to be adapted to handling polyglot code.

The construction of input data can also be impacted. Suppose a case similar to Figure 1b where P_{L2} is reusable code called many times in the program. One may want to write tests for this code in isolation, as if testing a function. However, the "arguments" would then be data from the calling sub-program P_{L1}^{a}. If L1 is Java and L2 is Python, using Pytest to test P_{L2} as a unit may pose difficulties in constructing the appropriate Java objects within Python tests.

Both of these impacts are illustrated in Listing 1. First, to properly test the `sum` method of the Java part, it is necessary for the test writer to read the code written in a different programming language

```java
public class Point {
    public Point(String name, int x, int y) { ... }
    public int sum() {
        return polyglot.eval("python", foo(this));
    }
}
```

```python
@polyglot_export
def foo(p):
    if len(p.name) > 0:
        return complex_math(p)
    return -1

def complex_math(p):
    return p.x + p.y
```

Listing 1: Pseudo-code of a Java-Python polyglot program, where the Python computation function is not directly accessible in the Java program.

(*i.e.*, take into account the code of the Python `foo` function). Second, the Python function `foo` cannot be tested easily in isolation as a standalone function, as it takes a Java `Point` instance as argument which cannot be built from a Python test framework alone.

3.2 Scenario specification and execution

When writing a test, the scenario refers to the specification of which code will be called as part of the test. In other words, the scenario specifies which part of the program is under test. In the case of a polyglot program, specifying the scenario can pose some issues.

An immediate issue is the problem of testing foreign parts of a program. Many modern white box test frameworks (*e.g.*, JUnit [2], Pytest [5], Catch2 [1]) rely on being overlayed on top of a specific language. This is practical in a single-language case, as it allows to specify a scenario at the same abstraction level as the program, making it easy to specify exactly which parts are under test.

However in a polyglot environment, the tests being written in a single language may limit the specification of scenarios. For instance, in Listing 1, the function `complex_math` cannot be tested from a Java test framework, since it is not exported in the global memory space (*i.e.*, accessible from Java code). However, this might be an important function reused multiple times in the program that requires individual testing. This complements the earlier problem of constructing input data: using Pytest we cannot test the method with appropriate inputs, but using JUnit we cannot necessarily test the appropriate scenario. It is then needed to use many different test frameworks, which complicates test suite organization.

Another type of impact polyglot situations may have on scenario specification is when the polyglot calls are directly visible to the test writer. In many cases, we infer that polyglot interoperability will be mostly hidden behind utility functions in each language, where the retrieved data will be adapted to suit the local language's type system (*e.g.*, a JavaScript `number` gets converted to a Java `int` as appropriate). In those cases, the test writer does not have to manipulate polyglot values. However, there may also be situations where the test writer does have to directly manipulate the primitives of the platform if data is retrieved as-is, such as for more complex object types. This requires the test writer to be familiar with how the polyglot platform operates to specify the intended scenario, as mixing language semantics can sometimes be a source of errors [25].

While not directly test related, proper interfacing of the different languages [32] in the program may lessen this impact.

After specification, the next step is the execution of this scenario. In many cases, this is simply a matter of ensuring the test runner uses the polyglot execution platform. In fact, platforms using a common representation as underlying implementation for polyglot capabilities (such as .NET or the JVM) already provide foreign code execution capabilities with their standard test frameworks. However, despite their execution capabilities, these frameworks still lack flexibility when it comes to mixing languages in test suites.

3.3 Oracle expression

The last step of writing a test is expressing an oracle, or a condition that determines whether the program has successfully passed the test, or if it has failed [7]. In the context of a polyglot program, the oracle may need to be polyglot itself. If the oracle needs to cover multiple values across languages (*e.g.*, after the interaction of two objects originating from different languages), there may be a need to adapt the comparison operators to the polyglot nature of the operands. For instance, the `'=='` sign in a given assertion may not be the usual Java operator, but a polyglot version. This polyglot operator may already exist in some form in the execution platform (*e.g.*, GraalVM has built-in type conversion between languages), in which case it simply needs to be reused for assertions.

Additionally, the expression of an oracle becomes more complex when the values are linked, but not accessible from other languages. This could happen for instance in a web computation application, where a user computes results but has a limited amount of daily tokens: we can imagine a case where the result is displayed in a JavaScript front-end, but the token counter is stored in a Python back-end, making it difficult to test properties over both variables in one test framework for a given scenario.

In this case, current technologies may require splitting the test case into two test cases, one in each language, repeating the same scenario over the same data but with different parts of the oracle while losing the information that this is in fact the same test (*i.e.*, the same oracle property). This split would lead to maintainability and clarity issues, as a single conceptual test case is then duplicated into two actual tests. Whenever one test evolves the other has to co-evolve without any provided automation. In the absence of polyglot test frameworks, additional test suite "metadata" would then be required to keep track of the duplicated test cases.

4 RELATED WORK

Some research showcases the potential of polyglot development [31]. There are also concerns about the issues it brings and the lack of tooling [25]. The latter has started being answered with work on both static [14, 26] and dynamic [18, 19, 35] analysis of polyglot programs. Polyglot platforms have also been the subject of research, both from the angle of performance [12, 38] and definition [16].

Software testing has been extensively studied [6, 8, 17, 21]. However to the best of our knowledge, while test frameworks and methodologies geared toward specific application domains (including web applications [11], big data [22], deep learning [15, 39], crash resilient programs [24]...) exist, there is no work focusing on polyglot program testing. However, there is previous work on building a language-agnostic and extensible test coverage framework [34], although not taking into account polyglot interactions. Our work is in line with the theme of domain-specific test approaches as defined by Bertolino [9], tackling the domain of polyglot applications.

5 CONCLUSION & PERSPECTIVES

We presented research work towards the goal of testing polyglot programs. As a first step to establish better tools for testing polyglot programs, we gave an appropriate decomposition of how polyglot programs are built. Then, based on this decomposition, we performed an analysis of how polyglot situations may impact test writing. This analysis highlights several potential problems that may arise when testing polyglot programs w.r.t. input data selection, scenario specification, and oracle expression. These problems both relate to the difficulty of mixing multiple languages (*e.g.*, the test writer needs to understand the polyglot semantics) and to the lack of supporting tools that would help with test writing (*e.g.*, single-language oriented test frameworks or test analysis tools).

Thus, we have laid the groundwork for future research aiming to tackle these problems. We envision several directions. As mentioned, some impact of polyglot programming on writing test is due to the lack of appropriate tooling. Hence, one possibility is to tackle the adaptation of many existing test tools to fit polyglot testing purposes. This ranges from relatively simple test suite analysis tools, to a fully dedicated polyglot test framework. We also envision and call for research work on how polyglot platforms themselves can be built in a way that facilitates understanding polyglot programs without a deep expertise in all of the languages present. This could help test writers grasp the code under test, and thus write more relevant test cases. Lastly, another contribution we envision is on how to configure the execution of polyglot tests, and build an adapted polyglot test runner based on the execution platform. Overall, we envision that this contribution will raise more interest towards the problem of testing polyglot programs. In particular, future work should focus on addressing the identified challenges. This will support developers in the crucial activity of testing and also contribute to further promote the adoption of polyglot programming and further research around its testing (*e.g.*, polyglot test generation, fuzzing [23], co-evolution [20], etc.).

ACKNOWLEDGMENTS

The research leading to these results has received funding from the *ANR* agency under grant *ANR JCJC MC-EVO² 204687*.

REFERENCES

[1] [n. d.]. Catch2 github repository. https://github.com/catchorg/Catch2.

[2] [n. d.]. Junit 5 Official web page. https://junit.org/junit5/.

[3] [n. d.]. Lisp FFI documentation. https://lispcookbook.github.io/cl-cookbook/ffi.html.

[4] [n. d.]. MetaCall Official web page. https://metacall.io/.

[5] [n. d.]. Pytest module documentation. https://docs.pytest.org.

[6] Paul Ammann and Jeff Offutt. 2016. *Introduction to software testing*. Cambridge University Press.

[7] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2014. The oracle problem in software testing: A survey. *IEEE transactions on software engineering* 41, 5 (2014), 507–525.

[8] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2014. The oracle problem in software testing: A survey. *IEEE transactions on software engineering* 41, 5 (2014), 507–525.

[9] Antonia Bertolino. 2007. Software testing research: Achievements, challenges, dreams. In *Future of Software Engineering (FOSE'07)*. IEEE, 85–103.

[10] Jan Ehmueller, Alexander Riese, Hendrik Tjabben, Fabio Niephaus, and Robert Hirschfeld. 2020. Polyglot code finder. In *Companion Proceedings of the 4th International Conference on Art, Science, and Engineering of Programming*. 106–112.

[11] Satish Gojare, Rahul Joshi, and Dhanashree Gaigaware. 2015. Analysis and design of selenium webdriver automation testing framework. *Procedia Computer Science* 50 (2015), 341–346.

[12] Matthias Grimmer, Chris Seaton, Roland Schatz, Thomas Würthinger, and Hanspeter Mössenböck. 2015. High-Performance Cross-Language Interoperability in a Multi-Language Runtime. *SIGPLAN Not.* 51, 2 (oct 2015), 78–90. https://doi.org/10.1145/2936313.2816714

[13] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. In *38th PLDI*. 185–200.

[14] Philémon Houdaille, Djamel Eddine Khelladi, Romain Briend, Robbert Jongeling, and Benoit Combemale. 2023. Polyglot AST: Towards Enabling Polyglot Code Analysis. In *ICECCS 2023*.

[15] Qiang Hu, Lei Ma, Xiaofei Xie, Bing Yu, Yang Liu, and Jianjun Zhao. 2019. Deepmutation++: A mutation testing framework for deep learning systems. In *2019 34th IEEE/ACM ASE*. IEEE, 1158–1161.

[16] Christian Humer, Christian Wimmer, Christian Wirth, Andreas Wöß, and Thomas Würthinger. 2014. A Domain-Specific Language for Building Self-Optimizing AST Interpreters. *SIGPLAN Not.* 50, 3 (sep 2014), 123–132. https://doi.org/10.1145/2775053.2658776

[17] Paul C Jorgensen. 2018. *Software testing: a craftsman's approach*. CRC press.

[18] Sebastian Kloibhofer, Thomas Pointhuber, Maximilian Heisinger, Hanspeter Mössenböck, Lukas Stadler, and David Leopoldseder. 2020. SymJEx: symbolic execution on the GraalVM. In *Proceedings of the 17th MPLR*. 63–72.

[19] Jacob Kreindl, Daniele Bonetta, Lukas Stadler, David Leopoldseder, and Hanspeter Mössenböck. 2020. Multi-Language Dynamic Taint Analysis in a Polyglot Virtual Machine. In *17th MPLRs* (Virtual, UK). 15–29. https://doi.org/10.1145/3426182.3426184

[20] Quentin Le Dilavrec, Djamel Eddine Khelladi, Arnaud Blouin, and Jean-Marc Jézéquel. 2021. Untangling spaghetti of evolutions in software histories to identify code and test co-evolutions. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 206–216.

[21] William E Lewis. 2017. *Software testing and continuous quality improvement*. CRC press.

[22] Nan Li, Anthony Escalona, Yun Guo, and Jeff Offutt. 2015. A scalable big data test framework. In *2015 IEEE 8th international conference on software testing, verification and validation (ICST)*. IEEE, 1–2.

[23] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. 2018. Fuzzing: State of the art. *IEEE Transactions on Reliability* 67, 3 (2018), 1199–1218.

[24] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. 2019. PMTest: A fast and flexible testing framework for persistent memory programs. In *Proceedings of the 24th ASPLOS*. 411–425.

[25] Philip Mayer, Michael Kirsch, and Minh Anh Le. 2017. On multi-language software development, cross-language links and accompanying tools: a survey of professional software developers. *Journal of Software Engineering Research and Development* 5 (2017), 1–33.

[26] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. 2021. A multilanguage static analysis of python programs with native C extensions. In *International Static Analysis Symposium*. Springer, 323–345.

[27] Gunter Mussbacher, Benoit Combemale, Jörg Kienzle, Lola Burgueño, Antonio Garcia-Dominguez, Jean-Marc Jézéquel, Gwendal Jouneaux, Djamel-Eddine Khelladi, Sébastien Mosser, Corinne Pulgar, et al. 2024. Polyglot Software Development: Wait, What? *IEEE Software* (2024).

[28] Fabio Niephaus, Tim Felgentreff, and Robert Hirschfeld. 2019. GraalSqueak: toward a smalltalk-based tooling platform for polyglot programming. In *Proc. of MPLR'19*. 14–26.

[29] Fabio Niephaus, Tim Felgentreff, and Robert Hirschfeld. 2019. Towards polyglot adapters for the graalvm. In *Onwards!'19*.

[30] Fabio Niephaus, Eva Krebs, Christian Flach, Jens Lincke, and Robert Hirschfeld. 2019. PolyJuS: a Squeak/Smalltalk-based polyglot notebook system for the GraalVM. In *Companion Proceedings of the 3rd International Conference on the Art, Science, and Engineering of Programming*. 1–6.

[31] Cole S Peterson. 2021. Investigating the Effect of Polyglot Programming on Developers. In *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 1–2.

[32] Alexander Riese, Fabio Niephaus, Tim Felgentreff, and Robert Hirschfeld. 2020. User-Defined Interface Mappings for the GraalVM. In *4th International Conf. on Art, Science, and Engineering of Programming* (Porto, Portugal) *(<Programming> '20)*. 19–22. https://doi.org/10.1145/3397537.3399577

[33] José Antonio Romero-Ventura, Ulises Juárez-Martínez, and Adolfo Centeno-Téllez. 2022. Polyglot programming with graalvm applied to bioinformatics for dna sequence analysis. In *New Perspectives in Software Engineering: Proceedings of the 10th International Conference on Software Process Improvement (CIMPS 2021) 10*. Springer, 163–173.

[34] Kazunori Sakamoto, Kiyofumi Shimojo, Ryohei Takasawa, Hironori Washizaki, and Yoshiaki Fukazawa. 2013. OCCF: A framework for developing test coverage measurement tools supporting multiple programming languages. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, 422–430.

[35] Michael L Van de Vanter, Chris Seaton, Michael Haupt, Christian Humer, and Thomas Würthinger. 2018. Fast, Flexible, Polyglot Instrumentation Support for Debuggers and other Tools. *Art Sci. Eng. Program.* 2, 3 (2018), 14.

[36] Christian Wimmer and Thomas Würthinger. 2012. Truffle: A Self-Optimizing Runtime System. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity* (Tucson, Arizona, USA) *(SPLASH '12)*. Association for Computing Machinery, 13–14. https://doi.org/10.1145/2384716.2384723

[37] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to rule them all. In *Proc. of Onwards!'13*. 187–204.

[38] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. 2012. Self-Optimizing AST Interpreters. *SIGPLAN Not.* 48, 2 (oct 2012), 73–82. https://doi.org/10.1145/2480360.2384587

[39] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jianxiong Yin, and Simon See. 2019. Deephunter: a coverage-guided fuzz testing framework for deep neural networks. In *Proceedings of the 28th ACM SIGSOFT ISSTA*. 146–157.