This is a repository copy of *Polyglot Software Development:Wait, What?*.

White Rose Research Online URL for this paper:
https://eprints.whiterose.ac.uk/211594/

Version: Published Version

**Article:**

Mussbacher, Gunter, Combemale, Benoit, Kienzle, Joerg et al. (10 more authors) (2024) Polyglot Software Development:Wait, What? IEEE Software. pp. 124-133. ISSN 0740-7459

https://doi.org/10.1109/MS.2023.3347875

**Reuse**
This article is distributed under the terms of the Creative Commons Attribution (CC BY) licence. This licence allows you to distribute, remix, tweak, and build upon the work, even commercially, as long as you credit the authors for the original work. More information and the full terms of the licence here: https://creativecommons.org/licenses/

**Takedown**
If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.

eprints@whiterose.ac.uk
https://eprints.whiterose.ac.uk/

# Polyglot Software Development

## Wait, What?


**Gunter Mussbacher**, McGill University and INRIA

**Benoit Combemale**, Université de Rennes

**Jörg Kienzle**, Universidad de Málaga and McGill University

**Lola Burgueño**, Universidad de Málaga

**Antonio Garcia-Dominguez**, University of York

**Jean-Marc Jézéquel**, **Gwendal Jouneaux**, and **Djamel-Eddine Khelladi**, Université de Rennes and CRNS

**Sébastien Mosser**, McMaster University

**Corinne Pulgar**, Université du Québec

**Houari Sahraoui**, Université de Montréal

**Maximilian Schiedermeier**, McGill University

**Tijs van der Storm**, Centrum Wiskunde & Informatica and Rijksuniversiteit Groningen

©SHUTTERSTOCK.COM/JIRSAK


*Digital Object Identifier 10.1109/MS.2023.3347875*
*Date of publication 3 January 2024; date of current version 12 June 2024.*



*// We propose a concise and unambiguous definition of polyglot software development with a conceptual model and characterize the techniques used for the specification and operationalization of polyglot software development with a feature model. //*

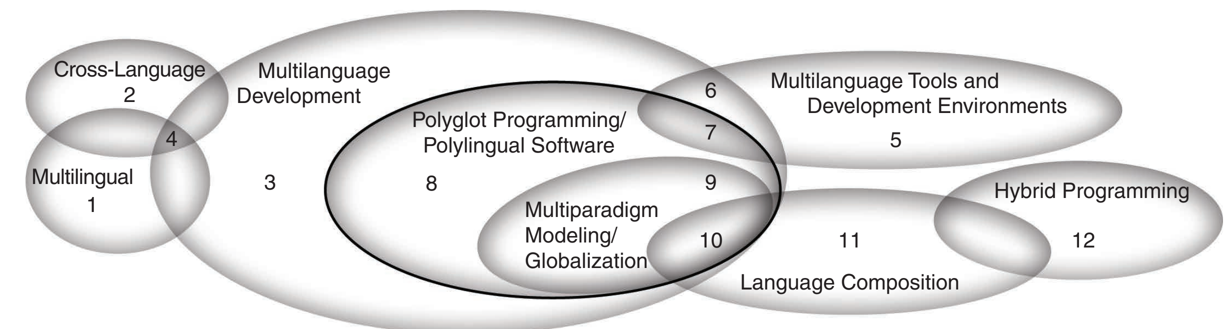

**MODERN SOFTWARE DEVELOPMENT** commonly requires the use of several languages in almost all activities, whether they involve requirements engineering, programming in one or more languages, or continuous integration and delivery. For example, requirements may be specified using templates for use cases or user stories and Gherkin scenarios.[1] Continuous integration and delivery may be specified with GitHub Actions and build languages such as Maven or Gradle.[2] The proliferation of domain-specific languages further adds to the incentive to use different languages for an activity.[3] Even a so-called Ruby project, such as Mastodon, an open source, distributed social media platform, in fact already uses many languages.[4] Besides Ruby, specifications in Docker Compose, Dockerfile, GitHub Actions, Haml, HTML, JavaScript, package.json, Rakefile, SCSS, and Structured Query Language are used to handle user interface, persistence, and build issues. Mastodon is not an isolated example. In 2017, Mayer et al. conducted a survey to gather responses

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

from 139 professional software developers, who reported an average of seven languages per project, with more than 90% of developers reporting problems related to language interactions.[5]

There are many reasons why several languages are used in combination: sociotechnical reasons, such as practitioner expertise/preferences and best practices; conceptual reasons, such as separation of concerns, design decisions, and variability management; technical reasons, such as availability of libraries/functionality, efficiency, automation/reproduction, reasoning/analysis, and quality assurance; and business reasons, such as coping with legacy applications/systems, technological debt, and vendor lock-in.

It is therefore no surprise that many communities are investigating the combination of several languages.[6] Yet, a long and ambiguous list of terms exists for polyglot software development from different communities. We have illustrated all of the terms we discovered in Figure 1, and we also provide references to representative articles in the scientific literature that use that terminology. While by no means exhaustive, this list already showcases the lack of a common view; that is, different communities often use the same term with different meanings, or use different terms for the same meaning. The effect is a vastly ambiguous picture of the term *polyglot* as well as a merely blurry sketch of common associated implications for a development process. Our goal is to clarify this fuzziness by providing a clear definition of polyglot software development. In turn, this may qualify as a common denominator for individual domain experts, to leverage an antisilo effect




**Legend**

1) "Lightweight Multilingual Software Analysis" doi.org/10.48550/arXiv.1808.01210
2) "CLCDSA: Cross Language Code Clone Detection Using Syntactical Features and API Documentation" doi.org/10.1109/ASE.2019.00099
3) "On Multilanguage Software Development, Cross-Language Links and Accompanying Tools: a Survey of Professional Software Developers" doi.org/10.1186/s40411-017-0035-z
4) "Cross-Language Interoperability in a Multi-Language Runtime" doi.org/10.1145/3201898
5) "The Design Space of Multilanguage Development Environments" doi.org/10.1007/s10270-013-0376-y
6) "Multilanguage Debugger Architecture" doi.org/10.1007/978-3-642-11266-9_61
7) "Debug All Your Code: Portable Mixed-Environment Debugging" doi.org/10.1145/1640089.1640105
   "Code Smells for Multilanguage Systems" doi.org/10.1145/3361149.3361161
8) "An Empirical Assessment of Polyglot-ism in GitHub" doi.org/10.1145/2601248.2601269
   "Investigating the Effect of Polyglot Programming on Developers" doi.org/10.1109/VL/HCC51201.2021.9576404
   "Automated Support for Seamless Interoperability in Polylingual Software Systems" doi.org/10.1145/250707.239123
9) "Multiparadigm Modeling for Cyber–Physical Systems: A Systematic Mapping Review" doi.org/10.1016/j.jss.2021.111081
10) "Globalizing Modeling Languages" doi.org/10.1109/MC.2014.147
11) "Language Composition Untangled" doi.org/10.1145/2427048.2427055
12) "A Hybrid Synchronous Language With Hierarchical Automata: Static Typing and Translation to Synchronous Code" doi.org/10.1145/2038642.2038664

**FIGURE 1.** Ambiguous terms related to polyglot software development.

that facilitates the exchange of contained knowledge.

In the remainder of this article, we first introduce a conceptual model for polyglot software development that allows us to clearly define polyglot software development and its polyglot processes and tasks and to discuss whether polyglot stakeholders are required. We exemplify the conceptual model with Mastodon and other examples. We further characterize polyglot software development and elaborate on polyglot programming, before concluding with open challenges and perspectives.

## Conceptual Model

To unify the large variety of terms related to the use of languages, this section proposes a conceptual model for software development with multiple languages in Figure 2. Note that we focus only on those development concepts that directly involve or somehow relate to languages.

At the heart of our conceptual model is the **Task**, which is a unit of work (for example, "specify web views") that involves a set of **StakeholderRoles** (such as "developer"). One **Stakeholder** may play one or more stakeholder roles. A task requires the use of one or several **Artifacts** expressed in one or more **Languages** because the artifacts are either consumed as **input** or produced as **output** by the task. Some artifacts may be integrated with each other using one or several **IntegrationTechniques**. A language offers one or more **Paradigms** in which to formulate the intended properties

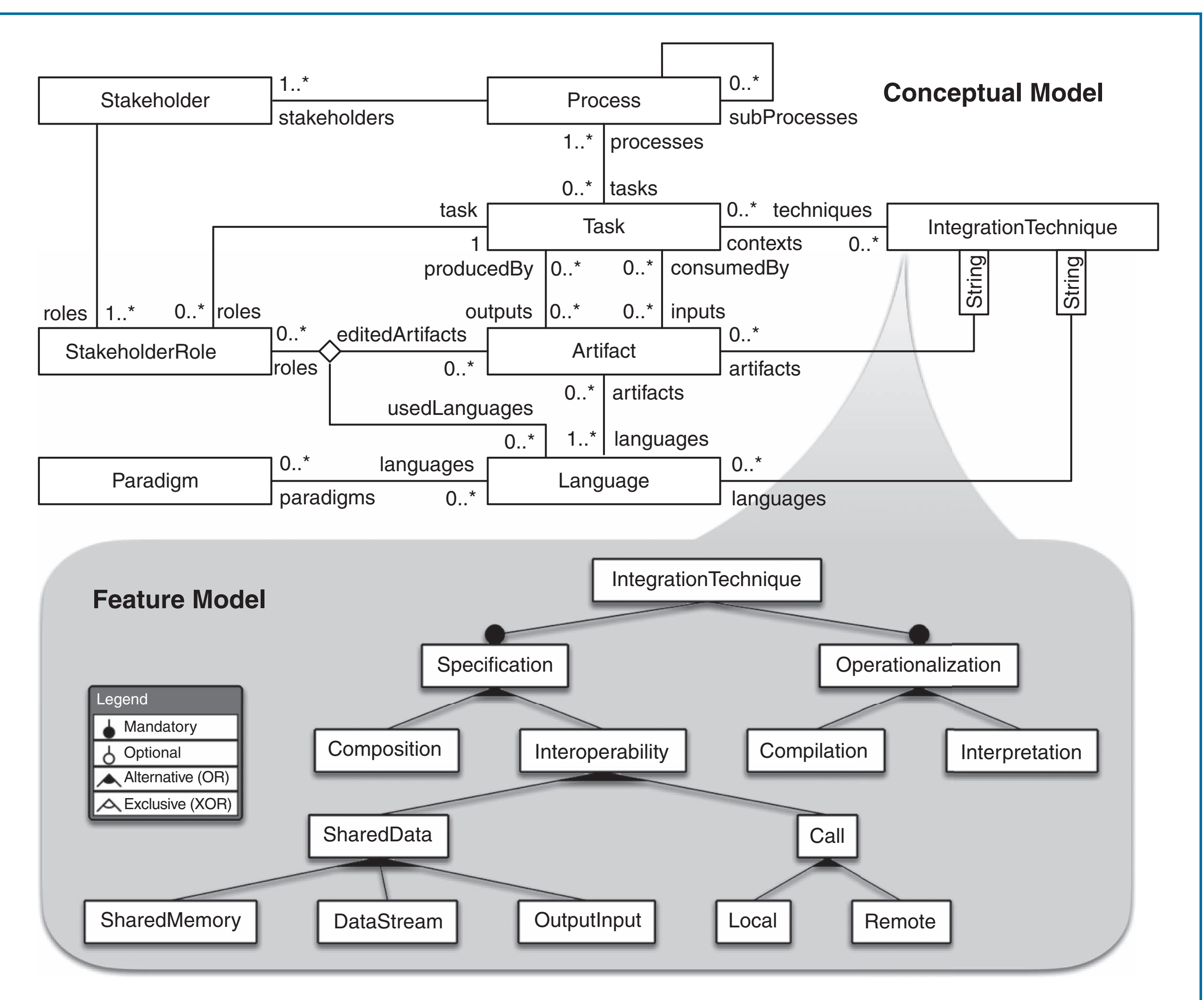



**FIGURE 2.** A conceptual model for polyglot software development and a feature model illustrating different integration techniques.

or behavior of the system under development (for example, "object-oriented programming," "functional programming," and "procedural programming" for Ruby).

An important distinction for a stakeholder role to be associated with an artifact of a language is that the role needs to actively **edit** something in the artifact (for example, write code, or add a model element). If this is not the case, then the stakeholder does not **use** the language. Simply viewing or executing an artifact does not qualify (such as the result of a model generation or compilation, respectively). For example, while the task of compiling code will require an input artifact and will output bytecode/machine code, most stakeholders will not directly engage with the compilation results. Hence, the stakeholders do not use the bytecode/machine code language, nor do they use the language of the input artifact since they do not edit it.

A ternary association is required since an artifact may be expressed in several languages, and a stakeholder role may only use some of those languages. For example, a performance specialist may edit only the MARTE annotations in a UML class diagram.

To bring artifacts of languages together for a task, a certain **IntegrationTechnique** is used, where each artifact and its language(s) play a role, captured in the conceptual model by the qualified associations between integration technique and artifact and between integration technique and language.

As an example, the "specify web views" task in Mastodon involves the creation of a "Haml" output artifact for the front-end developer and a "Ruby" output artifact for the back-end developer. These developers may in fact be the same person as a stakeholder may play multiple roles. Since this is a task that requires integrating two or more languages, the task uses an integration technique where Haml plays the role of "template" and Ruby is the "interpreter". The follow-up runtime task "generate web views" that produces artifacts in "HTML" from the integrated Haml+Ruby specifications is a task that involves no editing stakeholders but has two input artifacts and one output artifact.

Finally, during software development, tasks are typically performed in some order. For this purpose, our conceptual model contains the **Process** concept, which groups a set of tasks and a set of stakeholders. For the sake of practicality, we also allow processes to contain subprocesses, that is, to form hierarchies. We are not explicitly modeling the partial ordering of tasks within a process, though, as it is of no relevance regarding our discussion on polyglotism. Implicitly, a partial ordering is established nevertheless because tasks that require input artifacts can only be performed once the artifacts have been output by a preceding task in the process.

To finalize, we need to make the definition of a task more precise to avoid confusion among process, subprocesses, and tasks. A task is supposed to be the smallest unit of work; that is, it should not arbitrarily consist of artifacts with many languages that are not directly related to each other (for example, one task is defined for a whole process instead of splitting the process into several atomic tasks). We can do this by adding the following constraint to the conceptual model:

*A task may only contain artifact(s) of more than one language if the languages are integrated by a technique.*

```
context Task:
inv: roles.usedLanguages → asSet() → size() ≥
    2 implies techniques.artifacts
    → includesAll(roles.editedArtifacts)
    and techniques.languages
    → includesAll(roles.usedLanguages)
```

In the Mastodon project, for example, an activity such as "specify web views and build script" that includes Ruby, Haml, and Dockerfile would have to be modeled as two tasks.

## Polyglotism

Since the production of software always involves translation from human-readable languages to machine languages, all software development can be seen as polyglot. However, we are going to give a more nuanced definition of polyglot based on the *use of languages for a task* as explained earlier.

The conceptual model introduced allows for thinking about polyglotism at multiple levels, that is, at the task and the process levels and also with respect to stakeholder roles and stakeholders.

*A task is polyglot if the stakeholder roles of the task edit artifact(s) in more than one language.*

```
context Task def isPolyglot(): Boolean =
  roles.usedLanguages → asSet() → size() ≥ 2
```

For example, consider a task "specify web page" with an output artifact in two languages: HTML and Cascading Style Sheets (CSS). The task could require two stakeholder roles, one for HTML and one for CSS, or the same stakeholder role for both languages. In both cases the task is polyglot, and an integration technique is required because two languages are used in an edited

artifact. Another common situation occurs when a low-level language is embedded within a high-level programming language. For example, it is common to embed C code in Python to increase the performance of computationally expensive algorithms, and therefore any programming task with such a setup is polyglot. However, if the task is fully automated, that is, there is no stakeholder role, then the task is not polyglot. A polyglot task requires *active stakeholder involvement with multiple languages*.

This distinction is also exemplified by the tasks "write model transformation" and "run model transformation". Both tasks are not polyglot. The former is not polyglot because it involves a stakeholder role that edits the output artifact in only a single language, for example, an ATL script for the model transformation, based on two input artifacts, that is, the metamodels for the source and target languages of the transformation. The latter is not polyglot because it is automated and does not involve an active stakeholder role but three input artifacts (for example, the ATL script and two models corresponding to the source and target metamodels) and an automatically created output artifact in the target language.

Similarly, the specification of a consistency rule or an analysis script (such as energy consumption for webpages) is a task that is not polyglot unless the specification itself requires multiple languages. The metamodels of the languages for which a consistency rule is specified are the input artifacts and are not edited. Likewise, the webpages that are analyzed are also input artifacts that are not edited. The execution of the consistency rule (which may perform changes to the input models) and the running of the analysis are automated, and hence they are not polyglot because no stakeholder is actively involved.

Based on the definition of a polyglot task, similar definitions for stakeholder roles, stakeholders, and processes can be formulated:

*A stakeholder role is polyglot if it requires to edit artifact(s) in more than one language.*

```
context StakeholderRole def: isPolyglot():
    Boolean = usedLanguages → asSet() →
    size() ≥ 2
```

*A stakeholder needs to be polyglot if the union of roles they play edits artifact(s) in more than one language.*

```
context Stakeholder def: isPolyglot(): Boolean =
    roles.usedLanguages → asSet() → size() ≥ 2
```

*A process is polyglot if the stakeholder roles of the tasks that it or any of its subprocesses contains edit artifact(s) in more than one language.*

```
context Process def: isPolyglot(): Boolean =
    self.closure(subprocesses).tasks.roles
    .usedLanguages → asSet() → size() ≥ 2
```

For example, the earlier Ruby+Haml "specify web views" task has task-level polyglotism, but some other systems may exhibit process-level polyglotism. For instance, in a "data visualization" process, one task may use Python to transform data, and another task may use R to visualize the transformed data. At the uppermost process level, many modern systems will exhibit polyglotism (for example, using a formal requirements language and an implementation language).

On the other hand, there are still many projects that are not polyglot. For instance, there are numerous domains, such as data science, biology, or finance, whose projects use a single language (such as Python) for all tasks (for example, data curation, analysis, computation, visualization, etc.). Such a task is represented in the conceptual model by a task that produces an output artifact edited by a stakeholder role but only in the Python language and without any integration technique.

In the literature and in practice, different communities refer to the concepts in our conceptual model differently. This existing terminology (see Figure 1) can be mapped to our conceptual model as follows. "Polyglot development/programming" is in line with our definition of polyglotism. Within it, "multiparadigm modeling/globalization" are seminal approaches with an explicit focus on language integration (or *composition*) techniques. "Polyglot programming" and "polylingual software" as well as "multilanguage development" refer to a development process with tasks that span more than one language, but multilanguage development is more general and refers to approaches without language integration techniques. These terms should not be confused with "multilingual" software development tools, which include all language-agnostic tools that can be reused across a well-defined range of existing languages. "Cross-language" refers to tools that can operate across multiple languages while relating them (for example, when performing clone detection across Java and Python programs, the tool not only has to work on both Java and Python programs but also has to relate them). "Multilanguage tools and development environments" focus on the tooling aspect but do not contribute to the underlying foundations of

software development with multiple languages. By contrast, "language composition" techniques refer to work on the foundations for dealing with multiple languages, which may involve polyglot development but also language design and implementation for hybrid programming languages, that is, with multiple paradigms but without language integration techniques. Finally, "hybrid programming" refers to a single language that combines more than one paradigm (for example, continuous and discrete programming).

All communities depicted in Figure 1 build on the foundations of model-driven engineering (MDE) as well as language-oriented programming (LOP). In MDE, models play a central role during software development as the whole software life cycle is seen as a process of model production, refinement, and integration.[7] Similarly, in LOP a language is treated like any other development artifact, and, instead of using general-purpose languages, the creation and implementation of domain-specific languages for solving problems are preferred.[8]

## Integration Techniques

In this section, we provide more details on existing language integration techniques mentioned in the conceptual model by focusing on polyglot programming and hence executable artifacts. Figure 2 depicts the possible choices for the integration technique of executable artifacts as a feature model. Each feature represents a choice.

Each integration technique requires at least one choice for its **Specification** and one for its **Operationalization**. The former handles how we define the interaction between languages at design time, and the latter specifies how the interaction is realized during execution. The specification can be implemented with a **Composition** solution[9] and/or an **Interoperability** solution.[10] Composition covers all various techniques, from embedding of a language into another to unifying two languages at the syntax and/or semantic level. We do not provide further details on the many existing composition techniques and their classification, but the interested reader is referred to the survey article by Erdweg et al.[11]

> New opportunities await with the application of AI techniques to polyglot software development.

Interoperability covers the communication among different languages. Interoperability needs to deal with two important aspects, namely how data sharing (**SharedData**) and **Calls** are handled. The calls between languages can be either **Remote**, when the call goes through a network, or else **Local**. The shared data can either be implemented with a **SharedMemory**, a data streaming mechanism (**DataStream**), or simply by one language writing some output that another language consumes as an input, for example, through a file on disk (**OutputInput**).

Operationalization represents how the specification will be realized during execution. This can either be achieved through **Compilation** and/or **Interpretation**, that is, either by executing the relationships between the two languages at compile time, for example, in Melange,[6] or by interpreting the specified relationships at runtime, for example, in BCOoL.[12]

For example, a Scala program calling Java libraries fits the following choices in the feature model of Figure 2: shared memory and local call interoperability, and compilation operationalization. Another example is the case where code in one language invokes code in another language; for instance, the new Foreign Function and Memory application programming interface (API) in Java allows Java code to invoke low-level code and access data outside the Java virtual machine on the same machine. In other cases, interoperability happens through the use of an interface definition language, such as OpenAPI, from which client and server stubs are generated. This integration technique would use output/input and remote call interoperability. If, for example, Python talks to compiled C++, then the operationalization would use interpretation on the Python side and compilation on the C++ side.

Taking again the example of Mastodon, different integration techniques are used at various times. For instance, the integration technique between Haml and Ruby uses interoperability as specification through local calls to Haml code as well as shared memory, and it is operationalized using the Haml interpreter. A second used integration technique between Ruby and JavaScript relies

on interoperability as specification with a data stream using Redis and remote calls, and interpretation as operationalization.

As mentioned in the previous section, not every integration technique is associated with a polyglot task because stakeholder involvement is required. A fully automated task that is not polyglot may still have an integration technique. However, the earlier integration techniques between Haml and Ruby and between Ruby and JavaScript belong indeed to polyglot tasks since the stakeholders edit artifacts in all languages.

> To bring artifacts of languages together for a task, a certain Integration Technique is used, where each artifact and its language(s) play a role.

## Challenges and Perspectives

As mentioned previously, most software development is already polyglot to some extent, and it is not surprising that we see increasingly more languages appearing in modern software projects, for example, to build systems more efficiently or to separate concerns (see "To Make a Program"). Polyglot software development, however, faces many technical, process-related, educational, and community challenges. We discuss them and provide related perspectives.

### Technical Challenges and Perspectives

Some software development activities that are well understood within a single language become challenging in polyglot software development. For example, we need to develop novel and intuitive tools and techniques for polyglot software comprehension, polyglot software analysis (including, for example, semantic alignment, debugging, and profiling), and polyglot software documentation. Similarly, whereas testing each language separately is well supported, testing the overall polyglot program and its different interactions remains a challenge. Indeed, a test case would require one to integrate the "oracle states" of different programs written in different languages.

Techniques for software security will have to be revisited in the context of polyglot software development. For example, we need to ensure secure communication channels among languages and enable cross-language access control.

When developing polyglot programs, we often have to write the language integration logic from scratch. As a first step, our current code generators should be extended with a layer that automatically exposes the services by system components written in one language to the other languages. Ultimately, the goal is to have full-fledged code generation for polyglot programs that includes the integration logic.

Finally, new opportunities await with the application of artificial

## TO MAKE A PROGRAM

To make a program, it takes a language and a machine.
One language and a machine—at least in theory.
But practice asks for separation of concerns,
a division of labor between you, and me, and her.
The people demand speed and efficiency, but alas,
a language can compute anything, but is it fast?
So then we invite another and thus transgress
out of paradise with a bite, a sudden kiss of death,
and descend the tar pit of our fetished Babylon,
sentenced to tame the Hydra that we have
spawned.
Let's study the techniques of our tongues' embrace:
A language alongside another wants to communicate.
A language on top of another is one that generates.
A language within a language, a hatch for my escape.

So many tradeoffs at stake
when complexity procreates.— *Tijs van der Storm*

intelligence techniques to polyglot software development. More specifically, we should investigate how to capitalize on multilingual trained large language models.[13]

### Process-Related Challenges and Perspectives

We must develop strategies to determine the most appropriate combination of languages to use for a given task, also taking into account the sociotechnical context. We might even benefit from identifying antipatterns of language combinations from unsuccessful projects. We need to develop a theory for tradeoffs between productivity and complexity involved with polyglotism. Adding a language that is well suited to a task can speed up development, but it might also increase the cognitive load for the developer and require a broader range of development skills. Finally, a completely new challenge arises regarding language evolution. As many languages are used and interact with each other, when one evolves, others may be impacted as well. We would need to develop tools and techniques for polyglot impact analysis that can reason over multiple languages simultaneously. Then, when impacts are identified, they must be considered and languages have to coevolve accordingly.

### Educational Challenges and Perspectives

Most software engineering curricula contain courses that teach languages and paradigms, but only rarely are students explicitly exposed to polyglot software development with dedicated support for the coordinated use of multiple languages.[14] We need to find ways to use the presented conceptual model as an education tool to convey the real-life complexities to students who are used to "lab" projects as well as augment our teaching practices with examples of polyglot development activities and techniques to give a more holistic view of real-life software development.

### Community Challenges and Perspectives

In this article, we have identified similarities and variabilities in the terminology related to polyglot development used by various software engineering communities. Traditionally, different communities have been working in relative isolation from each other, and work like the one presented here can help break down the silos that separate them. Yet this work needs to be amended by the plethora of other communities dealing with polyglotism to enable global cross-fertilization.

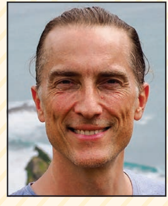
## References


1. M. S. Murtazina and T. V. Avdeenko, "Ontology-based approach to the requirements engineering in agile environment," in *Proc. 14th Int. Scientific-Tech. Conf. Actual Problems Electron. Instrum. Eng. (APEIE)*, Piscataway, NJ, USA: IEEE Press, 2018, pp. 496–501, doi: 10.1109/APEIE.2018.8546144.
2. M. Shahin, M. A. Babar, and L. Zhu, "Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices," *IEEE Access*, vol. 5, pp. 3909–3943, 2017, doi: 10.1109/ACCESS.2017.2685629.
3. T. Kosar, S. Bohra, and M. Mernik, "Domain-specific languages: A systematic mapping study," *Inf. Softw. Technol.*, vol. 71, pp. 77–91, Mar. 2016, doi: 10.1016/j.infsof.2015.11.001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584915001858
4. A. Raman, S. Joglekar, E. D. Cristofaro, N. Sastry, and G. Tyson, "Challenges in the decentralised web: The mastodon case," in *Proc. Internet Meas. Conf.*, 2019, pp. 217–229, doi: 10.1145/3355369.3355572.
5. P. Mayer, M. Kirsch, and M. A. Le, "On multi-language software development, cross-language links and accompanying tools: A survey of professional software developers," *J. Softw. Eng. Res. Develop.*, vol. 5, no. 1, 2017, Art. no. 1, doi: 10.1186/s40411-017-0035-z.
6. T. Degueule, B. Combemale, A. Blouin, O. Barais, and J.-M. Jézéquel, "Melange: A meta-language for modular and reusable development of DSLs," in *Proc. ACM SIGPLAN Int. Conf. Softw. Lang. Eng. (SLE)*, New York, NY, USA: Association for Computing Machinery, 2015, pp. 25–36, doi: 10.1145/2814251.2814252.
7. D. C. Schmidt, "Guest Editor's Introduction: Model-driven engineering," *Computer*, vol. 39, no. 2, pp. 25–31, Feb. 2006, doi: 10.1109/MC.2006.58.
8. R. Pickering, "Language-oriented programming," in *Beginning*. Berkeley, CA, USA: Apress, 2010, pp. 327–349.
9. J. Kienzle, G. Mussbacher, B. Combemale, and J. Deantoni, "A unifying framework for homogeneous model composition," *Softw. Syst. Model.*, vol. 18, no. 5, pp. 3005–3023, Jan. 2019, doi: 10.1007/s10270-018-00707-8.
10. M. Grimmer, R. Schatz, C. Seaton, T. Würthinger, M. Luján, and H. Mössenböck, "Cross-language interoperability in a multi-language runtime," *ACM Trans. Program. Lang. Syst.*, vol. 40, no. 2, May 2018, Art. no. 8, doi: 10.1145/3201898.
11. S. Erdweg, P. G. Giarrusso, and T. Rendel, "Language composition

## ABOUT THE AUTHORS

**GUNTER MUSSBACHER** is an associate professor at McGill University, Montreal, QC H3A 0E9, Canada. His research interests include model-driven requirements and software language engineering, sustainability, and human values. Mussbacher received his Ph.D. in computer science from the University of Ottawa. Contact him at gunter.mussbacher@mcgill.ca and http://www.ece.mcgill.ca/~gmussb1/.

**BENOIT COMBEMALE** is a full professor of software engineering at the ESIR, University of Rennes, 35065 Rennes, France, and cohead of the DiverSE research team. His research interests include model-driven and software language engineering and DevOps. Combemale received his a Ph.D. in software engineering from the University of Toulouse. He is a Member of IEEE. Contact him at benoit.combemale@irisa.fr and http://combemale.fr/.

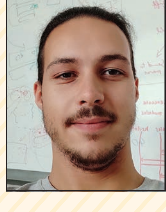


**JÖRG KIENZLE** is a researcher at ITIS Software, Universidad de Málaga, Málaga, Spain, and a full professor at McGill University, Montreal, QC H3A 0E9, Canada. His research interests include model-driven software development, software product lines, and modularity. Kienzle received his Ph.D. in computer science from the Swiss Federal Institute of Technology. Contact him at joerg.kienzle@uma.es, joerg.kienzle@mcgill.ca, and https://djeminy.github.io.

**LOLA BURGUEÑO** is an associate professor at the University of Málaga, 29071 Malaga, Spain. Her research interests include artificial intelligence in software development, uncertainty management, and software testing. Burgueño received her Ph.D. in software engineering and artificial intelligence from the University of Málaga. Contact her at lolaburgueno@uma.es and https://lolaburgueno.github.io.

**ANTONIO GARCIA-DOMINGUEZ** is a lecturer in software engineering in the Department of Computer Science at the University of York, YO10 5GH York, U.K. His research interests are model-driven software engineering and software testing. Garcia-Dominguez received his Ph.D. in engineering and architecture from the University of Cádiz. Contact him at a.garcia-dominguez@york.ac.uk and https://www-users.york.ac.uk/~agd516/

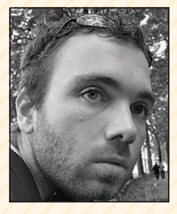
**JEAN-MARC JÉZÉQUEL** is a professor at the University of Rennes, 35042 Rennes, France, and a member of the DiverSE team at IRISA/Inria. His research interests include model-driven software engineering. Jézequel received his Ph.D. in computer science from the University of Rennes. He is a Senior Member of IEEE. Contact him at jezequel@irisa.fr and http://people.irisa.fr/Jean-Marc.Jezequel.



**GWENDAL JOUNEAUX** is a Ph.D. student in software engineering at the University of Rennes, 35042 Rennes, France, and a member of the DiverSE research team. His research interests are model-driven and software language engineering and self-adaptable languages. Jouneaux received his master's degree in software engineering from the University of Rennes. Contact him at gwendal.jouneaux@irisa.fr and https://www.gwendal-jouneaux.fr.

**DJAMEL-EDDINE KHELLADI** is a CNRS researcher at the IRISA lab in the DiverSE team, Université Rennes 1, 35000 Rennes, France. His research interests are model-driven engineering, scaling code analysis, and software processes. Khelladi received his Ph.D. in computer science from the University of Paris 6. Contact him at djamel-eddine.khelladi@irisa.fr and http://people.irisa.fr/Djamel-Eddine.Khelladi/.

## ABOUT THE AUTHORS

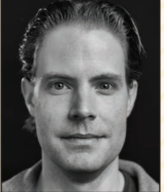


**SÉBASTIEN MOSSER** is a professor of software engineering at McMaster University, Hamilton, ON L8S 4L8, Canada, and a member of the McSCert centre. His research interests are related to domain-specific modeling and software composition from a language point of view. Mosser received his Ph.D. in software engineering from the Université de Nice-Sophia Antipolis. Contact him at mossers@mcmaster.ca and https://mosser.github.io/.

**CORINNE PULGAR** is a master's student at Ecole de Technologie Supérieure, Université du Québec, Montreal, QC, H3C 3P8 Canada. Their research interests include model-driven engineering, domain specific languages, DevOps, and inclusivity. Pulgar received their bachelor's degree in computer science from the Université du Québec. Contact them at corinne.pulgar.1@ens.etsmtl.ca and https://www.linkedin.com/in/corinne-pulgar-12a58190/

**HOUARI SAHRAOUI** is a professor in the Department of Computer Science and Operations Research at the Université de Montréal, Montreal, QC H3C 3J7, Canada. His research interests include artificial intelligence applied to software engineering and search-based software and model-driven engineering. Sahraoui received his Ph.D. in computer science from Université Pierre et Marie Curie. Contact him at sahraouh@iro.umontreal.ca.

**MAXIMILIAN SCHIEDERMEIER** is a Ph.D. student in computer science at McGill University, Montreal, QC H3A 0E9, Canada. His research focuses on domain-specific language-based tools for Representational State Transfer development/security protocol integration and empirical assessments. Schiedermeier received his master's degree in computer science from Universität Passau. Contact him at max.schiedermeier@mcgill.ca and https://m5c.github.io/.



**TIJS VAN DER STORM** is a senior researcher and group leader of the Software Analysis & Transformation group at CWI, 1098 XG Amsterdam, The Netherlands, and a professor of software engineering at the University of Groningen. His expertise spans language engineering, domain-specific languages, and model-driven engineering. Van der Storm received his Ph.D. from the University of Amsterdam. Contact him at storm@cwi.nl and http://www.cwi.nl/~storm.



untangled," in *Proc. 12th Workshop Lang. Descriptions, Tools, Appl. (LDTA)*, New York, NY, USA: Association for Computing Machinery, 2012, pp. 1–8, doi: 10.1145/2427048.2427055.

12. M. E. Vara Larsen, J. DeAntoni, B. Combemale, and F. Mallet, "A behavioral coordination operator language (BCOoL)," in *Proc. ACM/IEEE 18th Int. Conf. Model Driven Eng. Lang. Syst. (MODELS)*, 2015, pp. 186–195, doi: 10.1109/MODELS.2015.7338249.
13. T. Ahmed and P. Devanbu, "Multilingual training for software engineering," in *Proc. IEEE/ACM 44th Int. Conf. Softw. Eng. (ICSE)*, Los Alamitos, CA, USA: IEEE Computer Society, May 2022, pp. 1443–1455, doi: 10.1145/3510003.3510049.
14. M. Ardis, D. Budgen, G. W. Hislop, J. Offutt, M. Sebern, and W. Visser, "SE 2014: Curriculum guidelines for undergraduate degree programs in software engineering," *Computer*, vol. 48, no. 11, pp. 106–109, Nov. 2015, doi: 10.1109/MC.2015.345.