Towards flexible Integrated Development Environment

Fabien Coulon

DOCTORAT BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES 1

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par
Fabien COULON

Towards flexible Integrated Development Environment

Thèse présentée et soutenue à Rennes, le 3 mars 2022
Unité de recherche : IRISA (UMR 6074) Institut de Recherche en Informatique et Systemes Aléatoires

Rapporteurs avant soutenance :

Sophie EBERSOLD Maître de conférences HDR, Université Toulouse - Jean Jaurès
Jean-Remy FALLERI Maître de conférences HDR, Université de Bordeaux

Composition du Jury :

Président :	Guillaume PIERRE	Professeur, Université de Rennes 1
Examinateurs :	Réda BENDRAOU	Professeur, Université Paris Nanterre
	Sophie EBERSOLD	Maître de conférences HDR, Université Toulouse - Jean Jaurès
	Jean-Remy FALLERI	Maître de conférences HDR, Université de Bordeaux
	Olivier BARAIS	Professeur, Université de Rennes 1
Dir. de thèse :	Benoit COMBEMALE	Professeur, Université de Rennes 1

ACKNOWLEDGEMENT

Voici mes remerciments aux personnes qui m'ont permis d'achever cette thèse.
Merci à Benoit pour avoir toujours été de bon conseil et à Olivier qui à toujours des techno cool à tester.
Merci aux permanents, doctorants et postdocs de l'équipe DiverSE, trop nombreux pour être tous nommés, que j'ai eu le plaisir de côtoyer toutes ces années et qui sont des gens géniaux.
Merci à Kévin et Etiennes pour toutes ces parties de figurines et jeux vidéo.
Et bien sûr merci à ma famille.

So Long, and Thanks for All the Fish!

TABLE OF CONTENTS

CHAPITRE 1

RÉSUMÉ EN FRANÇAIS

1.1 Contexte

Les Environnements de Développement Intégrés (EDI) sont des logiciels qui regroupent de nombreux services pour les activités de développement et offrent un accès unifié à ces services, ce qui en font des outils majeurs pour les développeurs. Les développeurs sont des utilisateurs de langages, c'est-à-dire qu'ils manipulent les constructions d'un langage par le biais de services de langage. Ces services sont réalisés par les concepteurs de langages, qui attendent de plus en plus des EDIs qu'ils soient des plateformes extensibles afin de les utiliser comme base pour fournir des services de langages logiciels. Ces langages peuvent être des langages à usage général (General-Purpose Language ou GPL en anglais) conçus pour être utilisés dans n'importe quel domaine d'application (*i.e.,* Java, SQL, XML, etc), mais aussi des langages spécifiques à un domaine (DSL pour Domain Specific Language en anglais) dédiés à un domaine d'application particulier [101]. La conception et la réalisation des langages logiciels impliquent les mêmes tâches que celles requises par le cycle de vie des logiciels [45], allant de la spécification et de la réalisation des services de langage à leur déploiement. Les concepteurs de langage sont aidés par l'ingénierie des langages logiciels (SLE pour Software Language Engineering en anglais), qui est une discipline rationalisant le processus de développement des langages logiciels [81]. Les concepteurs de langage suivent les principes du SLE pour définir leur langage et l'implémenter dans une pile technologique, et à la fin du processus, peuvent intégrer leurs services de langage dans un EDI et les mettre à la disposition des futurs utilisateurs du langage.

La tendance récente pour les EDIs est d'être implémentés en tant qu'applications web. Les EDIs pour le Web adoptent le modèle de logiciel en tant que service (Software as a Service en anglais) pour fournir un accès à distance à un environnement de développement via un navigateur Web. L'un des principaux objectifs des EDIs Web est de fournir un environnement de développement sans installation. La configuration de l'environnement

de développement est la première activité d'un développeur. C'est une tâche qui consiste à récupérer le code source, à installer les outils, à installer les dépendances, etc. C'est une tâche qui prend du temps, qui est obligatoire mais aussi spécifique à chaque projet. L'EDI Web est une solution qui réduit le coût du passage d'un projet à l'autre en fournissant un environnement de développement prêt à l'emploi. Un autre avantage d'un EDI Web est qu'il permet de travailler n'importe où puisque seul un navigateur Web et une connexion Internet sont nécessaires. Les développeurs peuvent donc passer d'une machine à l'autre de manière transparente. La fourniture d'un EDI sous forme de services permet aux concepteurs de langage de faire de la livraison continue grace à laquelle les services de l'EDI peuvent être mis à jour et fournis directement aux utilisateurs d'un langage sans interrompre leurs activités. L'exécution des services de langage sur une machine distante est également un moyen de préserver les ressources du client, telles que l'utilisation du processeur, de la mémoire ou de la batterie lors de l'exécution de calculs intensifs.

La réalisation d'EDIs en tant qu'applications web soulève de nouveaux défis que nous avons abordés dans un contexte industriel par le biais d'un partenariat entre une entreprise et un laboratoire dans le cadre du programme doctoral CIFRE (Convention Industrielle de Formation par la Recherche).

1.2 Enoncé du problème

Lorsqu'il s'agit d'implémenter un langage, le concepteur choisit une pile technologique particulière, qui englobe la manière de définir le langage (grammaire, métamodèle,...), le langage de programmation à utiliser, quel framework, etc. Le concepteur de langage fait ce choix en fonction des atouts particuliers de la pile technologique (par exemple, la facilité d'écrire des transformations de code, de réaliser des interpréteurs, etc. Cependant, l'implémentation d'un langage dans une pile technologique particulière est un choix à long terme qu'il est difficile de modifier par la suite et, de plus, il est difficile de combiner plusieurs piles technologiques qui peuvent être éloignées tant dans leur formalisme que sur le plan technique. Ce manque de flexibilité dans l'implémentation empêche le concepteur de langage de bénéficier des avantages de plusieurs piles technologiques en même temps et ne permet pas aux utilisateurs de langage de choisir la pile technologique la plus appropriée à leur activité actuelle (c'est-à-dire d'utiliser les meilleurs services de langage fournis par une pile technologique particulière).

Les implémentations de langage sont constituées de services hétérogènes. Chaque service de langage a des besoins spécifiques, c'est à dire qu'ils peuvent avoir par exemple des fréquences d'utilisation différentes ou des complexité de calcul différentes ce qui implique une consommation différente des ressources (par exemple le CPU, la mémoire, etc). Dans le même temps, de nombreuses plateformes d'exécution sont disponibles pour ces services (serveurs dédiés, cloud, ordinateurs portables, etc.), chacune ayant ses propres caractéristiques (les serveurs dédiés disposent de beaucoup de CPU et de mémoire, le cloud fournit un mécanisme de déploiement automatique qui facilite la mise à l'échelle des applications, l'ordinateur portable de l'utilisateur du langage évite la latence du réseau, etc).

Les EDIs Web existants ne tirent pas parti de cette diversité de ressources fournies par les platesformes d'exécution disponibles, car ils sont composés d'un client s'exécutant dans un navigateur Web et d'un serveur monolithique fournissant les services de langage. Leur principal objectif est de fournir un environnement de développement prêt à l'emploi afin d'éviter la perte de temps liée à la configuration de cet environnement. À cette fin, ils s'appuient sur l'isolation de l'ensemble de l'environnement de développement. Cette isolation est assurée par le serveur qui fournit les services de langage et l'espace de travail de l'utilisateur contenant les artefacts développés. Cependant, les serveurs de langage sont des applications monolithiques, qui limitent la flexibilité des EDIs basés sur le Web, tant au niveau de l'implémentation que du déploiement du langage. Cela limite l'implémentation puisqu'elle ne permet au concepteur de langage d'utiliser qu'une seule pile technologique et cela limite le déploiement puisque la nature monolithique du serveur implique que tous les services de langage soient co-localisés. La co-localisation ne permet pas le déploiement indépendant des services de langage nécessaire pour allouer les ressources fournies par les différentes plateformes d'exécution disponibles.

Tirer parti de la diversité de plusieurs platesformes d'exécution en distribuant des services de langage hétérogènes aux endroits adaptés à leurs besoins pourrait profiter aux utilisateurs du langage en leur donnant de la flexibilité dans le déploiement des services de langage pour maximiser leurs performances. La distribution des services de langage est également l'occasion de construire des EDIs qui prennent en charge plusieurs utilisateurs en partageant des instances de services de langage entre plusieurs utilisateurs. Le partage des services de langage minimise les temps morts et évite le gaspillage de ressources par rapport au déploiement d'une instance de chaque service de langage pour chaque utilisateur.

Les concepteurs de langage ont une longue expérience de la construction des EDIs, mais le passage à une architecture distribuée ajoute des préoccupations supplémentaires. Dans ce contexte, les services de langage sont isolés et communiquent par échange de messages, ce qui nécessite la modularisation des services de langage. Ils peuvent également être déployés sur plusieurs plateformes d'exécution, potentiellement de manière dynamique, ce qui nécessite un moyen de configurer leur déploiement. Cependant, il n'existe pas d'abstractions appropriées pour décrire un environnement de développement distribué permettant de faciliter le développement de services de langage et de supporter l'automatisation de leur déploiement fiable.

Les limites de la flexibilité dans l'implémentation et le déploiement des langages peuvent être résumées par les deux défis suivants :

Défi #1 : Les concepteurs de langage doivent pouvoir tirer parti des atouts spécifiques des nombreuses piles technologiques possibles pour implémenter les langages et, inversement, les utilisateurs de langage doivent pouvoir manipuler les constructions du langage en passant de manière transparente entre les services de langage implémentés dans des piles technologiques différentes.

Défi #2 : Les services de langage doivent pouvoir exploiter les platesformes d'exécution disponibles pour répondre au mieux à leurs besoins en fonction des activités de l'utilisateur du langage, qui peuvent évoluer dans le temps.

1.3 Contributions

Nous adressons ces défis par deux contributions. Pour apporter plus de flexibilité dans l'implémentation des langages (**challenge #1**), nous proposons de connecter à un bus de communication différentes piles technologiques implémentant le même langage pour synchroniser les constructions du langage réalisées dans chaque pile. Le bus de communication est basé sur un modèle de publication/abonnement pour diffuser les changements survenus dans une pile technologique aux autres piles. L'idée générale est chaque pile technologique fournisse des constructions de langage équivalentes qui peuvent être manipulées par le biais de services de langage, et que tout changement résultant de ces manipulations est automatiquement signalé aux autres piles. Les constructions de langage sont toujours synchronisées et les utilisateurs peuvent passer sans problème d'un service de langage à l'autre à partir des différentes piles technologiques. Nous proposons un formalisme qui permet de représenter les changements sous la forme

d'une liste d'opérations atomiques agnostiques de toute pile technologique.

Nous avons implémenté le bus de communication comme un plugin Eclipse pour connecter les piles technologiques Eclipse Modeling Framework (EMF) [132], Rascal [84], et Java fluent API; permettant la combinaison de ces piles pour implémenter un langage et permettant de choisir dans quelle pile manipuler les constructions du langage.

Pour une plus grande flexibilité dans le déploiement des services de langage (**challenge #2**), nous proposons une approche générative qui permet de modulariser automatiquement les services de langage et qui permet de supporter leur déploiement de manière fiable sur la base de la spécification d'un protocole de communication. L'objectif est d'exploiter les ressources disponibles de plusieurs platesformes d'exécution pour répondre au mieux aux besoins des services de langage en les distribuant sur ces dites plateformes. Nous fournissons un DSL à destination du concepteur de langage pour spécifier les protocoles de communication qui sont ensuite utilisés comme entrées pour générer des microservices implémentant les services de langage. Nous générons aussi un modèle de variabilité pour piloter une fiable configuration de leur déploiement. Ce DSL nous permet de spécifier l'ensemble des services de langage, leurs interfaces et leurs interactions. Notre approche pour modulariser automatiquement les services de langage consiste à utiliser cette spécification de protocole en plus de la spécification d'un langage pour générer des microservices. Nous utilisons la spécification du langage pour générer le code source nécessaire au chargement d'un programme, qui est l'entrée de la logique du service de langage, et nous utilisons la spécification du protocole de communication pour générer le code source et les fichiers de configuration nécessaires à la communication avec d'autres microservices. Notre objectif est de générer des microservices prêts à être déployés, dans lesquels le concepteur du langage n'a plus qu'à implémenter leur logique interne. En complément, nous générons un modèle de variabilité, qui définit les contraintes de déploiement entre les services de langage, pour piloter la configuration de déploiement des services de langage de manière fiable. La configuration permet à l'utilisateur du langage de définir dynamiquement quels services doivent être déployés et sur quelle plateforme d'exécution.

Les générateurs ont été implémentés sous forme de plugins Eclipse pour être intégrés à l'écosystème EMF et fonctionnent avec des langages définis avec des métamodèles Ecore et des grammaires Xtext [44]. Les microservices générés sont intégrés dans des conteneurs Docker [100] et leur déploiement est effectué dans un cluster Kubernetes

[18] pour former un environnement de modélisation sous forme d'application nativement pour le cloud. Notre configurateur utilise l'API Kubernetes pour modifier dynamiquement la configuration du déploiement.

1.4 Résultats

Nous avons appliqué notre première contribution aux trois piles technologiques EMF, Rascal et Java fluent API dans lesquelles le langage Automate Fini à été implémenté. Nous avons pu diffuser les changements produits par les services de langage d'une pile technologique aux autres et ainsi manipuler le même automate fini à travers les services de langage des différentes piles.

Pour valider la généralisation de notre deuxième contribution, nous l'avons appliquée aux langages NabLab [94], Logo [1], MiniJava [121], et ThingML [61]. Sur ces langages, nous avons validé que nous étions capables de générer des microservices implémentant un ensemble de services de langage, que nous étions capables de les distribuer sur un cluster Kubernetes, et que nous étions capables de configurer dynamiquement le déploiement des services de langage.

Pour évaluer l'avantage de la distribution des services de langage, nous avons mesuré leurs temps de réponse pour le langage NabLab dans deux configurations : des services de langage fournis par un serveur monolithique déployé localement sur un ordinateur portable et des services de langage implémentés sous forme de microservices sans état déployés à distance sur de puissantes machines. Nous avons constaté qu'il est avantageux d'exécuter les services à forte intensité de calcul sous forme de microservices déployés à distance, car le gain en temps de calcul l'emporte sur la perte due à la modularisation et à la communication réseau.

Pour évaluer le coût de la distribution, nous avons mesuré la part due au protocole de communication et la part due au chargement des programmes dans les temps de réponse des microservices sans état implémentant les services de langage pour les quatre langages NabLab, Logo, ThingML et MiniJava. Nos résultats montrent que les temps de réponse des services de langage implémentés par des serveurs monolithiques, qui sont avec état, sont inférieurs à ceux des microservices, que les temps d'échange de messages entre microservices dues au protocole sont acceptables (i.e. inférieures à 200 ms), et que le coût de chargement d'un programme à chaque requête est une part significative de la mauvaise performance des microservices sans état par rapport aux

serveurs monolithiques.

En conclusion, notre première application montre que notre approche consistant à combiner des piles technologiques apporte une flexibilité tant au niveau de l'implémentation que de l'utilisation du langage. D'après la deuxième application et les résultats de notre expérimentation, nous concluons que la distribution de services de langage rend un EDI flexible dans son déploiement, ce qui nous permet de tirer parti des plateformes d'exécution disponibles pour améliorer les temps de réponse des services de langage.

CHAPTER 2

INTRODUCTION

This chapter first presents the research context of this thesis (Section 2) and defines the challenges addressed (Section 2.2). We then introduce the scientific contributions (Section 2.3) and the results of our evaluations (Section 2.4). Finally, we outline the content of the thesis (Section 2.5) and list the publications resulting from this work (Section 2.6).

2.1 Context

Integrated Development Environments (IDE) are software aggregating many services for development activities and providing unified access to these services, making them major tools for developers. Developers are language users which manipulate language constructs through language services. These services are implemented by language designers, who increasingly expect IDEs to be extensible platforms so that they can be used as the basis to provide the services of software languages. Such language can be General-Purpose Language (GPL) that are designed to be used for any application domain (*i.e.,* Java, SQL, XML, etc), but can also be Domain Specific Language (DSL) that are dedicated to a particular application domain [101]. The design and implementation of software languages involve the same tasks required by the lifecycle of software [45], ranging from the specification and implementation of language services to their deployments. The language designers are helped by the Software Language Engineering (SLE), which is a discipline rationalizing the process of developing software languages [81]. Language designers follow the principles of SLE to define their language and implement it in one technological stack, and at the end of the process may integrate its language services in an IDE and make them available to future language users.

The recent trend for IDEs is to be implemented as web applications. Web-based IDEs are adopting the Software as a Service (SaaS) model to provide remote access to a

development environment through a web browser. One of the main objectives of Web-based IDEs is to provide a development environment without installation. Configuring the development environment is the first and mandatory activity of a developer. It is a task consisting of retrieving the source code, installing tools, installing dependencies, etc. It is a time-consuming task, that is mandatory but also specific for each project. Web-based IDE is a solution to reduce the cost of switching between projects by providing ready to use development environment. Another advantage is that Web-based IDE makes it possible to work anywhere since only a web browser and an internet connection are needed. Thus developers can switch from machine to machine seamlessly. Providing an IDE as services opens for language designers the continuous delivery workflow by which the IDE services can be updated and provided directly to language users without interrupting their activities. Running language services on a remote machine is also a way to preserve the client's resources like CPU, memory, or battery from heavy computations.

Implementing IDE as a web application raises new challenges which we have addressed in an industrial context through a partnership between a company and a laboratory within the framework of the French doctoral program CIFRE [1].

2.2 Problem Statements

When he comes to implementing a language, the language designer chooses a particular technological stack, which encompasses the manner of defining the language (grammar, metamodel,...), the programming language to be used, which framework, etc. The language designer makes this choice based on the particular strengths of the technological stack (*i.e.,* the ease of writing code transformation, of implementing interpreters, etc). However implementing a language in one technological stack is a long-term choice that is hard to change later and, moreover, it is difficult to combine multiple technological stacks that may be distant both in their formalism and technically. This lack of flexibility in the language implementation prevents language designers from benefiting from the strengths of multiple technological stacks at the same time and does not allow language users to choose the most appropriate technological stack to manipulate language construct according to his current activity (*i.e.,* to use best language services from a particular technological stack).

1. stands for Industrial Agreement of Training through Research

Language implementations are made of heterogeneous services. Each language service has specific needs in the sense that they have different frequency of use, different computational complexity which implies a different consumption of resources (*e.g.,* CPU, memory, etc). At the same time, many execution platforms are available for these services, (*e.g.,* dedicated servers, clouds, laptops, etc), each with its own characteristics, (*e.g.,* dedicated servers have a lot of CPU and memory, the cloud provides an automatic deployment mechanism that facilitates application scaling, language user's laptop avoids network latency, etc).

Existing Web-based IDEs do not leverage this diversity of resources from available execution platforms because they are divided into a frontend component running in a web browser and a monolithic language server acting as a backend component. Their main purpose is to provide a ready-to-use development environment to avoid the time lost in configuring the development environment. To this end, they rely on the isolation of the whole development environment to provide an out-of-the-box environment. This is achieved through the server that provides the language services and the user workspace containing the developed artifacts. However, language servers are monolithic applications, which limit the flexibility of Web-based IDEs in both language implementation and deployment. It limits the implementation since it only allows language designers to use a single technological stack and it limits the deployment since the monolithic nature of the server implies that all language services are co-located and therefore prohibits the independent deployment of the language services necessary to allocate the resources provided by the different execution platforms of a cloud.

Leveraging the diversity of the available execution platforms by distributing the heterogeneous language services to the location tailored to their needs could benefit the language users by giving them flexibility in the deployment of language services to maximize their performances. The distribution of language services is also an opportunity to build IDEs that support multiple users by sharing instances of language services across multiple language users. Sharing language services minimize idle time and avoid wasted resources compared to deploying one instance of each language service for each language user.

Language designers have long experience building desktop IDEs, however, the move to a distributed architecture adds additional concerns. In this context, language services are isolated and communicate through the exchange of messages, which requires the modularization of language services. They can also be deployed across mul-

tiple execution platforms, possibly dynamically, which requires a means of configuring their deployment. However, there are no proper abstractions to describe a distributed developing environment to ease the development of language services and to support the automation of their safe deployment.

The limits of flexibility in language implementation and deployment can be summarized by the two following challenges:

Challenge #1: Language designers must be able to benefit from the specific strengths of the many possible technological stacks to implement languages and, on the other way around, language users must be able to seamlessly switch between language services implemented in different technological stacks to manipulate the same language constructs.

Challenge #2: The language services must be able to leverage the available execution platforms to best fit their needs according to the activities of the language user, which can evolve over time.

2.3 Contributions

We tackle the challenges with two contributions. To bring more flexibility in language implementation (**challenge #1**), we propose to synchronize the language constructs from different technological stacks implementing the same language by connecting them to a communication bus. The communication bus is based on a publish/-subscribe pattern to broadcast changes that happened in one technological stack to the other stacks. The main idea is that multiple technological stacks provide equivalent language constructs that can be manipulated through language services, and that any changes that may result from these manipulations are automatically reported across all stacks. Language constructs are always synchronized and language users can seamlessly switch between language services from the different technological stacks. We propose a formalism that allows changes to be represented in the form of a list of atomic operations agnostic of any technological stacks.

We implemented the communication bus as an Eclipse plugin to connect the technological stacks Eclipse Modeling Framework (EMF) [132], Rascal [84], and Java fluent API; allowing the combination of these stacks to implement a language and allowing to choose in which stack to manipulate the language constructs.

To enable greater flexibility in the deployment of language services (**challenge #2**),

we propose a generative approach to automatically modularize language services and support their safe deployment based on the specification of a communication protocol. The goal is to leverage the available resources of multiple execution platforms to best fit the needs of language services by distributing them across platforms. We provide a DSL at the destination of the language designer to specify communication protocols to be used as input to generate microservices implementing the language services and a feature model to drive the safe configuration of their deployment. This DSL allows us to specify the set of language services, their interfaces, and their interactions. Our approach to automatically modularize language services is to use this protocol specification in addition to the language specification to generate the microservices. We use the language specification to generate the source code needed to load a program, which is the input of the language service logic and we use the communication protocol specification to generate the source code and configuration files needed for the communication with other microservices. We aimed at generating ready-to-deploy microservices, in which the language designer only has to implement their internal logic. Complementary, we generate a feature model, which defines deployment constraints between the language services, to drive the safe deployment configuration of language services. The configuration allows a language user to dynamically define which services have to be deployed and on which execution platform.

The generators have been implemented as Eclipse plugins to be integrated into the EMF ecosystem and work with languages defined by Ecore metamodels and Xtext [44] grammars. The generated microservices are embedded in Docker [100] containers and their deployment is performed in a Kubernetes [18] cluster to form a modeling environment as a cloud-native application. Our configurator uses the Kubernetes API to dynamically change the deployment configuration.

2.4 Results

We applied our first contribution to the three technological stacks EMF, Rascal, and Java fluent API in which the Finite State Machine language was implemented. We were able to report the changes produced by the language services from one technological stack to the others and thus manipulate the same finite state machine through the language services of the different stacks.

To validate our second contribution is generalizable, we applied it to the NabLab

[94], Logo [1], MiniJava [121], and ThingML [61] languages. On these languages, we validated that we were able to generate microservices implementing a set of language services, that we were able to distribute them on a Kubernetes cluster, and that we were able to dynamically configure the deployment of the language services.

To evaluate the benefit of the distribution of language services, we measured their response times for the NabLab language in two configurations: language services provided by a monolithic server deployed locally on a laptop and language services implemented as stateless microservices deployed remotely on powerful machines. We have found that it is advantageous to run computationally intensive services as remotely deployed microservices, as the gain in computing time outweighs the loss due to modularization and network communication.

To evaluate the cost of distribution, we measured in the response times of the stateless microservices implementing language services of the four languages the part due to the communication protocol and the part due to the loading of the programs. Our results show that response times of language services implemented by monolithic servers, which are stateful, are lower than those of microservices, that message exchange overheads between microservices due to the protocol are acceptable (i.e., less than 200 ms), and that the cost of loading a program at each request is a significant part of the bad performance of stateless microservices compared to monolithic servers.

In conclusion, our first application shows that our approach of combining technological stacks brings flexibility for both implementation and language usage. From the second application and the results of our experimentation, we conclude that the distribution of language services makes an IDE flexible in its deployment, allowing us to leverage available execution platforms to improve language service response times.

2.5 Outline

The content of this thesis manuscript is organized in three parts containing 9 chapters as follows:

The first part (Part I) is composed of two chapters giving the context of this thesis. Chapter 3 presents the general background necessary for the understanding of this thesis. It presents the fields of Software Language Engineering and Cloud-native Applications. Chapter 4 reviews the state of the art of IDEs and discusses of their flexibility.

The second part (Part II) contains three chapters presenting the contributions of this

thesis. Chapter 5 gives an overview of the thesis and presents how the contributions are related. Chapter 6 describes our contributions on combining multiple technological stacks to implement and use languages. Chapter 7 presents our solution to language services modularization and language services distribution to leverage cloud infrastructure.

The third part (Part III) is composed of two concluding chapters. Chapter 8 is a summary of this thesis. Chapter 9 identifies the perspectives of our contributions opening to future works.

2.6 Publications

This section lists the publications in international conferences resulting from the work of this thesis.

— **Shape-diverse DSLs: languages without borders (vision paper)** *(distinguished vision paper award)* Fabien Coulon, Thomas Degueule, Tijs Van Der Storm, Benoit Combemale. In Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering, 2018 [30].

— **Modular and Distributed IDE** Fabien Coulon, Alex Auvolat, Benoit Combemale, Yérom-David Bromberg, François Taïani, Olivier Barais, Noël Plouzeau. In Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering, 2020 [29].

PART I

Background and State of the Art

CHAPTER 3

BACKGROUND

This thesis is about the application of cloud computing to software language engineering. We start this chapter by defining what a software language is (Section 3.1) and giving an overview of software language engineering (Section 3.1.1) before introducing the notion of metamorphic DSL (Section 3.1.2). We then introduce the principles of cloud computing (Section 3.2) and describe its different service models (Section 3.2.1). We conclude by presenting the properties of cloud-native applications (Section 3.2.2) and their realization in microservices (Section 3.2.3).

3.1 Software Language

Software languages are artificial languages implemented as software. They are involved in every activity of software and systems engineering [91] and are more and more numerous [90]. Software languages can be programming languages (e.g., C, Python, Javascript) but are not restricted to this domain. We can cite for example markup languages to structure documents (e.g., HTML, XML), query languages to request databases (e.g., SQL), configuration languages (e.g., YAML, JSON), modeling languages (e.g., Flowchart, ThingML), etc.

Software languages can be classified as General-Purpose Languages (GPL) or Domain-Specific Languages (DSL) [145]. GPLs are software languages general enough to target any application domain. It means that they don't contain features specialized for a particular domain. They are intended to be used by software-experts, which can make them difficult to handle by domain-experts who do not have the same skills. DSL suit better for domain-experts since they are languages at the right level of abstraction for them, making them easier for modifications and understanding [32]. DSLs are everywhere: web, embedded systems, simulation, security, testing, or education to cite a few domains where they are used [112]. Compared to GPLs, DSLs are tailored for one

particular application domain and have expressiveness that increases their ease of use [101]. DSLs might also improve language user efficiency [87] however their is a lake of usability evaluation [12].

Software languages can also be classified as internal or external languages. Internal languages are software languages using the constructs of host languages to express the concepts of their domain. Internal languages benefit from the tooling of the host language [72]. This makes it possible to reuse the existing infrastructure of the host language to avoid reimplementing it specifically for the internal language, and thus reduces the development cost. Reusing tools of the host language is also a limitation in the sense that they are not specifically tailored for the internal language, e.g., a source code validator may not detect errors specifics to the internal language. Fluent API [48] is an example of a technique used to implement internal languages in programming languages. This technique relies mostly on method chaining, object scoping and call of function as function's parameters. External languages are autonomous languages in the sense that they have their own dedicated tooling. The creation of an external language is the creation of a new language that requires writing its specification and implementing its tooling.

In the following section, we present the discipline of Software Language Engineering, which provides methods and techniques for software language development activities.

3.1.1 Software Language Engineering

Software Languages are software [45] and therefore face the same challenges in their development (e.g., design, test, deployment, maintenance, etc). The development tasks can even increase in complexity for the case of DSLs since it requires expertise in both application domain and language development [101]. The construction of domain-specific tools is expensive [149] and require language engineering skills [154].

Derived from Software Engineering, Software Language Engineering is the computing discipline for designing and implementing Software Languages. It is a systematic discipline providing methods to guide the language designers in the design and implementation of the aspects of a software language which mainly consist of the abstract syntax, the concrete syntax, and the semantics [60].

Figure 3.1 shows the relationships between these three aspects of software lan-

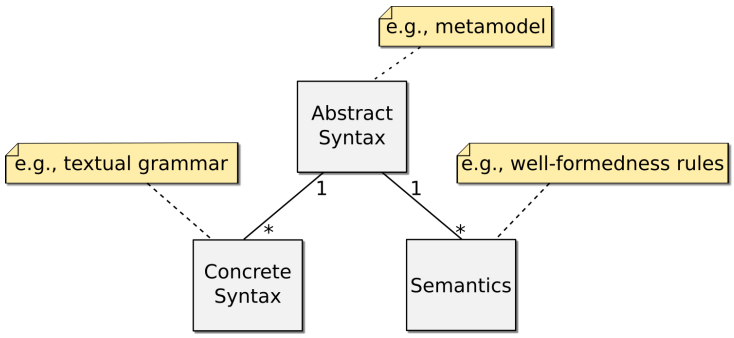
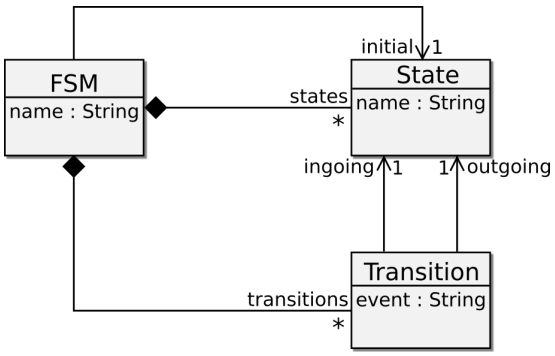


Figure 3.1 – Software language aspects and their relationships

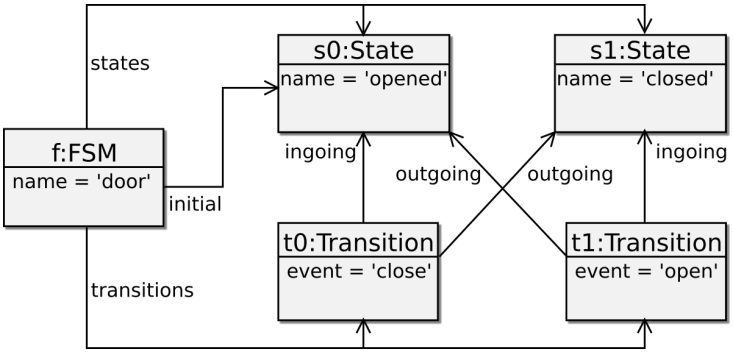
guages. The central aspect is the Abstract Syntax that structurally defines the concepts of a language, e.g., a metamodel. The Abstract Syntax is possibly complemented by Concrete Syntaxes that define how the concepts of a language will be presented to the language users, e.g., a textual grammar. The Abstract Syntax can also be completed by Semantics that add meaning, e.g., well-formedness rules. We detail these three software language aspects in the following.

An abstract syntax defines, for a given language, the concepts and their relationships. Concepts are used as language constructs to create the representation of a system of the application domain. Metamodels and models (i.e., a graph of objects) are examples of forms that such abstract syntax and representations can take. Language users manipulate representations through language services provided by the tooling of the language. However, these representations are not what the language users see since they are abstract data structures, meaning they are independent of concrete representations constructed thanks to the concrete syntax.

Figure 3.2 illustrates the definition and the usage of an abstract syntax for the Finite State Machine (FSM) language. The top sub-figure 3.2a presents the abstract syntax of the FSM language in the form of a metamodel. It define the three concepts of the language with the classes *FSM*, *State*, and *Transition*. The *FSM* is composed of *States* and *Transitions*, and has a reference to one *State* that design it has the initial state of the machine. A *Transition* has a reference to an ingoing *State*, a reference to an outgoing *State* and has an attribute *event* to define its trigger. Both *FSM* and *State* have an attribute *name*. The bottom sub-figure 3.2b presents an example of FSM representing the behavior of a door, in the form of a model conform to the metamodel presented in sub-figure 3.2a. The door is represented by an object *FSM* named 'door'. It has two objects *State* named 'opened' and 'closed', the former being referenced has the initial *State*. The ob-



(a) Abstract syntax of the FSM language, in the form of a metamodel



(b) The behavior of a door represented by a model conform to the metamodel of the FSM language

Figure 3.2 – Illustrative example of definition and usage of an abstract syntax with the Finite State Machine language

ject *FSM* contains also two objects *Transition*. The first *Transition* ingoing from the *State* named 'opened', outgoing to the *State* named 'closed' and is triggered by the event 'close'. The second *Transition* ingoing from the *State* named 'closed', outgoing to the *State* named 'opened' and is triggered by the event 'open'.

A concrete syntax defines the notations for representations presented to the language users. It is through these notations that the language users will understand and manipulate the abstract representations. The concrete syntax can be textual or graphical. Textual syntaxes are defined by textual grammars, i.e., production rules like EBNF [156]. Graphical syntaxes are defined by mapping elements of the abstract syntaxes with graphical representations like geometrical shapes, tables, etc [147, 151].

Listing 3.1 shows an illustrative concrete syntax for the Finite State Machine language in the form of a grammar. It is composed of production rules that define that

```
1 grammar FSM;
2
3 fsm : 'fsm' '{' state* transition* '}' ;
4 state : 'state' ID;
5 transition : ID '→' ID;
6
7 ID: [a-z]+;
```

Listing 3.1 – Illustrative example of concrete syntax with the Finite State Machine language

an FSM starts with the word *fsm* followed by a list of states and a list of transitions surrounded by curly brackets. A state must start with the word *state* followed by an identifier and a transition starts with an identifier followed by an arrow and another identifier. An identifier is defined as a sequence of at least one letter.

In addition to abstract syntax and concrete syntax to define languages, they are semantics that gives meaning to the language. Semantics are classified in static semantics or dynamic semantics. Static semantics are attached to language constructs and are about checking their well-formedness. Dynamic semantics is about computation and express the runtime behavior of languages. It can be one kind of operational, denotational or axiomatic, or hybrid [109]. Operational semantics defines language behavior with transitions between program states. The evaluation of operational semantics is the run of a sequence of steps that perform transformations of the program state. Denotational semantics defines the mapping between two languages. It is used to translate the constructs of a source language to constructs of a target language. Axiomatic semantics consists of making assertions on programs. It is used to define the properties of programs that have to be satisfied, e.g., by using Hoare logic [69].

Language workbenches are IDEs assisting language designers in the engineering of languages and improving the user experience of these languages [49]. In addition to tools and methods guiding language designers during the development, they provide metalanguages to specify the different concerns of software languages. Erdweg et al. [42] compared languages workbenches and proposed a feature model summarizing their different concerns. To give some examples of language workbenches, we can cite Gemoc [16], Xtext [15], Rascal [84], Spoofax [78] or JetBrains MPS [152].

3.1.2 Metamorphic DSL

When coming to implementing a DSL, language designers have to take design decisions. First of all, the language designer has to choose a shape for the DSL. The shape of a language is realized through a technological stack providing the means to implement the different concerns of a language. A good shape of a language depends on its user and its activity [28]. For example, an internal language, like an API is easily integrated with programs and therefore more suited for a programmer whereas an external form may be more intuitive for a non-software-expert. The decision on how to implement a DSL can also be based on the facilities provided by the language workbenches, each of which has strength in a particular concern of a language. For example, a language workbench like Rascal eases the implementation of refactoring tools.

The notion of metamorphic DSL was introduced in [3]. It is the idea that a DSL can have multiple shapes at the same time and that language users can switch between the shapes. Metamorphic DSL benefits both language designers who can leverage the strength of multiple technological stacks to implement a language (i.e., a shape) and language users who can use the most appropriate shape according to their activities.

```
1  SELECT *
2  FROM products
3  WHERE gps = 'true'
```

(a) SQL as external language

```
1  Optional<Product> person =
2  from(Product.class)
3    .where(Product::getGPS)
4    .equals("true")
5    .select(
6      productMapper,
7      connectionFactory::
           openConnection);
```

(b) SQL as fluent API

Figure 3.3 – Illustrative example of metamorphic SQL with an *external language* shape and a *fluent API* shape

Figure 3.3 shows an illustrative example of metamorphic DSL with the Structured Query Language (SQL), a language to write queries for databases. In this example, SQL as metamorphic language has two shapes: one as an *external language* and one as a *fluent API* in Java. Both sub-listings 3.3a and 3.3b show the same query (the selection of all products with the *GPS* property) in the two shapes of the metamorphic language. The both shapes of the metamorphic SQL are useful to the user but for different tasks. The strength of the *external language* shape is that the user can benefits of a dedicated

editor to write the queries. The *fluent API* shape offer to the user the opportunity to use the host language to build its queries (e.g., using Java's *for* loops, *variables*, *if*, etc).

In the previous sections, we presented what are software languages, SLE, and metamorphic DSL. We will then present the field of Cloud Computing, which is used as a means in this thesis to distribute language services, allowing to leverage of multiple execution platforms.

3.2 Cloud Computing

Cloud computing is a paradigm that enables access to resources (e.g., machines, storage, etc) on-demand through a network [99]. The principle of cloud computing is to provide a set of shared resources to customers, which on-demand self-serve the resources. Resources are allocated or unallocated in an automatic way to avoid any human intervention. The automation enables rapid scalability to follow the needs in resources of deployed applications which can increase or decrease over time. Resources are monitored to measure the usages of each customer. Cloud computing allows each customer to pay the cloud provider only for the resources they consume, which avoids paying for a static infrastructure dimensioned for peaks of occasional consumption [160].

It exists different kinds of clouds: public, private, or hybrid. The public cloud serves multiple customers whereas the private cloud serves only one. The choice between these two infrastructures has to take multiple concerns into consideration such as security or reliability but also costs since public vs private can be seen as pay-for-what-you-use vs pay-fixed-cost [155]. The hybrid cloud is a composition of public and private clouds that aims to take advantage of both. One usage is that a company uses a part of its infrastructure as a private cloud and sells unused parts as a public cloud, or at the inverse use all of its infrastructure and use an external public cloud to be flexible in case of peaks of consumption.

3.2.1 Service Levels

Cloud providers business offers multiple service levels to their customers: Infrastructure as a service, Platform as a service, Software as a service, and Function as a Service [117]. They differ in the level of abstraction of the resources that can be allocated to the customer, ranging from low level (e.g., storage) to high level (e.g., software).

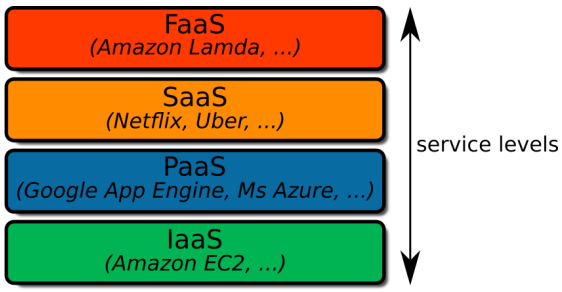


Figure 3.4 – Service Levels in Cloud Computing

Figure 3.4 presents the service levels of Cloud Computing, ranked by their level of abstraction of the resources. Infrastructure as a service (IaaS) is at the lowest level, followed by Platform as a service (PaaS), Software as a service (SaaS), and Function as a Service (FaaS) at the higher level. Although these service levels are perceived as layered, a service level does not necessarily rely on a lower service level and can be relying on an ad hoc solution. For example, an SaaS can be provided on top of servers without relying on a PaaS offering. In the following, we detail the presented level of services.

Infrastructure as a service (IaaS)

IaaS level is the offer that gives the most control to customers. It allocates low-level resources, such as CPU, storage, or network to customers and provides access to physical infrastructure or virtual machines. Customers can manage, configure, and install by themselves everything from the operating system to end-user applications. Amazon EC2 [5] is an example of IaaS.

Platform as a service (PaaS)

PaaS level offers to customers an execution environment (i.e., an operating system) in which they can deploy their applications. In this level, customers can deploy applications but have no access to the underlying infrastructure. Google App Engine [56] and Microsoft Azure [102] are examples of available PaaS.

Software as a service (SaaS)

SaaS level offers ready to use applications to customers. The application runs on a cloud infrastructure that is totally invisible to the customer, who only has access to the application, often a web or a mobile application. To give some examples, we can cite Netflix's video streaming [113] or Uber's food delivery [144] services.

Function as a Service (FaaS)

FaaS level is an approach to serverless computing that offers an execution environment for a particular language [11]. Serverless computing allows customers to upload code and pay only for the time of its execution. In FaaS, the deployment unit is the function. Amazon Lambda [6] is an example of FaaS.

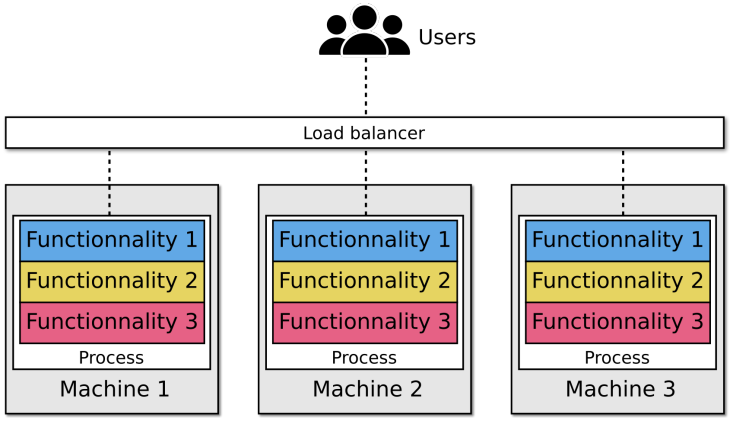
3.2.2 Cloud-Native Application

A cloud-native application is an application designed to exploit the capabilities of cloud computing. It is composed of independently deployable functionalities and manages horizontal scalability (i.e., duplication of functionalities in several instances to cope with the load) [88]. In addition to the scalability, they are other motivations for the adoption of a cloud-native application. The distribution of functionalities provides robustness to the application by avoiding the propagation of failures and allowing automatic recovery. In addition, application delivery is done at the granularity of functionality, which makes it faster and continuous.

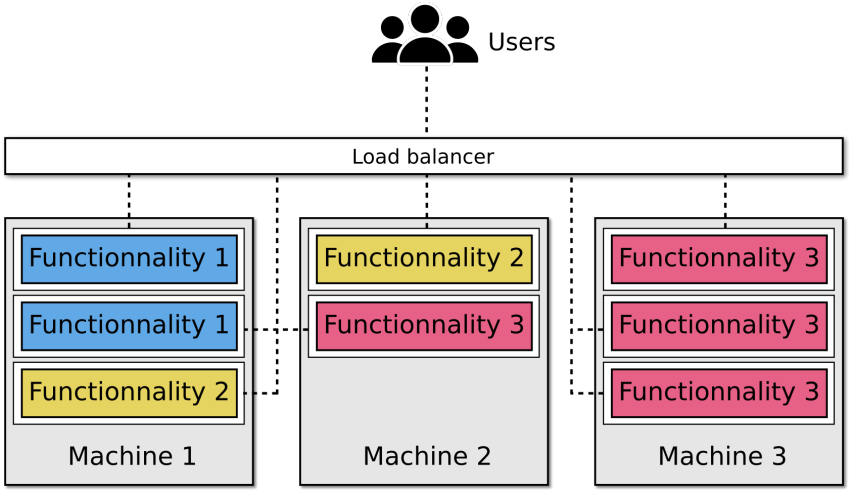
Leymann et al. [96] define the properties a cloud-native application must have to take advantage of the capabilities of a cloud, summarized by the acronym IDEAL (Isolation of state, Distribution, Elasticity, Automated Management, and Loose coupling). This acronym means that :

— the state of the application must be passed to the functionality with each call if possible or be isolated in its own component
— the functionality must be distributable, replicable, and have low coupling between them
— the application is automatically capable of duplicating or deleting functionality to keep up with load increases and capable of restarting functionality in case of an error

Bundling an application in a virtual machine or a container is not enough to call it a cloud-native application since it will not have all the IDEAL properties to fully exploit possibilities of clouds [97]. Such bundled application is easily deployable on a cloud and therefore match the Automated Deployment property but its monolithic nature prevents the others, i.e., the faults from functionalities of the monolith are not isolated and if a functionality is a bottleneck the whole application is duplicated and may lead to duplication of states.



(a) Scaling of monolithic application



(b) Scaling of cloud-native application

Figure 3.5 – Comparison of the scaling of monolithic and cloud-native applications in a cloud

Figure 3.5 illustrates the difference in the use of cloud machines between a monolithic application and a cloud-native application when handling multiple users. In Fig-

ure 3.5a, a monolithic application with 3 functionalities running in a process is deployed in a cloud with three machines. To handle the number of users, the monolithic application is duplicated on the three machines in the cloud. Users of the application access its functionalities through a load balancer that dispatches requests across instances of the application. In Figure 3.5b, a similar application with 3 functionalities but cloud-native is deployed in a cloud with three machines. Each functionality of the application is running in its own process. To handle the number of users, the functionalities are duplicated on the three machines in the cloud. Users of the application access the functionalities through a load balancer that dispatches requests across the functionality instances. The difference with the monolithic application is that each functionality runs in its own process and can be deployed independently. Scaling of both monolithic and cloud-native applications is based on functionality usage, but the monolithic deploys every functionality on each duplication, while cloud-native deploys only the functionality being used. The granularity of cloud-native application scaling enables better utilization of machine resources by closely following user usage.

3.2.3 Microservices

Microservices are a way to build Cloud-Native applications. Microservices architecture is an architectural style allowing a functional decomposition of an application into autonomous services [125]. Microservices follow the principle of single responsibility, i.e., they provide only one functionality. They are isolated from each other and communicate by exchanging messages (e.g. REST [1] requests). Microservices are possibly embedded in software containers and their deployment is automatically managed by a container orchestrator, making a microservices-based application fault-tolerant and managing scalability by specifically replicating the most commonly used microservices. Another benefit of a microservices architecture is the ability to continuously deploy each functionality, as opposed to a monolithic architecture that redeploys the entire application even if the change only concerns one functionality. A new version of a microservice can be deployed to replace the old version without interruption of service, allowing parts of the application to be updated independently of each other and transparently to the user.

Microservices architecture is often viewed as an evolution of Service Oriented Ar-

1. Representational State Transfer

chitecture (SOA) [161]. However, these two architectural styles differ in their approach to service-driven applications. The microservices architecture style can be resume as "dumb pipes and smart endpoint" [95] while SOA style as "simple services and smart pipes" [21]. SOA connects services with an Enterprise Service Bus (ESB). ESB is in charge of dispatching messages that comprise performing complex operations such as transforming messages into another format and orchestration of services. In microservices architecture style, there is no central component to connect services and the service coordination is choreography-oriented (i.e., each microservice implements its own subpart of the global coordination of the application).

The straightforward approach to build microservices-based application is by hand using frameworks like Spring [148], MicroProfile [40], Micronaut [26] or Quarkus [65]. However languages promote microservices as first class entitiy, e.g., programming language [58], aspect-oriented language [62], agent-oriented language [158] or modeling language [36].

Microservitization

The development of applications based on microservices architecture involves the process of microservitization which promotes microservices as first-class entities in the development activities. Microservitization is defined by Hassan, Ali, and Bahsoon [62] as "a shift towards transforming services/components into microservices — more fine-grained and autonomic services that isolate fine-grained business functionalities by boundaries and interact through standardised interfaces". The microservitization process is applicable for both the creation of new microservices-based applications (greenfield approach) and the migration of existing monolithic applications to microservices-based applications (brownfield approach) [63]. Both greenfield and brownfield microservitization approaches entail the isolation of functionalities and communication through interfaces, the goal being to get the properties of cloud-native applications.

Determining the good granularity for microservices is a major concern in developing microservices-based application [64]. The granularity of microservices is a trade-off to define stable boundaries that requires a good understanding of the domain of the application by developers. Defining too small microservices makes the application having more microservices which increases the deployment cost and which impacts performances since the number of exchanges between microservices will be more numerous. It makes the application harder to understand and reason about which in-

crease its maintenance cost. On the opposite, too coarse-grained microservices make it harder to achieve the cloud-native application properties. Microservices with bigger scope make applications less flexible in the scalability management since it relies on replication of the functionalities. It also reduces the fault isolation of functionalities that impacts the robustness of the whole application.

3.3 Summary

In this chapter, we presented the domains we use as background in this thesis. We defined Software Language Engineering which is is a computing discipline for designing and implementing software languages and introduced the notion of metamorphic DSL, i.e., a DSL with multiple shapes. We also presented the paradigm of Cloud Computing which is used to leverage a cluster of machines to provide on-demand access to their resources. We described what Cloud-Native applications are, which are applications designed to leverage the capabilities of a cloud and the Microservices approach to Cloud-Native applications. In the following chapter, we present the state of the art in desktop and web-based Integrated Development Environments, their use of the cloud, and discuss the limitation of their flexibility with regard to the challenges presented in Chapter 2.

CHAPTER 4

STATE OF THE ART

In this chapter, we present the state of the art of IDEs in regards to the challenges presented in Chapter 2. We first review the different kinds of IDEs, i.e., desktop IDEs, web-based IDEs and cloud-based IDEs (Section 4.1). We start by presenting the desktop IDEs, their component-based architectures and the rise of language protocols used for communication between IDEs and language services. We then present the web-based IDEs (Section 4.1.2) We last present cloud-based IDEs and there usage of cloud infrastructure to manage user workspaces (Section 4.1.3). We secondly discuss of the flexibility limitations of IDEs regarding the challenges to be addressed (Section 4.2).

4.1 Integrated Development Environment

IDEs are software dedicated to the development of software. Development of software involves many activities that includes in addition to programming, the designing, debugging, testing, the organization of development projects, collaboration with other developers, etc. IDEs assist software developers by bringing together a set of development tools and by providing a unified interface for ease of use. We use the term of *development tool* to refer to the software components used by software developers that support their activities. Development tools include editors, compilers, or debuggers but also development tools that are not directly related to languages, e.g., version control systems. Language constructs are manipulated through the language services provided by these tools. For example, in program development, textual documents are parsed to obtain programs, which are then validated before being interpreted or compiled at the end.

Mastering individually the many tools of a development environment may be hard. Indeed without an IDE, software developers have to install the development tools, learn how to configure them and how to use them individually. Moreover, in a devel-

opment environment the tools are not used separately, they form a tool chain. Software developers have to understand how to connect them to have a usable development environment.

IDEs mitigate these difficulties by providing an environment where all the development tools are in one place that avoids to install them one by one. Development tools in IDE are already connected in a preconfigured tool chain that allows an immediate usage of them. Tools are also accessible in a unified interface that hides the specificities of each tool to ease the usage. IDEs aim to reduce the cost of software development by easing the experience of the developer.

We give in the following subsections an overview of the existing kinds of IDE and the opportunities they offers. First, we present desktop IDEs and the emergence of language protocols.

Second, we present the Web-based IDEs which are the result of the trend to move software from desktop applications to web-based applications for development environments [79]. They provide an online experience for software development through a web browser. The intent behind the move from desktop UI to an in-browser interface was to avoid the cost of installing and configuring development environments by providing ready-to-use solutions to the users.

Last, we present Cloud-based IDEs which adopted the SaaS model by taking the opportunities offered by cloud computing.

4.1.1 Desktop IDE

We review in this section the architecture used to integrate development tools in desktop IDEs (i.e., IDEs running on the machine of the software developer) and present afterwards the language protocols that emerged from these architectures and which are used to externalize language services from IDEs.

Component-based architecture

Although IDEs can be dedicated to a single language, the 10 most searched IDEs on Google [141] support multiple languages, i.e., Visual Studio [106], Eclipse [38], Android Studio [55], Visual Studio Code [107], PyCharm [74], IntelliJ [73], Netbeans [8], Xcode [10], Atom [53] and Sublim Text [71]. IDEs supporting multiple languages are also a popular basis for language workbenches, as can be seen in [42] where 6 of the 10

languages workbenches compared are based on Eclipse or IntelliJ.

The support for multiple development tools in these IDEs is allowed by component-based architectures. The extension of the IDE to add a language service consists of providing components implementing a set of APIs.

The IDE Eclipse relies on the OSGi standard [4] for its modularity. Equinox, the Eclipse's implementation of the OSGi standard, is a Java framework running on top of a JVM which allows to implement an application following a dynamic component model [137]. OSGi components are Jar archives containing Java classes called *bundles*. Each bundle declares its dependencies to other bundles and the set of services it requires and offers, which are implemented by Java classes. OSGi manages the life cycle of the bundles dynamically, i.e., bundles are installable on the fly without a reboot. At the installation of a bundle, its services are registered in a central registry that is used to satisfy the provide/require dependencies of all services. Eclipse uses OSGi's dynamic bundle management, but although OSGi provides a service registry, Eclipse uses the *plugin* concept. Eclipse plugins act similarly to OSGi services in that they define extension points to contribute, which declare Java interfaces that other plugins can implement to contribute the extension point.

Leveraging from the experiences of Eclipse, Visual Studio Code is a component-based IDE that improves the extensibility design and opens up to new usage scenarios. Visual Studio Code was designed with performance in mind. Eclipse startup time suffers from the lack of isolation of its plugins. Visual Studio Code proposes to run its extensions in separate processes. This way an extension taking too much time does not impact the whole IDE. The time the IDE is ready to serve the user is not increased by slow running extensions. Visual Code Studio also allows running extensions remotely. The core component of the IDE communicates with an agent deployed remotely. The extension and an agent are deployed remotely. The agent is connected to the core component deployed on the user machine and acts as a gateway with the remote extension. This architecture allows scenarios like working with multiple file systems, with containers or SSH Boxes.

Visual Studio Code faced the problem of integrating the support for the many existing languages. Their approach was to decouple the IDE from the language implementation with a communication protocol. The proposed to have a language agnostic IDE that accesses language services provided by dedicated language servers running at the side of the IDE for each language via a language protocol, i.e., a standardized

communication protocol for using the language services. In the next subsection, we introduce the *IDE portability problem* and review the language protocols that standardize the communication between IDE and language servers.

Language Protocols

The support of multiple languages has led IDEs to be language-agnostics. For the integration of each language, a generic API must be implemented to connect the language services in the IDE, therefore this adaptation code is specific to both the host IDE and the integrated language. IDE maintainers must make an effort to support multiple languages and this effort has to be repeated in other IDEs since the APIs differ, even for the same languages. Keidel, Pfeiffer, and Erdweg [80] identified this issue as the *IDE portability problem*. Language designers also face the same problem when creating new languages. Indeed, to increase the adoption of a new language, language designers must do the integration work for many IDEs, which requires implementing the many APIs of the targeted IDEs. The global effort to get n IDEs to support m languages is therefore n x m implementations of integration logics.

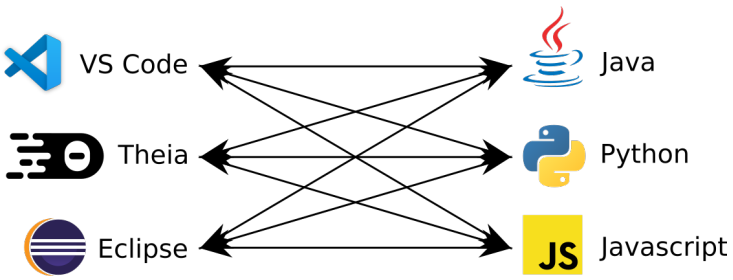


Figure 4.1 – IDE portability problem

Figure 4.1 illustrates the *IDE portability problem* with the IDEs Visual Studio Code, Theia, and Eclipse at the left side and the languages Java, Python, and Javascript at the right side. Each arrow between an IDE and a language represents an adaptation effort to integrate both sides. Each of the three IDE has to integrate with the three languages, which represent in total an effort of nine integrations.

The *IDE portability problem* due to the multiplication of IDEs and language has led to the emergence of language protocols that standardize the communication between IDEs and language servers providing the language services. They try to solve the problem by decoupling language-agnostic clients (e.g., IDEs) from language-specific

servers that provide language services thanks to standardized inter-processes communication. The clients and servers run in separate but linked processes through a communication channel and implement the same protocol specifications, i.e., they agree on the format of the messages exchanged and on their sequence order. Language protocols make IDEs interoperable with language servers. This means that the implementation of a protocol in an IDE makes any existing and future language servers implementing the same protocol available "for free", and conversely, language designers have access "for free" to any IDE implementing the same protocol. Language protocols standardize communication by defining interfaces for a set of language services for specific activities in various domains such as program editing, program debugging, program building, etc.

The Language Server Protocol (LSP) [104] is a language protocol initiated by Microsoft that specifies communication for language services for program editing activities (e.g., content assist, goto definition, program validation, etc). LSP provides interoperability between development environments and language servers providing language services. It is a standardized protocol that defines the format of data exchanged, which is based on text positions in addition to the specificities of each language service. It is a protocol that uses the remote procedure call paradigm. It is based on the JSON-RPC [108] which defines a notification, request, and response format using JSON. LSP was first introduced in Microsoft's multi-language IDE Visual Studio Code to solve the *IDE portability problem* but it gained popularity and has been implemented, to date, by 24 clients and supported by 79 languages [129]. The languages supporting LSP are not restricted to GPLs like Java or Python since it has been applied to modeling language based on the EMF [122] and has been used for textual DSLs to facilitate their integration into multiple editors [17]. Although the objective of LSP is to support the integration of any language, it is limited to textual languages. However, a workaround solution based on an intermediate representation in textual format for a graphical representation has been proposed [123].

Figure 4.2 illustrates the benefit of using LSP with the same IDEs and language presented in figure 4.1 which illustrated the *IDE portability problem*. The support for each of the three languages is implemented by language servers implementing LSP and each of the three IDE implements the same protocol, that represent in total an effort of 6 protocol implementations.

Software debugging has always been a major concern in software engineering. It

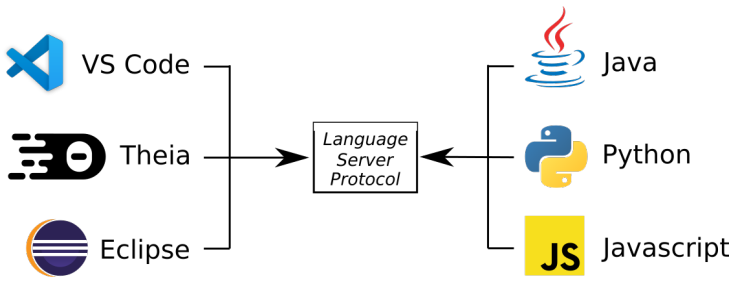


Figure 4.2 – IDEs and languages integration with the Language Server Protocol

requires some form of language protocol to allow an external tool to monitor the execution of programs, whether it is an interpreted program or a compiled language (through the instrumentation of the code). IDEs such as Eclipse provide generic APIs to connect debugged software to their debugging interface. Tools like Gemoc [16] benefits of this infrastructure to implement over a debugging protocol for model execution.

In parallel with LSP, Microsoft has developed the Debug Adapter Protocol (DAP) [103] to decouple language-agnostics clients from language-specific debuggers. This is a protocol complementary to LSP that standardizes communications between the client and the server for debugging activities (add breakpoint, step into, drop frame, etc). It allows to control stack-based execution of a program and to inspect its internal state. The protocol is based on the remote procedure call paradigm. It specifies the format of the messages exchanged, which is based on the document identifier and the text positions.

Build Server Protocol (BSP) [126] is a protocol to address the *IDE portability problem* between clients and build tools (e.g., sbt, Graddle), where clients can be language servers or directly IDEs. Like LSP, BSP relies on JSON-RPC. BSP allows a client to launch tasks and follow progress through notifications. The tasks include compilation, testing, and execution.

Jupyter Notebook [76] is a web-based environment for editing interactive documents made up of cells that contain chunks of program, each of which is independently evaluable. Jupyter Notebook supports multiple languages through the concept of *kernel*, which are language-specific interpreters built as Read-Print-Eval-Loop (REPL). Jupyter Notebook uses the Wire Protocol to establish a communication between the frontend and the *kernel* with request/reply messages.

Keidel, Pfeiffer, and Erdweg [80] propose a solution to the *IDE portability problem* called Monto. Inspired by the domain of compilers that face a similar portability prob-

lem and which use intermediate representations to write compilers for m language targeting n architectures, Keidel, Pfeiffer, and Erdweg [80] propose to use intermediate representations common to IDE clients and language services to enable communication agnostic of any IDE or language. Monto makes IDEs clients interoperable with language services and supports their distribution as microservices.

Language protocols solve the *IDE portability problem* and standardize communications for language services in many domains, such as program editing, program debugging, program building, etc. In the following section, we discuss why component-based architecture and language protocols from the state of the art are not enough flexible to address the challenges presented in section 2.2.

4.1.2 Web-based IDE

Web technologies offer the possibility to have a development environment accessible remotely and ready to use thanks to a web browser. This simplifies the start of a development project and eases the collaboration between users since only a browser is needed. Web-based development environments come in a variety of forms, such as code editors that can be integrated into web pages, notebooks for data scientists, full web-based IDEs, or code playgrounds that emphasize collaboration over snippets of code.

Code editors are the simplest form of a web development environment. They are web components that can be integrated into a web page and provide few language services to assist in program writing. As web component, they are used in more complete web-based development environments, e.g., the editor ACE [2] is used in the web-based IDE Cloud9 [7], the editor CodeMirror [24] is used in the IDE Codeanywhere [23] and Jupyter Notebook [76], the editor Monaco [105] is used in the web-based IDEs Visual Studio Code [107] and Eclipse Theia [143].

Code playgrounds are a new trend in developing environments emphasizing interactivity and allowing collaborative editing. It exists web-based environments that allow publishing code snippets to share or review source code such as Pastebin.com or GitHub Gist [131] but they differ from code playgrounds since they are not 'language award', i.e., they do not provide language services. JSFiddle [75] and CodePen [25] are code playgrounds that provide collaborative program editor for web page development, complemented by a rendering view and a console. The source code (HTML,

CSS, and Javascript) can be executed to view the resulting web page. Repl.it [120] is a code playground supporting dozens of programming languages. It integrates a file browser, a program editor, a console, and a shell. Programs can be executed and the result is displayed in a console.

Web-based IDEs such as Visual Studio Code, Cloud9, Eclipse Orion [39] or Eclipse Theia are web-based development environments close to desktop IDEs in terms of functionalities and user experience. They are integration frameworks that, in addition to program editor aggregate project management, version control system, debugger, etc. Web-based IDEs are referred to as cloud IDE when deployed in a cloud. They exploit the horizontal scalability of clouds, i.e., new instances of the IDE are created for each new user. Some web-based IDEs like Cloud9, Eclipse Che [41] or Codeanywhere [23] also exploit the cloud isolation capability to manage user workspaces. The boundary between web-based and desktop IDEs can be fuzzy since a web-based application can be run locally. This is the case for example with Visual Studio Code which is an IDE built with web technologies but used mainly as a desktop IDE. It uses Electron [47], a framework using the Chromium rendering engine that turns web page to graphical representation and using the Node.js Javascript runtime to run Javascript programs. Electron allows the execution of both the backend and frontend of a web application, making it a desktop application for the user. Eclipse Theia is another example of IDE built with web technologies that can be used as a desktop application using Electron. Although they are packaged as a desktop application, they are not limited to the desktop since they have an architecture designed for the web (i.e., they have a frontend and a backend). Microsoft provide Visual Studio Code as web-based IDE in GitHub Codespaces [54] and Eclipse Theia is usable as web-based IDE in Gitpod [142].

The plugin architecture is also used by web-based IDEs to extend both frontend and backend. Visual Studio Code is composed of a UI part (i.e., the frontend that runs in a web browser) and a Workspace part (i.e., the backend). Visual Studio Code has a plugin architecture that provides a set of contribution points to extend both the UI and the Workspace parts. Plugins must declare the extension point to which they contribute. The distinction between UI and Workspace allows running plugins on the right machine when Visual Studio Code is deployed in a remote development setup. Similarly, Eclipse Theia is based on a plugin architecture but also offers an alternative extension mechanism based on dependency injection. These *extensions* are NPM packages composed of Javascript classes that can declare injectable contribution interfaces. The

difference is that *extensions* are used to build Javascript applications running in the same process while plugins are isolated in their own process (or in a worker for the frontend part). Since plugins run in process, they are isolated which allows them to be loaded at runtime but forces them to communicate through well-defined APIs. Maróti et al. [98] proposed a plugin architecture for WebGME to extend both frontend and backend. Plugins interact with the model through a Javascript API that allows them to be deployed on the frontend or backend. To support plugins written in a language other than Javascript, they also provide a REST API to interact with the model.

Notebooks like Apache Zeppelin [9] or Jupyter Notebook [76] are also a form of a web-based development environment. Notebooks are interactive documents containing parts of natural language and cells of source code. The cells of code can be executed to produce text or graphical outputs that can be interactive. Each cell shares the same execution context but is executed independently following the Read-Eval-Print-Loop model. The combination of natural language, snippets of executable code, and interactive output makes notebook environments suitable for education and data exploration purposes.

Web-based collaborative modeling environment such as WebGME [98] or AtomPM [134] has been investigated. Sirius Web [115] is a web-based modeling environment with collaborative capabilities that allows using in web browser modelers defined with Sirius [147], an Eclipse-based technology to design and implement modeling languages with graphical syntax. MDEForge [13] is a web-based modeling environment centered on modeling repository. It focuses on two aspects: model storage and model transformation execution in a cloud. Distil [20] allows to describe MDE services and generate web environment. Zweihoff, Naujokat, and Steffen [162] introduced Pyro, a graphical modeling environment with collaborative support. They propose a generative approach to produce the frontend and backend of the web modeling environment from specifications.

4.1.3 Cloud-based IDE

In this section, we overview how cloud-based IDEs leverage the capabilities of clouds to manage user workspaces. The user workspace has a specific configuration for each development project. This includes source code, dependencies, runtime, language services, etc. Cloud-based IDEs provide remote access to development environments

that are provisioned on-demand in the cloud. They complete web-based IDEs by supporting a reproducible development environment that allows automatic workspace setup. Fylaktopoulos et al. [52] define cloud-based development as "the complete transfer of developer workspace into the cloud. The developer's environment is a combination of the programming IDE, the local build system, the local runtime, the connections between these components and their dependencies".

Clouds facilitate the management of development environments by allowing on-demand workspace. Each user can have its own isolated workspace deployed in a cloud thanks to virtualization (e.g., virtual machines or containers). Virtualization brings independence to execution platforms, enabling reproducible workspaces. It removes the need for user involvement in workspace configuration, and thus provides a ready-to-use development environment.

Container-based solutions running on clouds complement web-based IDEs. Eclipse Che [41] proposes to use Docker containers to deploy whole workspaces on a Kubernetes cluster. Workspaces are configured through YAML files named "devfile". They contain for example the location of a source code repository, the language server to be deployed or the IDE to be used (e.g., Eclipse Theia). Eclipse Che read these "devfiles" to produce the containers to be deployed on the Kubernetes cluster. As text format, the "devfile" allows the versioning and the sharing of workspace configuration. Amazon provides Cloud9 [7] for AWS which is a web-based IDE that also uses Docker containers to deploy workspaces in Amazon's clusters. Codeanywhere [23], which relies on DevBoxes [33], is another example of web-based IDE using containers to manage workspaces. GitPod [142] is an online service integrated with GitHub, Bitbucket, and GitLab that promotes the *dev-environment-as-code*, i.e., the idea that development environments are complex things that have to be described in an editable, shareable, and versionable configuration file. It uses Docker to containerize whole workspaces, containing at least a runtime environment with source code and a web-based IDE. Microsoft proposes Codespaces [54], an online service interfaced with GitHub that provides a container-based solution for workspaces. They use JSON files to configure the content of the workspaces.

The capabilities of the cloud provide web-based IDEs with on-demand and reproducible workspaces, that are customizable with configuration files. We discuss in the next section why the cloud-based IDEs in the state of the art do not fully address the challenges presented in section 2.2.

4.2 Flexibility in IDEs

In this section, we discuss on the aforementioned approaches to flexibility in IDEs and identify what is missing to address the challenges presented in section 2.2. We first discuss on the support of modularity in IDEs and then discuss on their capacity to distribute language services.

4.2.1 Modularity

IDEs are modular thanks to component-based architectures and the use of language protocols. Component-based architectures allow support for modular development tools. Components are reified through the concept of plugins, which define how IDEs can be modularized. Plugins declare extension points with API, which are implemented by the language services of the new languages integrated into the IDE. Although this is an architecture that allows modularization, it imposes limits on the flexibility of the IDE with regard to the challenges of combining technological stacks and leveraging multiple execution platforms.

Component-based architecture does not allow the combination of multiple technological stacks for one language. The integration of a language into the IDE is done through the implementation of APIs corresponding to the language services supported by the IDE (or a subset). The implementation of the language is done in the technological stack chosen by the language designer. A language could be implemented multiple times in different technological stacks, but each implementation would be isolated from the others. A language is integrated into the IDE framework as one set of cohesive plugins implemented in the same technological stack and despite the API abstraction layer provided by the plugin that allows modularization, this approach to the modularization is not flexible enough to support multiple technological stacks. Indeed, the plugin approach allows language services to be implemented in any technological stacks, but the use of language services from different technological stacks requires the addition of a synchronization mechanism for the language concepts manipulated by the language services.

Language protocols solve the *IDE portability problem*, i.e., they provide interoperability between IDE clients and language servers by standardizing their communication. Protocols allow language services to be isolated from IDEs and give total independence in their implementation from technologies used in the IDE. Each protocol

is dedicated to a given activity (e.g., editing, debugging, etc) making them support a well-defined set of language services. For a given activity, a language protocol has a fixed specification that defines the language services and the formats of exchange messages, formats that are tied to particular concrete syntaxes. Fixed specifications solve the *IDE portability problem* but this has led to the multiplication of protocol to cover new activities. Existing language protocols are not enough flexible to allow modularity for any language. They limit the modularity of IDEs since only language services specified by a protocol can be integrated into an IDE. Keidel, Pfeiffer, and Erdweg [80] proposed a solution based on standardized intermediate representations allowing the distribution of language services as microservices. However, they specified only three language services, which limit its usability.

Language protocols are technology agnostics, which allow technological stacks to be interoperable. However, language protocols were first designed to communicate with local processes providing language services for an IDE running in another process. It means that they are designed with a single-source model, which prohibits the use of multiple technological stacks to implement a language. Indeed using language services from multiple technological stacks implementing the same language implies multiple source models and therefore requires additional synchronization mechanisms for the models.

4.2.2 Distribution

Component-based architecture does not allow to leverage of multiple execution platforms for the language services. Although the component-based architecture allows modularity, the resulting IDE is a monolithic software. This means that all language services run on the same execution platform. To leverage multiple execution platforms, modularizing an IDE require isolation of the language services and the ability to distribute them.

In web-based IDEs, the development is driven by the need of providing ready-to-use development environments. The state of the art web-based IDEs make development environments accessible through web browsers and rely on client/server architecture to externalize the user interface and discharge the language users from the responsibility of set up the development environment. However, the server part of web-based IDEs that provide language services uses component-based architecture

like desktop IDEs that allows the same modularity, which is limited by its monolithic nature, i.e., leveraging multiple executions platforms require the distribution of the language services.

Cloud-based IDEs use clouds to manage user workspaces. They rely on container technologies to provide a runtime environment for language services. Workspace containerization addresses the need to provide a ready-to-use development environment for each user on demand. Containers help to scale IDEs in function of the number of users. Each user has its own deployed container containing its workspace with all the services he needs and its source code. Although containers allow deployment on multiple execution platforms, cloud-based IDEs package the whole development environment into one container. This means that all language services are deployed as a whole on the same execution platform. While cloud allows deploying on multiple execution platforms, cloud-based IDEs don't fully exploit this possibility. The containerization of workspace allows configuring cloud-based IDE by specifying which language servers will be embedded in the container, which allows only coarse-grained modularity. The language services can't be deployed individually across multiple execution platforms, so their needs can't be met individually. Language services should be deployable individually, and since they are provided by a single container, it is necessary to modularize them into dedicated containers for distribution across multiple execution platforms.

4.3 Summary

In this chapter, we first presented the modularity of the IDEs, which is enabled by component-based architecture and is used to support multiple languages. We also presented languages protocols, which complete the modularity brought by component-based architecture to solve the portability problem by making IDE clients interoperable with language servers. We then introduced web-based IDEs and presented how they allow extensibility. We also reviewed the usage of cloud capabilities by cloud-based IDEs to provide on-demand and ready-to-use workspaces.

Regarding the challenge of combining multiple technological stacks to benefit from their strength to implement a language and enable to seamlessly switch between language services from different technological stacks, we identified that IDEs lack of a synchronization mechanism for multiple technological stacks.

Both component-based architecture and language servers are monolithic, which prevents leveraging execution platforms to best fit the needs of language services. To address this challenge, language services have to be distributed on the execution platforms, which is not possible with the current usage of cloud capabilities by cloud-based IDEs that deploy the whole user workspace in a single container.

In the next part, we will present our contributions to the flexibility of IDEs. We first detail our approach to combine multiple technological stacks to allow language designers to benefit of their specific strengths when implementing language and to allow language users to switch between language services implemented in these stacks. The approach consists of a communication bus used to synchronize the multiple implementations of a language in different technological stacks. We secondly detail our approach to modularize and distribute language services to leverage multiple execution platforms. It is a generative approach to implement language services as microservices and to drive their safe deployment.

PART II

Contributions

CHAPTER 5

THESIS OVERVIEW

In this chapter, we present the vision we defend and the global view of this thesis. We start by presenting our vision regarding the challenges of flexibility in IDEs. We then give an overview of our contributions which are detailed in the two following chapters.

5.1 Vision

We identified in Chapter 2 two challenges regarding the IDEs flexibility, i.e., the challenge of using multiple technological stacks to implement and use languages and the challenge of leveraging execution platforms for language services.

We showed in Chapter 4 that IDEs are not enough flexible to address these challenges.

Our vision is that language services are central to IDEs and that they should not be all implemented in the same technological stack and not be all deployed on the same execution platform. We claim that combining multiple technological stacks bring more flexibility to language designers and ease the activities of the language users. We also claim that distributing language services provides more flexibility to language users and gives them the possibility to improve the performances of language services.

From the challenges of IDEs flexibility we formulate the two research questions that drove this thesis:

— RQ1: How to combine the strengths of multiple technological stacks in the language design and implementation?
— RQ2: How to leverage the available execution platforms to best fit the needs of language services?

5.2 Overview

In this section, we draw the big picture of this thesis. We present how we tackled the identified challenges through the two research questions of the previous section and show how the two contributions of the thesis are related.

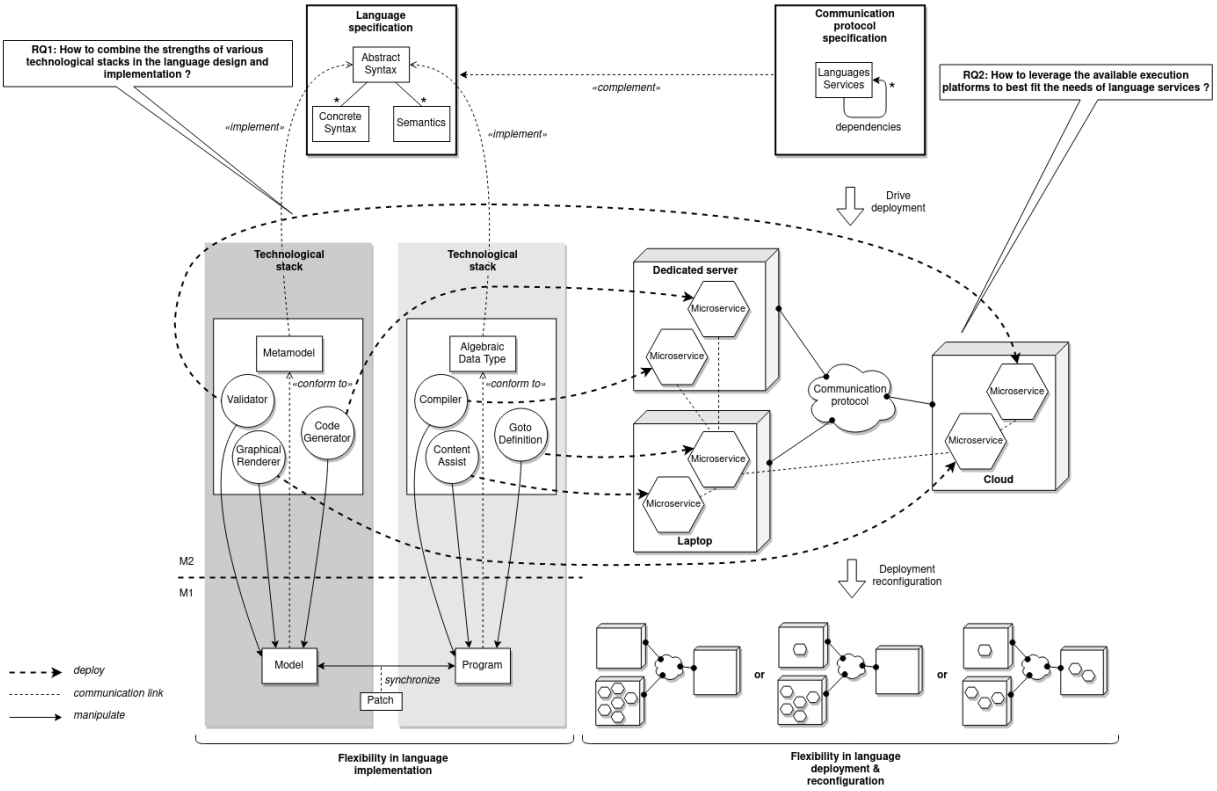


Figure 5.1 – Thesis overview

Figure 5.1 presents an overview of our two complementary contributions to two different aspects of the IDE flexibility. The first contribution brings flexibility in language implementation and is depicted on the left side of the figure. The second contribution brings flexibility in language deployment and reconfiguration and is depicted on the right side of the figure.

The first aspect of the IDE flexibility was explored by answering the research question *RQ1: How to combine the strengths of multiple technological stacks in the language design and implementation?* An abstract syntax, concretes syntaxes and semantics are the main components of language specifications and are used to implement languages in technological stacks. We consider the technological stack as the technological means for implementing a language (we precise the term *technological stack* in section 6.1).

For example, a same language specification can be implemented with a Metamodel or with an Algebraic Data Type. Both implementations define language services and, depending of the technological stack, ones manipulate a Model conform to a Metamodel and others manipulate a Program conform to an Algebraic Data Type. With our contribution presented in Chapter 6 we propose to synchronize the Model with the Program (both being equivalent, i.e., they represent the same thing). We propose in this contribution to enable synchronization by producing a Patch when a language service manipulates the Model or the Program. The Patch represents the changes after the manipulation and is applied to the other Model or Program. The goal is to keep the equivalence between the Model and the Program over time, which gives the flexibility to language users to use language services from different implementations of the same language regardless of their technological stack.

The second aspect of the IDE flexibility was explored by answering the research question *RQ2: How to leverage the available execution platforms to best fit the needs of language services?* We propose with our contribution presented in Chapter 7 to distribute language services across the different execution platforms as microservices that exchange messages through a communication protocol. To handle the distributed context for a language, we complement language specification with a communication protocol specification that describes language services and their dependencies. We use these specifications to generate microservices and drive their safe deployment. The goal is to give the flexibility to dynamically configure the deployment of language services across multiple execution platforms.

As previously mentioned, the two contributions of this thesis cover two aspects of the IDE flexibility. While the second contribution is a systematic approach to automate the modularization and the distribution of language services, it is limited to one technological stack. The first contribution complements the second contribution by enabling the use of multiple technological stacks.

We present our contributions to the flexibility of IDEs in the next chapters of this section. The Chapter 6 details our approach to design, implement and use a language by combining multiple technological stacks. The chapter 7 details our approach to the distribution of language services.

CHAPTER 6

SHAPE-DIVERSE DSL

In this chapter, we present our first contribution, which is an approach to shape-diverse DSLs that consists of combining technological stacks for the implementation and use of DSLs. We start by introducing the context of this contribution (Section 6.1) and by describing a motivating example (Section 6.2). We then define the notion of shape-diverse DSL (Section 6.3) before presenting PRISM*, our prototype approach to connect multiple technological stacks (Section 6.4) and detailing our approach by using an example with the FSM language implemented in multiple technological stacks. (Section 6.5). We end the chapter with our conclusions on* PRISM *(Section 6.6). This chapter is based on our SLE'18 [30] publication.*

6.1 Technological Stacks

One of the first steps in designing a new DSL is to choose which technological stack will be used to engineer it. We define a technological stack as the technological means for implementing a language. This includes language workbenches as well as programming languages and ontology languages, to name a few. The notion of technological stack is orthogonal to the distinction between technological spaces (e.g., grammarware [82], modelware [89]); between graphical and textual syntax; between internal, embedded, and external DSLs, etc. For instance, we consider Rascal [84] and Spoofax [78] as two distinct technological stacks within the broader technological space of grammarware and meta-programming; EMF [132] and UML [50] (using Profiles [127]) as two distinct technological stacks within the broader technological space of modelware. Technological stacks usually come with their own meta-languages for expressing the various aspects of a DSL: abstract syntax, concrete syntax, static and execution semantics, tools, etc. As implementation techniques radically differ from one technological stack to another, this initial design choice commits the development of a DSL in

a set direction that can hardly be reconsidered later on.

From the language designer's point of view, however, every technological stack has its own strengths. The ecosystem around EMF is strong in the definition of user-friendly editors and persistence frameworks for large models, while the Rascal environment is better in the definition of interpreters and refactoring tools. The benefits of various technological stacks are also visible from the language users' point of view. While domain experts may prefer to manipulate domain concepts through a dedicated syntax, advanced users (e.g., system integrators) may favor the flexibility of a fluent API in their favorite programming language to manipulate the very same constructs.

However, it is currently hard to combine multiple technological stacks to engineer a DSL, since they may be distant conceptually and technically, and making them interoperable requires implementing adaptation layers between technological stacks.

In this chapter, we present PRISM, a framework for combining multiple technological stacks. PRISM enables the engineering and the use of DSL with multiple shapes. We first motivate the usefulness of having a DSL implemented in multiple technological stacks using an example. We then introduce the notion of shape-diverse DSL, i.e., a DSL engineered in multiple technological stacks. We detail afterwards PRISM, which mainly consists of a communication bus relying on a publish/subscribe pattern and a DSL-agnostic formalism to express changes on models. We conclude the chapter with an example of shape-diverse DSL with the Finite-State Machine (FSM) language implemented in three technological stacks. We demonstrate that it is possible, for a language implemented into multiple technological stacks, to switch seamlessly between language services from the different technological stacks.

6.2 Motivating Example

The language designer can choose from many technological stacks when coming to implementing a language. This choice is based on the strength of the selected technological stack in a specific aspect of language engineering.

Let us consider a simple FSM language as a motivating example. An FSM is an abstract machine reacting to events and is used for example to describe the behavior of systems. The main concepts of the FSM language are Machine, State, Transition, and Event. Machine, which is the root concept, is composed of States and Transitions. The Transition connects a source State with a target State and is triggered by an Event.

States have a name and one of them is declared as the initial State of the Machine. To complete the motivating example let also consider EMF, Rascal, and Java Fluent API as possible technological stacks to implement the FSM language. As depicted in Figure 6.1, one would like to combine the strengths of these multiple technological stacks to engineer the FSM language. Rascal could be used to develop its interpreter, a set of refactoring tools (e.g., state collapsing and minimization), and a textual editor; EMF to develop a graphical animator for debugging FSM models and a persistence layer; Java to offer a fluent API for advanced users who focus on its integration with other system concerns.

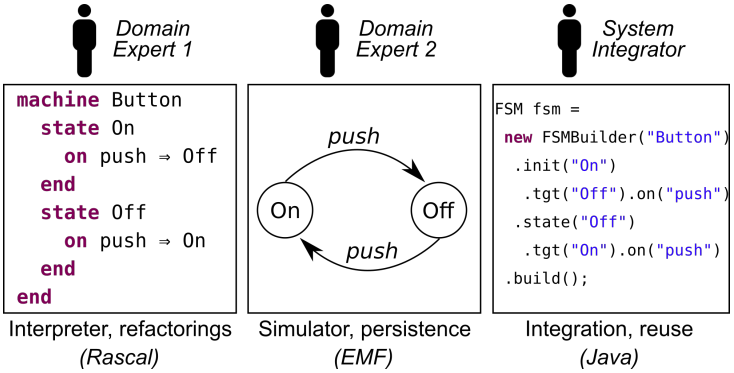


Figure 6.1 – Three incarnations of the same FSM model in three technological stacks: different representations and tools for different users and tasks.

Using today's techniques, it is possible to define the same FSM language in these three technological stacks separately. It is not possible, however, to apply the tools of a given technological stack on the models or programs created in another technological stack—for instance, animating an FSM model written in EMF using the Rascal interpreter, or synchronizing a textual FSM model in Rascal with its equivalent incarnation as a Java AST. Achieving this goal requires to *efficiently synchronize the diverse representations of the same model in different technological stacks*; for instance to let the FSM interpreter written in Rascal update its own representation of an FSM model after each execution step and synchronize it with the representation of the same model in EMF for animation purposes.

We envision a language engineering approach enabling (i) language designers to combine tools from multiple technological stacks to engineer diverse shapes for a single DSL and (ii) language users to manipulate language constructs in the most appropriate shape.

6.3 Shape-Diverse DSLs

The cornerstone artifact defining a DSL in any technological stack is its abstract syntax. The way abstract syntax is expressed differs drastically from one technological stack to another: GEMOC [16] and Xtext [15] use Ecore metamodels [132], MPS uses *concepts* [149], Rascal [84] uses Algebraic Data Types (ADT), etc. Language embedding techniques, on the other hand, use the constructs of a host language to materialize the constructs of a DSL in the host language itself (e.g., a set of Java classes). Concrete models are then built as instances of the corresponding abstract syntax formalism: Ecore models, ADT values, Java ASTs, etc. The tools defined within a particular technological stack (an interpreter in Rascal, an editor in EMF) manipulate models in the corresponding formalism (respectively, ADT values and Ecore models). These formalisms radically differ in many ways [83]: object-oriented vs. functional, graphs vs. trees, mutable vs immutable datatypes, cross-references vs. symbolic names, etc. As technological stacks are developed by independent groups of people and rely on different underlying theories, it is neither possible nor desirable to establish a common foundation upon which all technological stacks would agree.

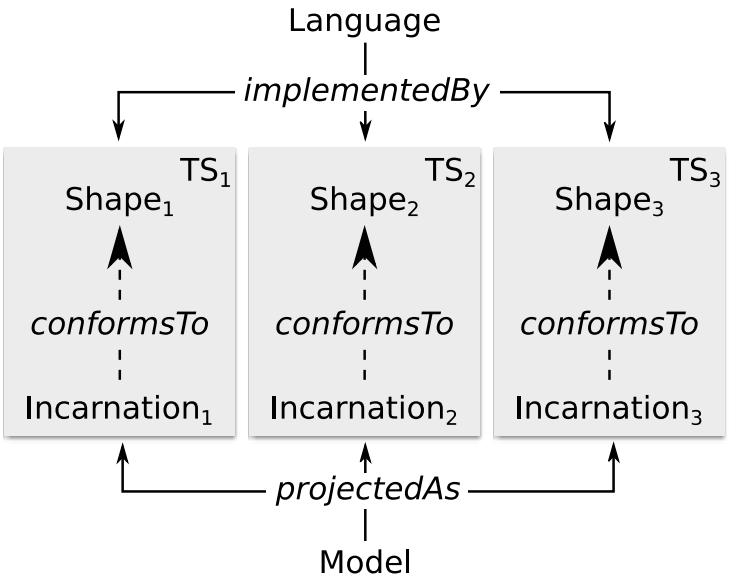


Figure 6.2 – Languages are implemented as shapes in technological stacks and models are projected as incarnations conforming to the shapes.

Figure 6.2 gives an overview of the concept of shape-diverse DSL and the terminology we use throughout the chapter. A shape-diverse DSL $\mathcal{L}$ (e.g., the FSM language

of Figure 6.1) is a language that is implemented in multiple technological stacks $\mathcal{TS}_i$ through multiple shapes $\mathcal{S}_i$. As mentioned earlier, Ecore metamodels, ADT definitions, and Java APIs, along with their associated tooling, are examples of shapes. Similarly, a "conceptual" model m that uses the constructs of $\mathcal{L}$ (e.g., the simple `Button` machine) is projected [1] as an incarnation $\mathcal{I}_i$ conforming to the shape $\mathcal{S}_i$ in a technological stack $\mathcal{TS}_i$: an Ecore model, an ADT value, or a Java AST.

As the same model is incarnated many times, each of its incarnations $\mathcal{I}_i$ must remain synchronized. This synchronization mechanism must ensure three important properties. First, it should be efficient. This rules out any synchronization mechanism that would require doing a full traversal or serializing and deserializing the incarnations after every change. Second, it must account for any extra shape-specific information the various technological stacks have to maintain to function properly. Examples of such extra information include layout information in a textual or graphical editor, or runtime state in a simulation environment. The synchronization mechanism must thus isolate the information that relates to the model itself from the information that is specific to a particular shape. Third, the synchronization mechanism must be language-agnostic, meaning it should not have to be implemented from scratch for every shape-diverse DSL.

6.4 Synchronizing Incarnations with PRISM

In this section, we present PRISM, our prototype approach to the problem of synchronizing various incarnations of a model. Figure 6.3 gives an overview of PRISM, which is used as a communication bus between technological stacks and keeps the technological stacks fully independent. The key idea is that every change occurring on one incarnation is shipped to all other incarnations of the same model in the form of a *patch*. This patch represents the exact set of changes that occurred on one incarnation. It allows synchronizing connected incarnations efficiently without requiring serialization or a full traversal of any of the incarnations. PRISM keeps track of a matrix that associates every conceptual model to its incarnations in various technological stacks. When a change occurs on one incarnation, for instance resulting from a user edit or a refactoring, the technological stack hosting this incarnation generates a patch

1. The notion of projection here is unrelated to the notion of projectional editing [153] as there is no underlying AST to project from.

describing the change as a set of CRUD-like operations. The technological stack ships the patch to PRISM, which then propagates the patch to every other incarnation of the same model.

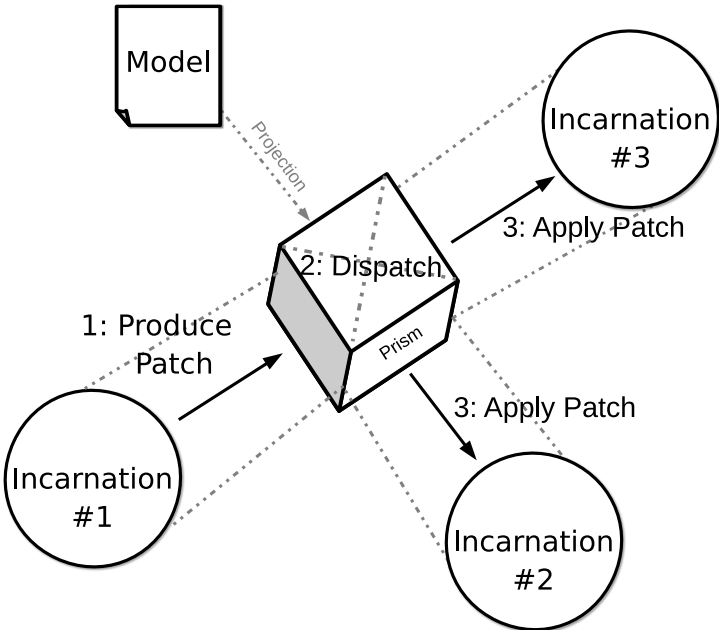


Figure 6.3 – Using PRISM to synchronize three incarnations of the same model. Here, a change occurs on Incarnation #1 and the resulting patch is shipped to Incarnation #2 and #3.

Every technological stack then interprets the patch in its own way to keep the representation synchronized. In EMF, for instance, the patch is interpreted as a set of changes that impact a model conforming to an Ecore metamodel, while in Rascal it is interpreted as a set of changes that impact an ADT value.

We detail, in the rest of this section, the patch formalism and the communication bus of PRISM.

6.4.1 Patch Formalism

Patch describes changes that occurred during the editing of an incarnation of a model. Changes are represented by a sequence of operations, which correspond to the delta between the state of the incarnation of a model before and after its editing. In our prototype implementation, the structure of this patch is prescribed by the Rascal ADT shown in Figure 6.1, largely inspired by the *edit scripts* used by Rozen and Storm [124].

Essentially, patches consist of a set of operations, which include *create* or *destroy* an object in a model, *set* or *unset* a value in an object property, *insert* or *remove* a value in an object property which is multi-valued.

```
1  @doc{A patch consists of a sequence of edits}
2  alias Patch = tuple[Id root, Edits edits];
3
4  @doc{Edits are operations attached to object identities}
5  alias Edits = lrel[Id obj, Edit edit];
6
7  data Edit
8    = put(str field, value val)
9    | unset(str field)
10   | ins(str field, int pos, value val)
11   | del(str field, int pos)
12   | create(str class)
13   | destroy();
```

Listing 6.1 – CRUD-like patch definition in Rascal.

Operations are attached to identities [83] that represent particular objects in the model. To ensure that every technological stack can apply the operations on the right elements, identities are preserved across technological stacks and, in our case, they are represented by URIs [14].

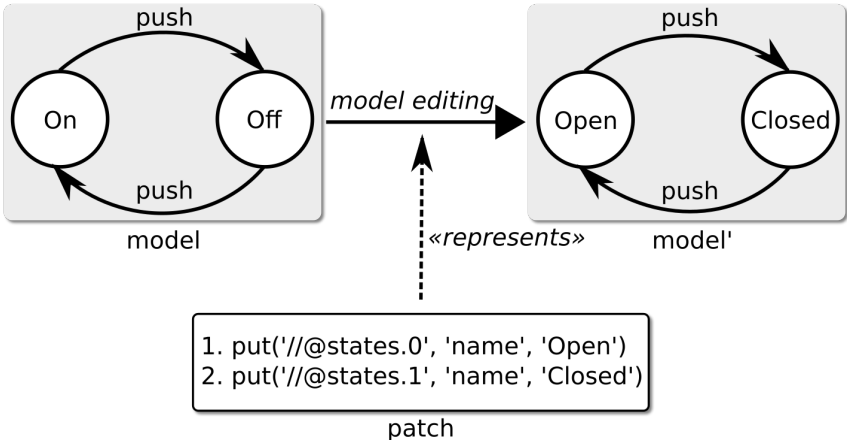


Figure 6.4 – Produced Patch representing model editings

Figure 6.4 illustrates the production of a patch. An FSM model with two states named 'On' and 'Off' is edited to rename the states 'Open' and 'Closed'. These changes take place on one incarnation of the model and have to be applied on the other incarna-

tions in the other technological stacks. To do this, we create a patch that describes these changes and that will be sent to notify the other stacks. Here the patch contains the two renames, represented by the *put* operations. Each of them contains three pieces of information: the identity of the modified object represented by the URIs '//@states.0' and '//@states.1', the name of the changed properties (i.e.,'name'), and the value assigned to the properties (i.e.,'Open' and 'Closed').

As mentioned earlier, each technological stack may want to preserve extra shape-specific information across the patches. A textual editor in Rascal, for instance, needs to keep some of the parsing information to maintain layout whenever patches are applied. So it should be possible to apply the patch while maintaining the extra information specific to a given technological stack. Intuitively, our approach supposes that all the information that does not directly relate to the constructs of a language is "extra" and therefore should not be part of the patch itself. There might be cases where sharing extra-information from one shape to the other is desirable, for instance, to share layout information between two textual editors.

6.4.2 Communication Bus

The central component of PRISM is the communication bus. It connects the different technological stacks and its role is to distribute the patches across all the stacks by following a publish/subscribe pattern to synchronize the incarnations of a model.

New technological stacks can be connected to PRISM by implementing a simple interface that consists of two operations, namely (i) *produce* which creates a patch materializing the changes on an incarnation and notifies PRISM, and (ii) *apply* which receives a patch from PRISM and interprets it to update an incarnation, taking into account the specificities of the technological stack. The way changes are detected in an incarnation and patches are produced is not prescribed by our approach. For instance, our Rascal implementation computes patches from a *diff* operation between two ADT values, while our EMF implementation captures the result of transactions on an Ecore model to produce the patches. The produce and apply operations are implemented once for every technological stack and do not have to be re-implemented for every language.

Editing model in multiple technological stacks starts with the creation of a *stream* (of patches). Other technological stacks have to connect to the *stream* after it is created. For each new connection, the creator of the *stream* sends them the current state of its

model incarnation in the form of a patch, which contains all the operations to create the complete model from an empty state. After applying the initial patch received from the connection, a technological stack is free to produce or receive patches representing changes to the model. To avoid inconsistencies, patches are only produced when the model state is valid.

Technological stacks can contain multiple models incarnations involved in different editing *stream* thus, a cornerstone artifact in PRISM is the dispatch mechanism that routes patches to the appropriate incarnations. When receiving a patch, PRISM looks up its internal matrix to determine which other incarnations of the same model should be updated. The patch is then copied and routed accordingly.

Our current implementation of the dispatch mechanism is kept simple and does not support concurrent editing on different incarnations of the same model. This would scale PRISM to advanced scenarios, such as collaborative editing, but this is beyond the scope of this work.

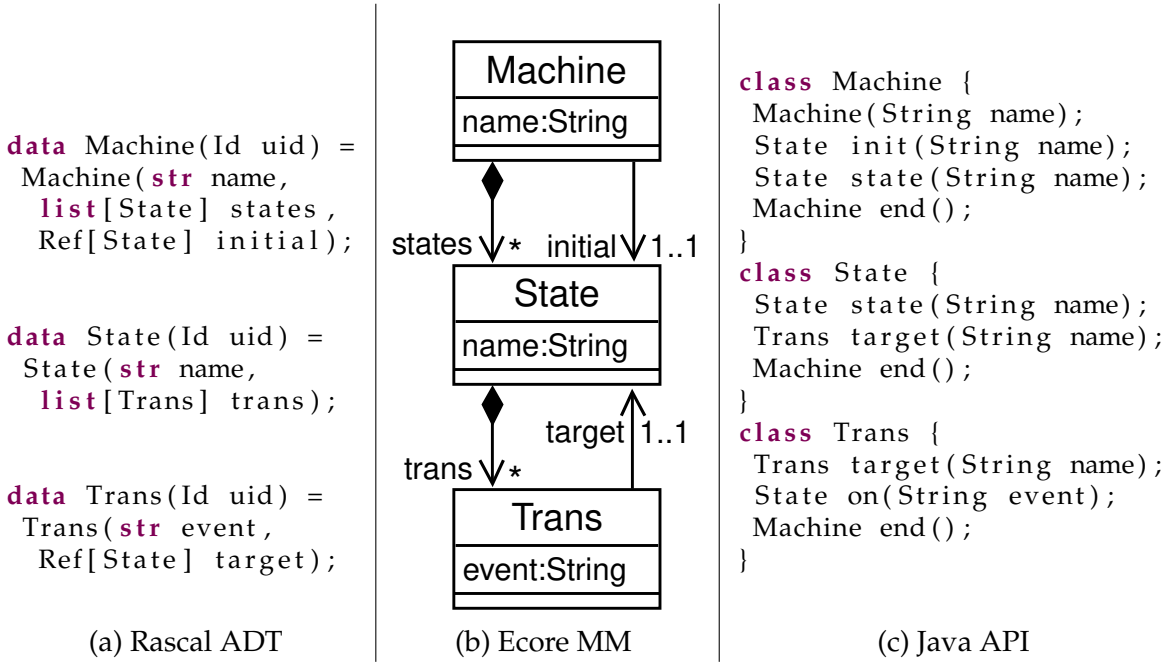
6.5 A Shape-Diverse FSM Language

We detail in this section the usage of PRISM to build a shape-diverse FSM language conjointly in Rascal, EMF, and Java technological stacks. Figure 6.5 depicts the implementation of the abstract syntax of this FSM language in the three technological stacks. The corresponding incarnations are those given in Figure 6.1.

The abstract syntax of the FSM language in Rascal is an Abstract Data Type (ADT). We declare a data type for each concept of the FSM language: a data type Machine with a name, a list of States, and a reference to an initial State; a data type State with a name and a list of transitions; a data type Trans with a triggering event and a reference to an outgoing State. All of these data types also declare an identifier.

The abstract syntax of a language in EMF is defined with a metamodel (MM). The FSM language is defined with three classes corresponding to the concepts Machine, State, and Transition. The class Machine has an attribute name, contains multiple States, and has a reference for an initial State. The class State has an attribute name and contains transitions. The class Trans has an attribute name and a reference to a target State.

Java Fluent APIs used for internal languages are defined with Java classes. The FSM language is represented with the three classes Machine, State, and Trans. The



(a) Rascal ADT (b) Ecore MM (c) Java API

Figure 6.5 – Three shapes of an FSM language; the corresponding incarnations are those depicted in Figure 6.1.

class Machine declares a constructor taking its name as a parameter, a method state taking a parameter name and returning the created State, a method init similar to the method state but used to identify the initial state. The class State has a method target taking the name of an outgoing state and has a method state with a parameter name which is used to append a declaration of a new state. The class Trans has a method taking a triggering event as a parameter and a method target taking the name of a state which is used to append a declaration of an outgoing state in the last declared state. All the classes have a method end to close their declaration, which returns the type Machine necessary to open new declarations. We build the Java API following a simple systematic convention, so as to easily pinpoint which parts of the Java AST have changed (to compute a patch) or need to be updated (to apply a patch).

In addition to the abstract syntaxes, we defined concrete syntaxes and tools for each of the technological stacks. We use Rascal to define a textual editor and a simple transformation that inserts a new state in a machine. We use EMF to define two graphical editors: a classical tree editor and a domain-specific representation with Sirius. [2] Since our fluent API is an internal language, we rely on Java tooling for the edition of an

2. https://www.eclipse.org/sirius/

FSM, which includes for example the Java content assist.

We defined three shapes for the FSM language in different technological stacks. However, the technological stacks are isolated from each other, which prevents a language user from seamlessly switching from one shape to another and using their tools without concern for the synchronization of the different incarnations. PRISM is used to bridge the technological stacks and thus break this isolation. Whenever an incarnation of the FSM model is updated, the technological stack in which the change happens produces a patch (cf. Figure 6.1) that is shipped to the other technological stacks through the dispatch mechanism of PRISM. Every technological stack interprets the patch in its own way to keep its incarnation updated, accounting for the extra information it has to manage (e.g., layouts within the textual and graphical editors). A simple matrix, internal to PRISM, keeps track of which model each incarnation is projecting to route the patch to the right incarnation.

While the Rascal and EMF shapes synchronize seamlessly, we noticed a number of challenges with the Java API. As the Java API inherits the (domain-agnostic) tooling of Java itself, it lacks the domain knowledge necessary to always generate correct patches. Due to the lack of domain-specific static semantics, a well-formed Java program may indeed produce an ill-formed FSM that cannot be interpreted by the other shapes. Besides, our prototype implementation does not account for complex string manipulation when invoking the API or use of variables. However, we believe that these are purely engineering concerns and that enough effort spent on the Java API shape would provide a flawless experience.

6.5.1 Connecting technological stacks with PRISM

PRISM is a language and technological stack agnostic means of communication. Connecting a new technological stack to PRISM consists of implementing interfaces to publish and receive patches. We have connected the technological stacks EMF, Rascal, and Java Fluent API with PRISM to synchronize their incarnations of the same model.

For the EMF technological stack, we listen to the changes on the model to register modifications. On the *save* operation of the model editor, we produce a patch representing these modifications and ship it to PRISM. Upon notification of changes by PRISM, we interpret the received patch to apply model operations corresponding to the operations from the patch. The interpretation consists of updating the current state of the

model.

For the Rascal technological stack, a listener is triggered by the program editor on *save* operation. This listener compares the last ADT tree value with the current one and translates the difference into a patch that is sent to PRISM. When PRISM notifies a change, the patch is interpreted to produce a new ADT tree value from the last tree and from the operations from the patch. Since in Rascal values are immutable, the result of the interpretation is a replacement of the current value.

For the Java Fluent API technological stack, *save* operation triggers a listener which will compare the expressions corresponding to the FSM in the current file and in the last modified version of the file to produce a patch. The patch received from PRISM notification is interpreted to edit the current file according to the operations from the patch. While in the two other technological stacks the production/interpretation of the patch is generic for any language, in this technological stack we are specific to the FSM language since there are multiple solutions to represent a concept with a fluent API (e.g., methods, parameters, ...) and there is no coding convention for fluent API.

To illustrate the application of a Patch, let's consider that an incarnation of an FSM model is edited to change the name of the initial state to 'Open'. The produced Patch will look like *put('//@states.0', 'name', 'Open')* to represent the change of property 'name' on the state identified by the URI '//@states.0'. The interpretation of Patch's operations is specific to each shape :

— In the EMF shape, the interpretation of the *put* operation starts by retrieving the modified object thanks to the identifier '//@states.0', which is supported natively by the EMF framework. We then use the EMF reflective API to look for an object's property matching the name 'name' and set the new value 'Open'.

— In the Rascal shape, applying a Patch starts by copying the ADT tree value representing the model and maintaining a map of identifier (i.e., URI) for each element of the tree. The interpretation of the *put* operation constructs an ADT value representing the property 'name' with the value 'Open'. We then search in the map for the element corresponding to the identifier '//@states.0' and we attach to it the created ADT value for the property. Once the Patch is fully applied, the ADT tree value is serialized back to text.

— In the Java fluent API shape, we start by visiting the Java AST to search for the sequence of method invocations corresponding to the incarnation of the model (i.e., the first sequence ending with an invocation 'end()'). For each invocation

'state()' or 'target() we compute an identifier (i.e., an URI). To apply the operation *put*, we look for the invocation corresponding to the identifier '//@states.0'. When then set the parameter of this invocation to 'Open'. To finish the application of the Patch, the Java AST is serialized back to text.

For the three technological stacks, we restricted the production of patches to *save* operation on editing to avoid ill-formed patches (i.e., we require a valid model before expressing changes) and since we don't support concurrent editing. We don't address collaborative editing but it is a perspective of this thesis.

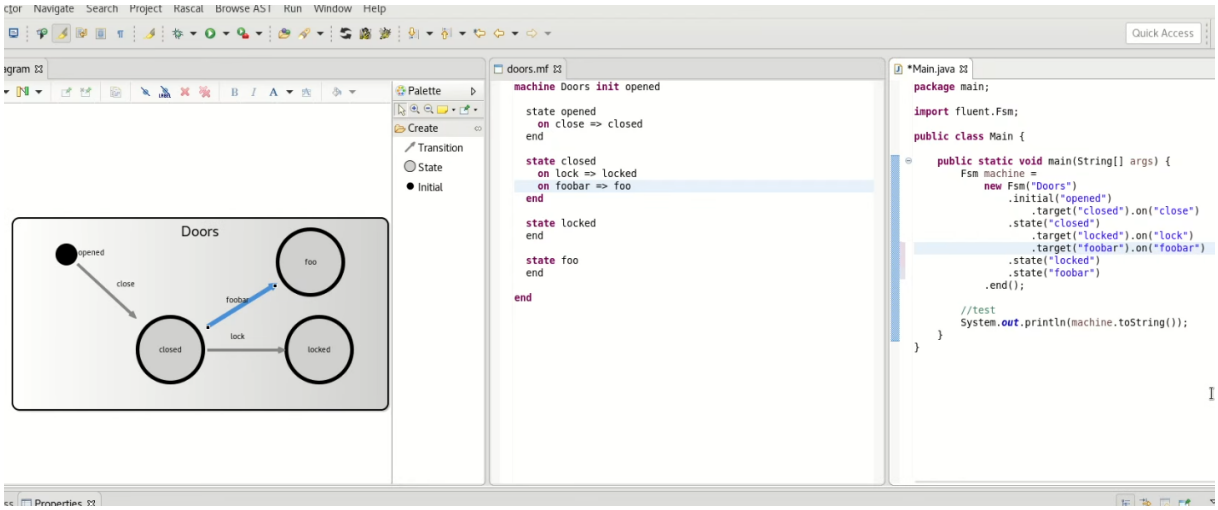


Figure 6.6 – Screenshot of PRISM connecting EMF, Rascal and Java Fluent API for the FSM language in the Eclipse IDE

Using PRISM, which connects the EMF, Rascal, and Java Fluent API technological stack, a language user can switch between them seamlessly to manipulate a model incarnated in multiple shapes. For example, editing the Java expression representing the FSM model will update the two other incarnations, afterwards, the language user can apply a program transformation in the Rascal incarnation and then finish editing the model in an EMF graphical editor. Figure 6.6 presents our prototype PRISM running in the Eclipse IDE. An FSM model is incarnated in the EMF, Rascal, and Java Fluent API technological stacks and can be edited by a language user while PRISM keeps the different incarnation synchronized.

6.6 Conclusion

When designing a DSL, language designers choose a particular technological stack from those available, which allows them to benefit from its strengths but also prohibits access to the strengths of other stacks. In this chapter, we proposed an approach to combine multiple technological stacks for the implementation and use of DSLs and thus allowing the engineering of shape-diverse DSL. To keep the different incarnation of a model synchronized across the technological stacks, we proposed a patch formalism that is agnostic of the DSL shape to express model changes and a communication bus based on a publish/subscribe pattern to deliver these patches. We implemented the approach with our prototype PRISM to support the engineering of shape-diverse DSLs in the Eclipse IDE. We applied our approach to the FSM language and showed that we were able to manipulate an FSM model incarnated in the EMF, Rascal, and Java fluent API technological stacks.

We have shown through PRISM that synchronizing models incarnated in multiple technological stacks is a solution that brings more flexibility to language designers who can combine the strengths of the stacks and to the language users who can switch between language services regardless of their stack.

CHAPTER 7

DISTRIBUTED INTEGRATED DEVELOPMENT ENVIRONMENT

In this chapter, we present the second contribution of this thesis, which complements the contribution presented in the previous chapter. This contribution is an approach to IDE distribution consisting of modularizing language services and deploying them on multiple execution platforms. We first introduce the context of the contribution (Section 7.1) and present a motivating example for language services distribution (Section 7.2). We then give an overview of the approach (Section 7.3) and detail it on the NabLab language example (Section 7.4) before presenting our evaluation of the approach and our results (Section 7.5). We finish the chapter with our conclusions on the contribution (Section 7.6). This chapter is based on our SLE'20 [29] publication.

7.1 Distributed Language Services

Modern IDEs are moving to the Software as a Service model [34] in order to benefit from advantages [67] such as reduced availability delay (since the application is already installed and configured), lower costs to maintain and/or upgrade, scalability, etc. The rise of a protocol such as the Language Server Protocol that standardizes the protocol used between a language-agnostic IDE and a language server that provides language services such as auto completion, search for definition, search for all references, compilation, etc. has allowed the emergence of high quality generic Web components to build the IDE part that runs in the browser. For instance, Monaco [1] (used

1. https://microsoft.github.io/monaco-editor/

in VSCode [2], Theia [3] , ...), Atom [4], CodeMirror [5] (CodePen [6], Jupyter [7]), are now embeddable Web components with a direct support of most of the LSP features, thereby simplifying the development of Web-based IDEs.

However, defining the architecture of an LSP server implementation and more generally the server implementation for a particular language remains a complex step. The simplistic deployment of the language server part in a sufficiently powerful cloud does not in reality provide the optimal user-experience. Each of language services has specific requirements in terms of latency and bandwidth, but also in terms of specific computing capacity. It is therefore important to tune the deployment according to the services of a particular language but also according to the context of use of the IDE for a given user, and the available execution platforms. For example, it could be required to reduce the network requirements if the quality of the network decreases for a specific user (i.e., move a language service from a server to the user's machine). Such an implementation of a language server should therefore be essentially a Dynamically Adaptive System (DAS)[119] in which we could provide tailored distribution of the language services that optimizes the user experience and their overall performance. Defining the architecture of such a system requires fine-grained modularity in both design and deployment, and the ability to run in a distributed and heterogeneous environment.

In contrast with the current approaches that provide IDEs in the form of a monolithic client-server architecture, we explore in this work the modularization of all language services to support their individual deployment and dynamic adaptation within an IDE. We propose a generative approach to automatically obtain microservices implementing language services from a language specification, complemented with a feature model that drives the safe configuration and automates the deployment of IDE features (*i.e.,* coherent groups of language services). We study the impact on performances when distributing the language services across the available execution platforms. We evaluate our approach on four EMF-based languages and demonstrate the benefit of a custom distribution of the various language services. In particular, we apply our approach to NabLab, Logo, MiniJava and ThingML to compare response times

2. https://code.visualstudio.com/
3. https://theia-ide.org/
4. https://atom.io/
5. https://codemirror.net/
6. https://codepen.io/
7. https://jupyter.org/

of language services in our approach with monolithic language server.

7.2 Motivating example

To illustrate the heterogeneity of the various services provided by modern IDEs, we use in this section and throughout this chapter the open-source and industrial DSL NabLab [8]. NabLab provides a productive development environment for numerical analysis over exascale HPC technologies. The associated IDE provides all the common editing services (syntax coloring, auto-completion, validators...) and a complex compilation chain targeting various backends. NabLab users are mathematicians and physicists that write algorithms for numerical analysis. NabLab programs are mainly composed of jobs with a complex data flow between them, representing physical systems. As the computation used to simulate physical systems is expensive, NabLab programs are given to a compilation chain generating efficient source code to run the different jobs in parallel. We chose NabLab because developing NabLab software provides a wide range of requirements, from responsive code edition operations to CPU intensive compilation and execution.

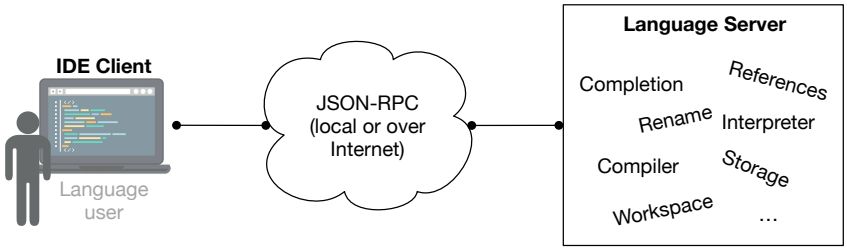


Figure 7.1 – Current IDE Architecture, including a language-agnostic client that provides the user interface, and a language-specific server that provides all the language services for a given language

Figure 7.1 represents the current IDE architecture according to the state of practices. This architecture separates the IDE client, which is the language-agnostic interface for language users, from the language server which implements the language-specific services. The IDE client remotely calls these services by sending JSON-RPC [9] messages to the language server. This architecture makes it possible to deploy the IDE client and

8. https://github.com/cea-hpc/NabLab
9. https://www.jsonrpc.org/

the language server possibly in different execution platforms (*e.g.*, the client on the development laptop, and the server on the cloud or an application server). In practice, NabLab has been developed using the *Eclipse Modeling Framework* [132], including the Ecore [10], Xtext [11] and Sirius [12] technologies.

For the sake of illustration, we selected four representative language services provided by the NabLab IDE: *completion* is a content assist that returns a list of proposals for a given context, *references* searches for elements in a file referring to a given symbol, *rename* changes names for a given element and for all its referring elements, and *compiler* performs graph analysis of the concurrent job to generate optimized Java source code (one of the possible backends in NabLab). We have chosen these language services to be representative of language user's activities: code edition, code navigation, code refactoring and code transformation.

In the NabLab IDE, *completion*, *references* and *rename* are obtained with Xtext and the support of the *Language Server Protocol*, while *compiler* is a separate compilation chain integrated and prompted from the IDE. To illustrate the heterogeneity of these language services and the potential benefits of distributing them, we measured their

10. https://www.eclipse.org/ecoretools
11. https://www.eclipse.org/Xtext
12. https://www.eclipse.org/sirius

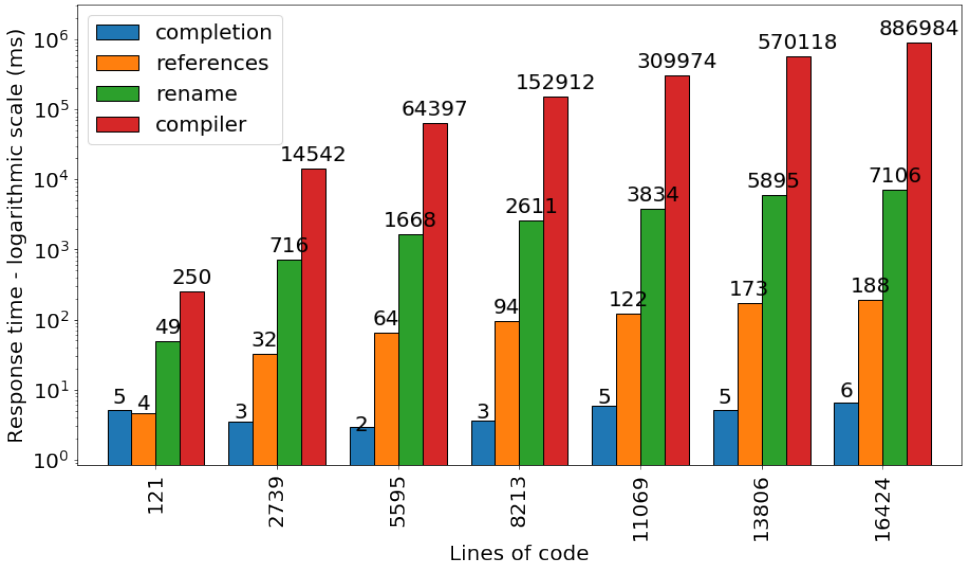


Figure 7.2 – Response times of four NabLab services (client and server deployed on the same local development laptop)

response times on NabLab files of increasing size. The measurements were performed 100 times for each language service, with a client and a server both deployed on the same machine with an Intel Core i7-7600U CPU at 2.80GHz, 32 GiB of RAM, and the HotSpot JVM 11.0.5. Figure 7.2 presents the means of the response times expressed in milliseconds (ms) for the different files and, due to the large range of values, with a logarithmic scale.

We observe an important heterogeneity among the language services, ranging from *completion* that lasts about 5 ms constantly over the files, to *references* that goes up to about 188 ms, *rename* that goes up to 7.106 seconds, and finally *compiler* that goes up to 14.78 minutes. This difference in response times between the language services is of several orders of magnitude. It can be explained by the workload of each service: *completion* traverses object's references, *reference* is a query in a graph of objects, *rename* rewrites the file and *compiler* performs complex graph analysis and generates source code. This motivates the need for an individual and distributed deployment of each language service to leverage better the available execution platforms, fit the activities performed, and eventually provide the best user experience within the IDE client. In particular, we focus on the following research questions:

RQ1 Is it possible to provide a systematic approach that automates the modularization of the language service implementations, supports their individual deployment, and enables their dynamic adaptation according to a given context (*e.g.*, usage, environment)?

RQ2 Is it possible to optimize the distribution of the language services across the available runtime platforms (*e.g.*, local platform, application server, cloud) to improve their performances within the IDE?

7.3 Approach overview

We propose a systematic approach that eases the modularization and distribution of highly configurable IDEs for DSLs. The approach takes as inputs i) a software language specification, in the form of a metamodel, a syntax description, and any additional concerns such as validators, compilers, etc., and ii) a set of desired features (*i.e.*, coherent groups of services) that the IDE must or may provide. As output, the approach generates a set of modular, language-specific, IDE features and a tool-supported fea-

ture model to configure and automate their distribution and integration within a Web-based IDE.

The following quality criteria guided our current implementation of this systematic approach:

— IDE configurability for the end user;
— efficiency of resource usage: CPU time, bandwidth, reactivity as perceived by the end user;
— extendibility and reusability from the point of the language designer, *i.e.,* when the set of features evolve or when distributed platform technologies change (long term, human driven adaptation);
— adaptability, possibly dynamically (according to the usage and environment).

Our process design decisions were taken to obtain a satisfactory balance of these criteria.

We distinguish two different user roles: language users, and language designers. In our process, configurability of an IDE by an end-user relies on software product line principles: using a language specification as input, language designers build in fact a family of distributed IDEs. Language designers, or even language users, can then configure the family to obtain an IDE that suits the user's needs and experience.

Figure 7.3 presents the overall approach, from the specifications to the deployment of an IDE. First, a language designer provides a *language specification* along with a *protocol specification* that describes the expected modularity of the language services and their interactions. From these specifications, we automatically generate a set of microservices implementing the IDE features and a feature model that captures their valid configurations. The feature model offers a description of the variability of a system, here an IDE, presented as a tree of features enriched with logical constraints. It essentially describes what the feature alternatives are, their dependencies and whether they are mandatory or optional. The IDE and its deployment can then be configured by a language designer or user, depending on who has the knowledge to decide where to deploy the microservices. Configurations could also be proposed by an automated process involving a predictive model in the deployment, or through dynamic reconfiguration, in an attempt to maintain metrics such as user experience. The IDE configuration consists in selecting the microservices that will be available to the language user, together with information on where to deploy them. The feature model is used to validate the set of microservices to be deployed by checking the variability constraints.

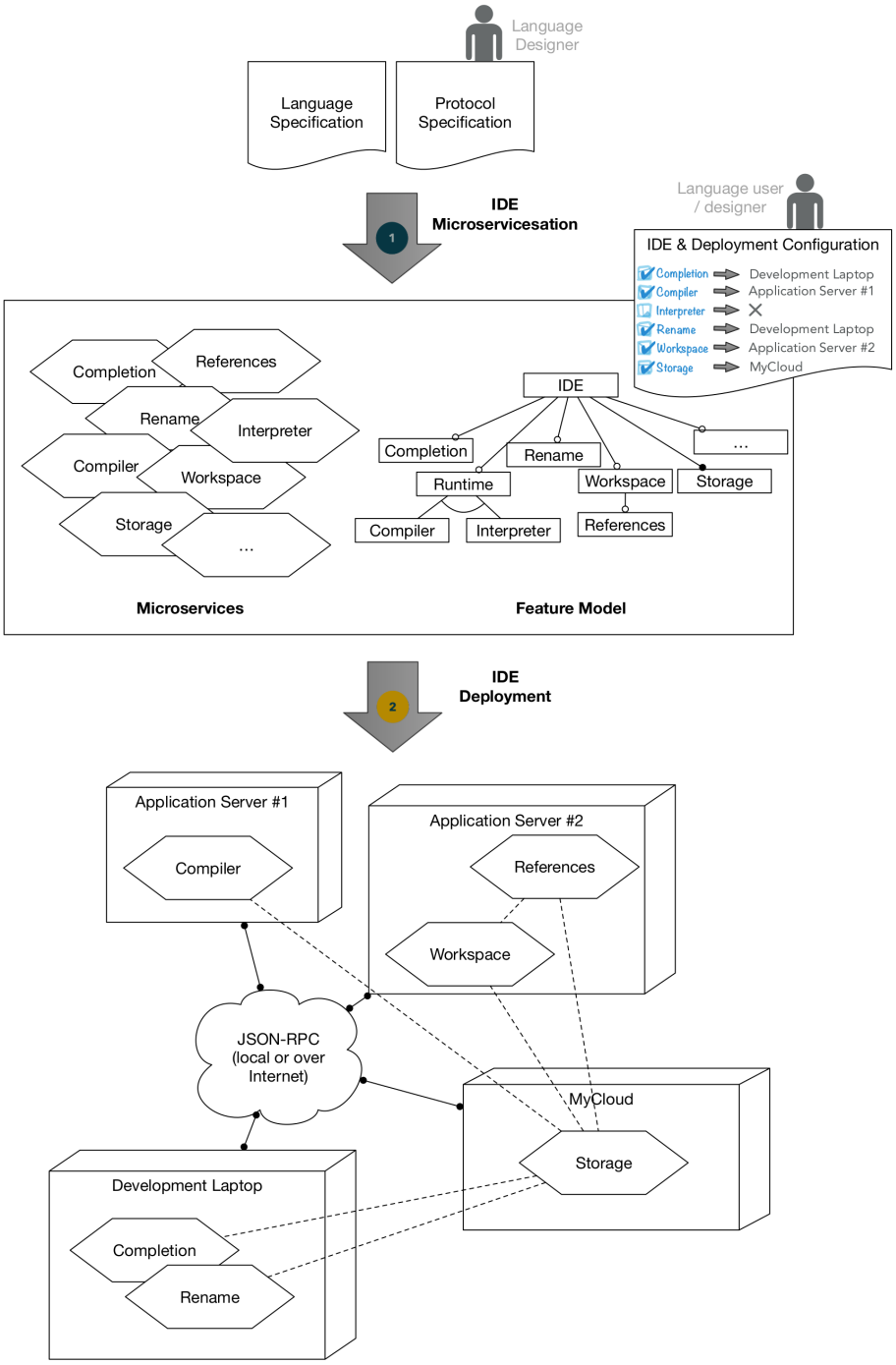


Figure 7.3 – Approach overview, with the two main steps: ① IDE microservicesation and ② IDE deployment

At the end, the microservices are distributed to different execution platforms such as

development laptop, cloud or application servers. At this stage, the language services are running and the language user can use them through an IDE client.

In the rest of this section, we detail the two main steps of our approach (① and ② in Figure 7.3).

7.3.1 Designing IDE microservices

The design of an IDE family is based on two main inputs that are required to improve flexibility and reusability of elementary design elements, thereby supporting the language designer in providing a highly and dynamically customizable distributed IDE.

Language specification The language designer needs a language specification. In our prototype, the language specification comes as an Ecore metamodel, an Xtext grammar description, and additional services such as compilers. Using tools such as XText, the language designer is able to produce a software module that acts as a parser and builds a model from a program file, as an internal, metamodel compliant, form of the program.

Protocol specification The language designer also provides a description of the expected modularity of the language services in the form of a so called *protocol*. Code completion, symbol renaming, compiler, are examples of such language services (*capabilities*) that may be grouped into IDE features as deployment units. The grouping of the language services into IDE features, and their inter-dependencies, are expressed in a specific model, for which we provide a specific DSL with a concrete syntax, and a metamodel shown in Figure 7.4.

Using this DSL the language designer is able to declare the properties and relationships of each feature declared in a protocol specification. The DSL relies on the following types (cf. Figure 7.4):

— The Capability type describes the definition of a basic service point, akin to a callable function in the architecture. Details of the function implementation, such as a port number for micro-service based implementations are also indirectly provided by a Capability.

— The Feature type regroups a coherent set of capabilities. It typically represents

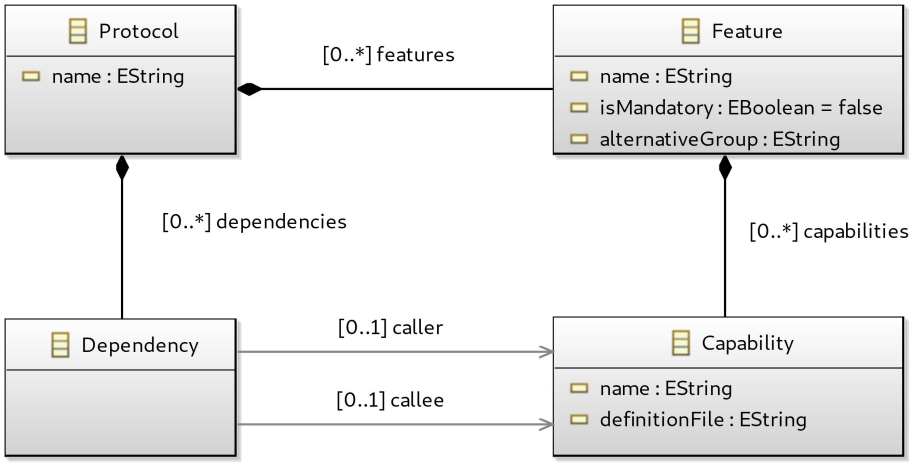


Figure 7.4 – The metamodel used to describe a protocol for IDE features

an elementary tool supported by the IDE (*e.g.*, code completion, name refactoring).

— The Protocol type regroups the features that are potentially supported by an IDE.

— The Dependency type describes the relationships between capabilities supported by a protocol, for instance the call dependency between two capabilities.

From the language specification and the protocol specification, we implemented a generative approach (① in Figure 7.3) that produces:

— a set of modular language services in the form of cloud-native applications as microservices, and the required code that will take care of the communication between language services as described in the protocol.

— a feature model that represents the IDE family.

The feature model can then be enriched with deployments constraints, *e.g.*:

— for efficiency reason some capabilities need to be implemented by the same feature, and therefore deployed on the same execution node;

— some features are alternatives in a group, *e.g.*, if at run time the currently deployed feature becomes unavailable then one alternative will be deployed automatically.

7.3.2 IDE Deployment

The final task to build a usable, running IDE is the deployment phase. As mentioned before, we aim at providing a family of IDEs to the final language user, as different users have different needs, and a given user may also wish to tailor the IDE depending on the current tasks he is involved in. To support this flexibility we defined a configuration language to allow the user to control on which capabilities to deploy and where, while maintaining the constraints defined in the feature model.

A point to consider in order to support reconfiguration of the deployment (*i.e.,* to move microservices) is that we implement language services as stateless microservices. A stateless microservice does not keep any states between requests. This has benefits for the scalability of distributed applications since a microservice can be replicated and the requests dispatched among the different instances to handle load increases. It also allows the microservices to be moved easily from one location to another, and it avoids data loss in the case of a microservice crash. However, stateless microservice involves retrieving and parsing programs before processing language services, which increases response times compared to stateful microservices. This additional cost on response times must be taken into account to benefit from the reconfiguration of a distributed IDE.

The proposed generative approach (② in Figure 7.3) takes as inputs a specific configuration of the feature model and the microservices, and produces a distributed IDE integrated with the Web-based client.

7.4 Towards a modular and distributed IDE

In this section we use a running example to detail the steps introduced in the previous section. Our task is the design of a distributed IDE for Nablab users.

Developing NabLab software provides a wide range of requirements, from responsive code edition operations to CPU intensive compilation and execution. Taking care of this range of requirements is best addressed by distributing language services on various types of execution platforms, which makes NabLab a good candidate language for our experiments on modular and distributed IDE construction.

NabLab users are supported by a set of tools that form a specific IDE:

— A textual editor supports contextual code completion, code folding, syntax high-

lighting, error detection, quick fixes, variable scoping, and type checking.
— A model explorer provides a dedicated outline view and a contextual LaTeX view.
— A debugging environment provides variable inspection, plot display and 2D/3D visualization.
— A NabLab compiler generates efficient implementations thanks to the associated compilation chain.

7.4.1 Language and protocol specifications

As mentioned in the previous section, in our approach a language specification consists of defining a metamodel and associated concerns such as the concrete syntax and semantics. We use the Eclipse Modeling Framework [13] (EMF) and its ecosystem to define our language specification.

The concrete syntax of NabLab is a grammar defined with Xtext[43]. Taking a grammar as input, Xtext is able to generate the source code implementing a set of language services for a text editor. Xtext can also generate a language server, which embeds the language services that are then callable remotely. In our approach we use this generator to produce implementations of IDE features.

The compilation chain comes as a separate language service implemented with Xtend [14], and this service is integrated and prompted from the IDE client.

Listing 7.1 is an excerpt from the protocol specification for NabLab. This protocol specification conforms to the metamodel described in Figure 7.4. Our tool to edit a protocol textual specification and simultaneously build a protocol model is based on Xtext. Our NabLab protocol model declares the *storage* and *completion* features, and a dependency between their respective capabilities. The *storage* feature provides the following capabilities: *document*, to retrieve a persisted document (a program), and *update*, to change the contents of a program. The *completion* feature provides the *complete* capability to get content assist proposals. In the dependencies, the *complete* capability first retrieves a program by calling the *document* capability before computing the set of completion choices. The protocol specification of NabLab, in addition to the *storage* and *completion* features shown in the listing, also has the following features: *workspace*, which computes diagnostics for programs and indexes their content, *definition*, which

13. https://www.eclipse.org/modeling/emf/
14. https://www.eclipse.org/xtend/

```
1  Protocol {
2    mandatory feature storage {
3      capabilities :
4        document
5        update
6    }
7    feature completion {
8      capabilities:
9        complete
10   }
11   (...)
12   dependencies {
13     completion.complete → storage.document
14     (...)
15   }
16 }
```

Listing 7.1 – Excerpt of the protocol specification for the completion feature of NabLab

gives the location of an element's definition, *highlight*, which looks up the element's definition as well as other elements linked to the same definition, *hover*, which returns the element's description, *documentSymbol*, which returns all the elements of a program, *formatting*, which computes text edition operations to normalize program's indentation, *rename*, which returns text edition operations to rename an element's definition and other elements linked to the same definition, *references*, which finds elements having the same definition, *symbol*, which returns all the symbols in the opened program matching a query, and *compiler*, which generates a Java source file from a NabLab file.

7.4.2 Feature model generation

In a second step we generate a NabLab feature model by taking the NabLab protocol specification as input. Figure 7.5 presents the resulting feature model built by applying the approach described in the previous section. The features of the model match the IDE features of the protocol specification and the hierarchy is derived from the dependencies between capabilities. For example, in the protocol specification *symbol* has a capability calling the *index* capability declared in *workspace*. We infer that *symbol* requires *workspace* to run, and we set a parent-child relationship in the feature model. It means that a configuration of this feature model containing *symbol* is valid only if

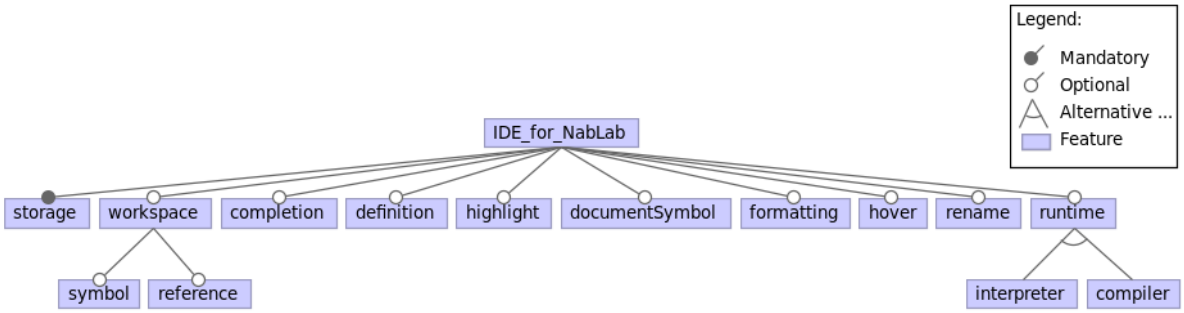


Figure 7.5 – Feature model of the language NabLab

workspace is also present. We compute similarly the hierarchy for other features.

Our implementation of the feature model generator relies on the Feature IDE framework [139], which provides facilities to construct and manipulate feature models.

The feature model generated with this framework is then used in the deployment configurator to check the validity of a given configuration by analyzing the constraints defined in the feature model.

7.4.3 Microservice generation

We use the Quarkus [15] framework to implement Java microservices for each IDE feature defined in the protocol specification. A Java class is generated with methods corresponding to the *Capabilities* of the IDE feature. Each *Capability* is implemented as a REST API, by annotating methods with HTTP verbs, query parameters, and endpoint paths. These methods are callbacks for the HTTP requests. We process the Capability's JSON definition file to generate arguments and return type for the methods. We also generate proxy classes from the dependencies of the IDE features. These classes provide methods to remotely call the capabilities implemented in the other microservices.

The language service implementations of NabLab are provided by Xtext, except for the compiler. To leverage these available implementations, we modularized an Xtext implementation to have a set of modular language services usable through a generic protocol. By looking at the name of the language service to detect if it matches Xtext's features, the code generator inserts calls to the methods of the Xtext module that implements the language service. For language services that do no match Xtext's features, such as the compiler, we generate the required code to embed them into microservices

15. https://quarkus.io/

and allow their integration into the IDE.

7.4.4 Deployment configuration

Our distribution of the IDE features relies on Docker [16] and a container orchestrator (*e.g.*, Kubernetes [17] or Nomad [18]). The microservices are packaged in Docker containers, which isolate the microservices and ease the deployment by embedding their runtime environment (*e.g.*, a JVM). We deploy containers in a cluster by using a container orchestrator, since it permits to plan their deployment at different locations and move them at runtime. Our deployment configurator monitors the deployment using the Kubernetes API, to get the list of deployed microservices with their locations to construct the configuration representing the current deployment. We provide a frontend for the configurator as a web page that displays configurations to the language user and designer. Through this web page, the language user can change the deployment configuration by disabling microservices, selecting new ones and change their deployment location. If the new configuration is valid, a deployment plan is sent to the Kubernetes API.

7.4.5 Distributed IDE architecture

The architecture of the final application is divided into a frontend part and a backend part, as depicted in Figure 7.6. The frontend part is a web page executed in a web browser. The backend part is a set of microservices running a Kubernetes cluster. The frontend contains the Monaco program editor that communicates with the backend through the Language Server Protocol and Javascript code calling the *compiler* service through direct HTTP requests.

As language services are distributed in multiple microservice, a microservice acting as a router is connected to Monaco through a websocket. The router forwards the LSP messages to the microservice implementing the relevant language service and sends their response back to Monaco.

Language services are implemented as stateless microservices. With each request they receive, they retrieve the current program from a microservice dedicated to stor-

16. https://www.docker.com/
17. https://kubernetes.io/
18. https://www.nomadproject.io/

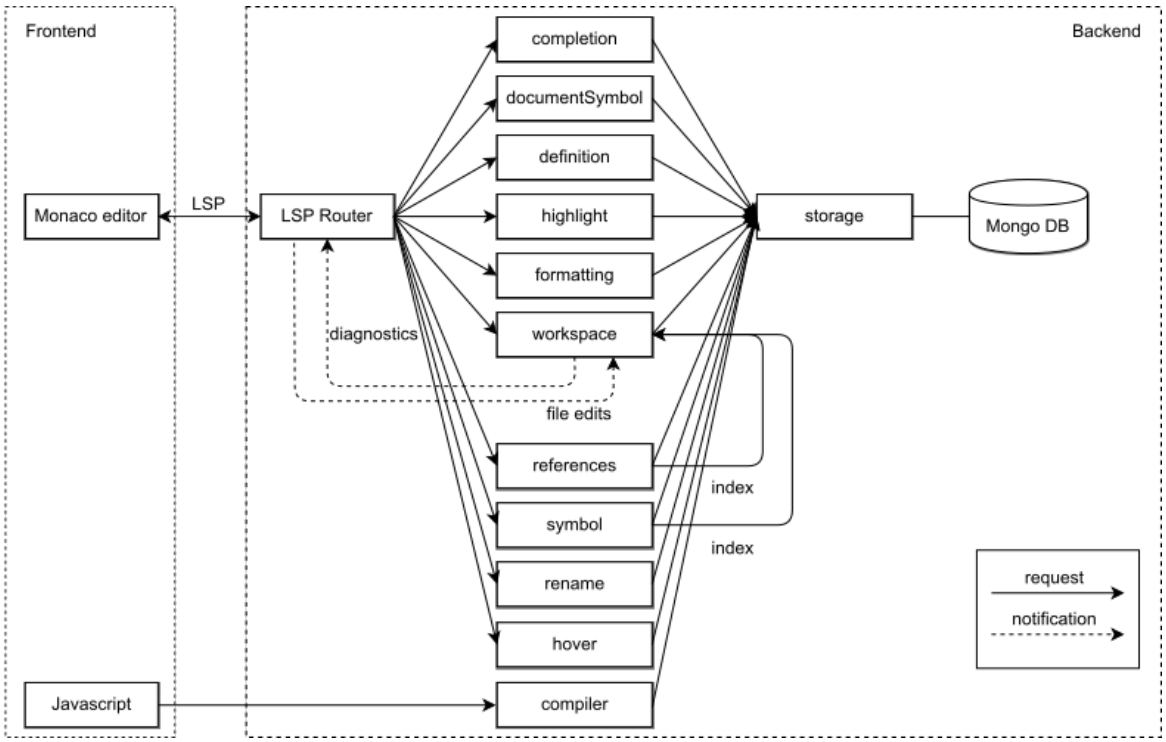


Figure 7.6 – Microservice architecture

ing it. We use Mongo as the database to store the programs. The microservice implementing the *workspace* language service receives program edits notifications and sends diagnostics notifications to the frontend (i.e., errors markers). It also computes an index of the symbols in the programs, which is required by the microservices implementing *references* and *symbol*.

7.5 Experimentations

In this section we present our experiments to answer the research questions introduced in Section 7.2 (RQ1 and RQ2). We conducted these experiments on four EMF-based languages: NabLab, our own implementation of the Logo language, ThingML [19] and MiniJava [20]. Logo is an educational programming language dedicated to 2D drawings, ThingML is dedicated to applications for the Internet of Things, and MiniJava is a subset of the Java language. We selected this mix of general purpose and domain

19. https://github.com/TelluIoT/ThingML
20. https://github.com/tetrabox/minijava

specific languages of distinct domains to be representative of the generalization of the proposed approach.

Our experiments compare language services implemented by monolithic servers deployed locally and modular servers distributed over the available execution platforms. Our results show that monolithic servers deployed locally offer better response times when not resource demanding but quickly limited for computationally intensive IDE features, and that parsing models and loading model elements constitute a bottleneck for microservices, which would prevent the design of an entirely stateless architecture.

7.5.1 Experimental setup

The evaluation was performed on three machines in a Nomad [21] cluster. All machines were Dell PowerEdge R330 with Intel(R) Xeon(R) CPU E3-1280 v6 @ 3.90GHz 4 cores, hyper-threading and 31GB of RAM. The IDE features were deployed in Docker containers and running on OpenJDK 11.0.5.

The evaluation measured the response times of language services implemented as monolithic language servers, and as distributed language servers with microservices. Both are implementing the Language Server Protocol [22] (LSP), and possibly additional services such as a compiler. Monolithic language servers were deployed on a single machine of the cluster. We generated programs of a similar number of lines of code based on the content of existing examples for each language [23]. For each program, we initialized an LSP session and opened the program. We then called 100 times each language service sequentially and measured the time elapsed between the request and the reply.

We performed the same experiments with the distributed servers after a distribution of the language services over the available cluster. On the first machine we deployed the language service *storage*, on the second machine *workspace*, *completion*, *definition*, *highlight*, *documentSymbol* and on the third machine *hover*, *references*, *rename*, *symbol*. Since we wanted to measure the cost of message exchanges, we deployed depending IDE features on different machines: for instance, *workspace* feature was not on the same machine than *references* and *symbol*, that *storage* is on its own machine. The

21. https://nomadproject.io/
22. https://microsoft.github.io/language-server-protocol/
23. https://anonymous.4open.science/r/e03961ac-9f27-4c52-ab28-87cf105a83f4/

client that sends the requests to language services was deployed outside of the cluster, on a machine connected to the cluster by a local network.

In the case of the distributed servers, we also measured the times taken to load and resolve the references of models, in order to highlight impact of load and resolve phases in the response time of the microservices. More precisely, models are graphs of objects that reference each other but these references are not resolved at the first load, they are resolved on demand when language services browse the models, which has an impact on the processing time of the language services. For each language we repeated 100 times the measure of loading and resolution time of each program .

7.5.2 Results

To investigate the gains and the costs of distributing language services, we first compared the response times of language services of the motivating example in Section 7.2, which are implemented by a monolithic server running on a laptop, to the response times of the same language services implemented by microservices running on the Nomad cluster for the NabLab language. In a second experiment, we compared the response times of language services for monolithic and distributed architectures running both on the Nomad cluster to evaluate the cost of the distribution for the languages NabLab, Logo, ThingML and MiniJava. We then measured the loading time of the programs for these four languages and compare them to the response times of language services to evaluate the impact of the stateless architecture.

Microservice-based version

We measured the response times of microservices running on a cluster implementing the language services of the motivating example presented in Figure 7.2 and compare them with the response times of language services running locally on the laptop. Figure 7.7 presents response times of these microservices deployed in the Nomad cluster. The percentage given for each bar represents the overhead with regard to the time presented in Figure 7.2. Response times of the feature *completion* increase with file size and are several orders higher than in Figure 7.2, where they are between 2 and 6 milliseconds. Response times for *references* are 2 to 10 times higher than those in Figure 7.2. Feature *rename* takes 6% more on the 121-line file, but gains between 16% and 25% for the other files. Feature *compiler* gains 51% on the 121-line file and gains between 9%

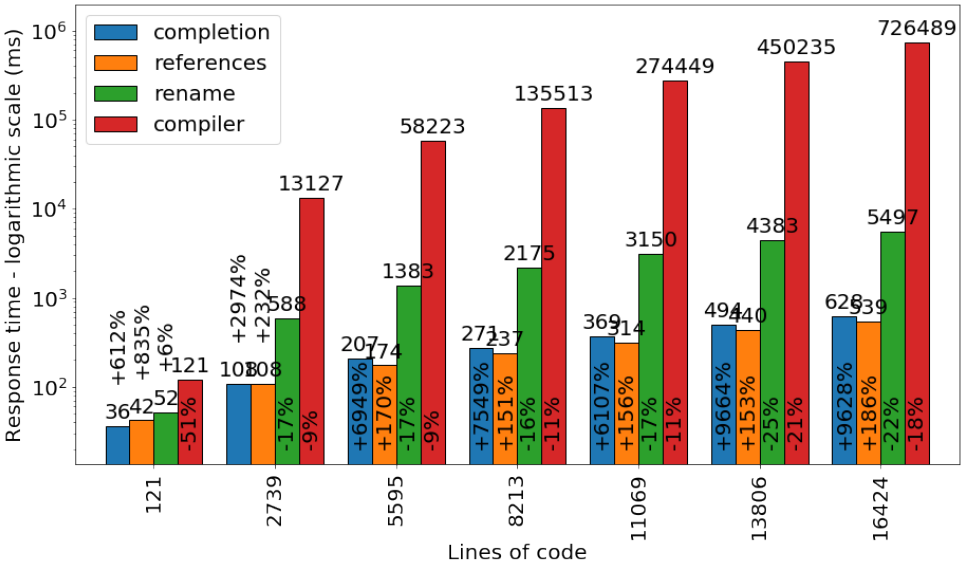


Figure 7.7 – Response times of language services of the motivating example deployed as microservices in the Nomad cluster, and comparison with the response times from Figure 7.2.

and 21% on the other files.

Protocol

To evaluate the cost of the distribution of the language services, we measured response times on programs of increasing sizes for the languages Logo, NabLab, ThingML and MiniJava, with both monolithic and distributed servers. For the distributed architecture, we also measured the overheads from message exchanges between the microservices to fulfill the request to language services. We performed 100 measurements for each language service on each program and for both servers, and computed the means. For the 320 measured means, 232 of them have a coefficient of variation (*i.e.*, the ratio of the standard deviation to the mean) below 30%.

Figures 7.8, 7.9, 7.10 and 7.11 presents a comparison of the response times of the language services for the monolithic and distributed architectures on programs of increasing size, respectively for NabLab, Logo, MiniJava and ThingML. The response times of the distributed architecture contains the *protocol overhead* parts representing the times taken by the exchanges between the microservices, which include the retrieval of programs from the *storage* microservice, and the retrieval of the workspace's index (only for *references* and *symbol* features). The upper parts of the bar displayed in

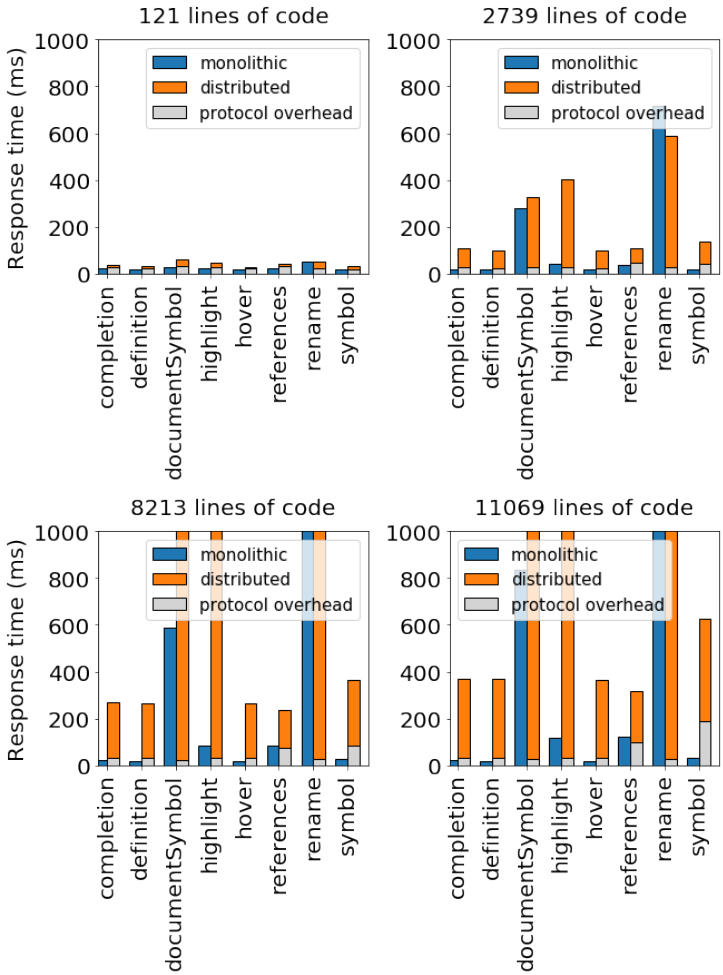


Figure 7.8 – Comparison of response times in monolithic and distributed architectures for language services of NabLab

orange represent the times not due to the exchanges between the microservices, which include the processing times of the internal logic of the microservice implementing the language service, and the time for sending the response to the client.

In most cases, the distributed architecture introduces an overhead in comparison to the monolithic implementation. However, we also observe that the overhead is marginally due to the protocol (implied by the modularization), but rather due to the microservice execution time. We further explore this in the rest of this section.

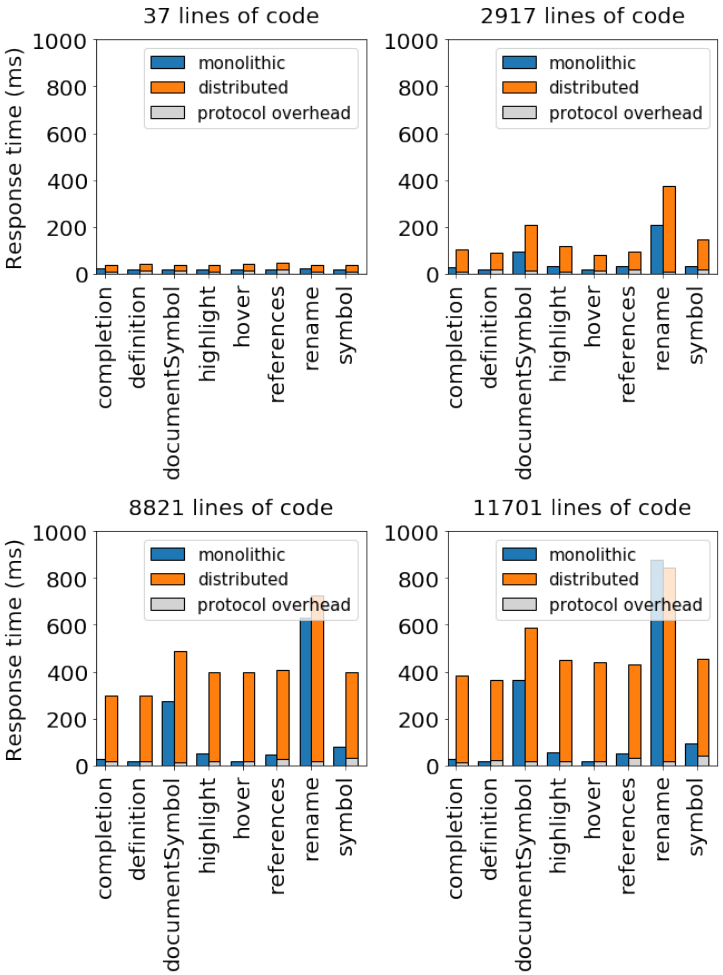


Figure 7.9 – Comparison of response times in monolithic and distributed architectures for language services of Logo

Statelessness impact

The microservices being stateless, they all require to fetch and load the necessary part of the model in addition to executing the corresponding language service(s). To evaluate the impact on response times of the stateless nature of microservices implementing language services, we measured the initial load times and the full references resolution times for the four languages NabLab, Logo, Minijava and ThingML on the same programs used before. The considered programs are EMF models, in the form of graphs of objects which are loaded lazily. An initial load is performed to build the objects of the model but references between them are resolved on demand. Language services browse models when performing their internal logic. This process requires

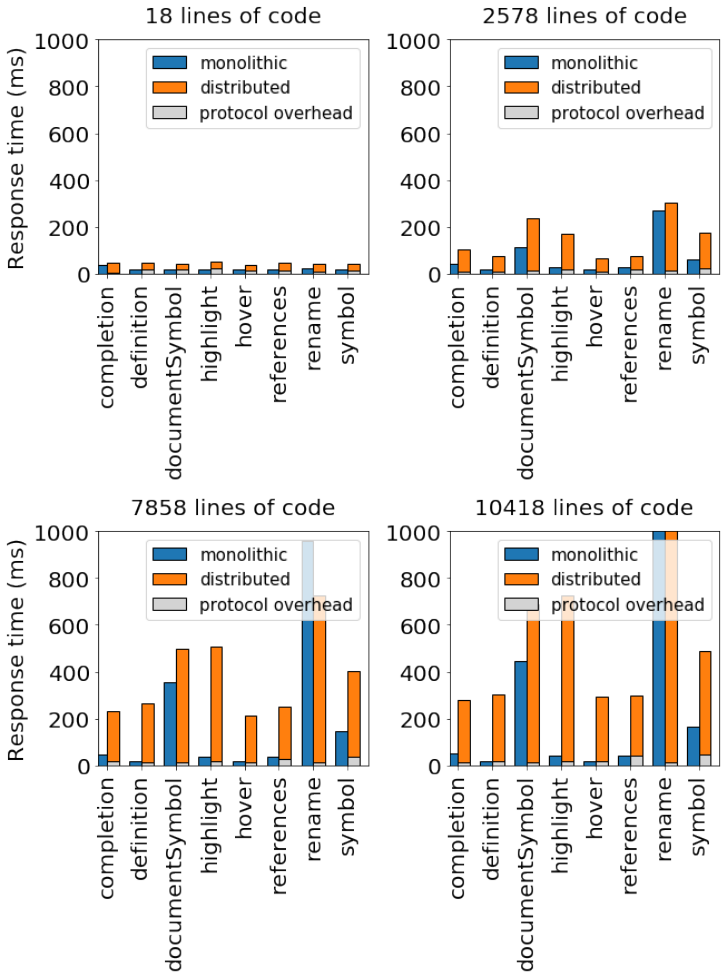


Figure 7.10 – Comparison of response times in monolithic and distributed architectures for language services of MiniJava

to resolve visited references, which consist of finding the referenced elements in the model, and therefore has a cost in time. This means that language services performing simple queries on a program, such as the *completion* feature which browses few object references, are less impacted by model loading than language services that browse the whole model, such as for instance the *documentSymbol* feature which collects all named elements of a model.

Figure 7.12 shows the means of the measurements for the four languages, on programs of increasing size. Since models are loaded lazily, we measured the initial load times and the reference resolution times. The *load* curve represents the times to parse programs and build the corresponding model. The *resolve* curve represents the times

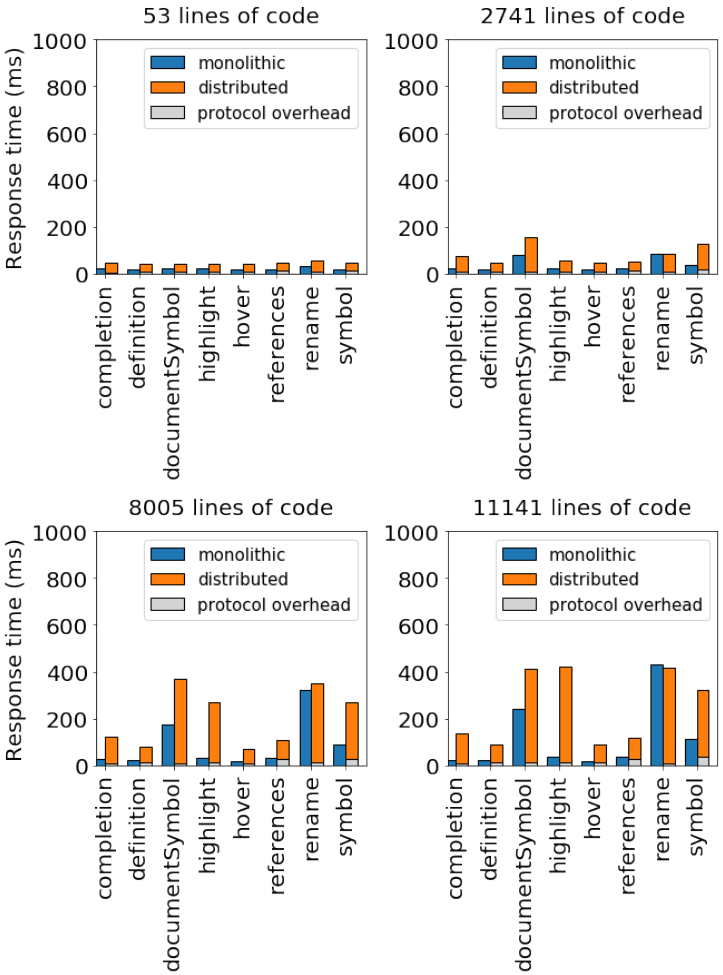


Figure 7.11 – Comparison of response times in monolithic and distributed architectures for language services of ThingML

to resolve all the references between objects in the model. The *load* and *resolve* are cumulative times that correspond to the complete model loading.

In all cases we observe a linear time for *load*, while *resolve* can be exponential according to the size of the considered program and consume an important part of the overall model loading time.

To further compare the load and resolution times of model with language service response times, we measured *completion*, which is one of the fastest of our language services and *documentSymbol* which is one of the most time consuming. Figure 7.13 shows their response times on programs of increasing size for the monolithic and distributed architectures of the four languages. The differences of response times of *completion* and

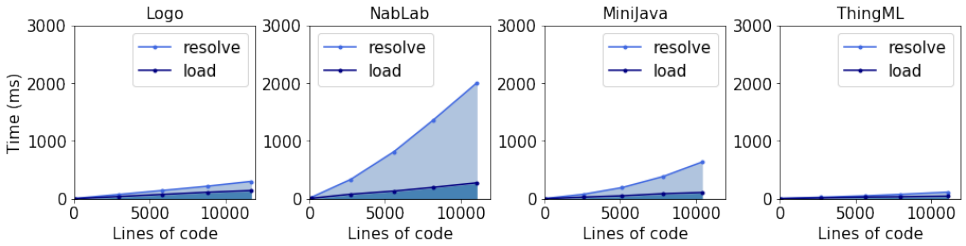


Figure 7.12 – EMF model load and resolution times

documentSymbol between monolithic and distributed architectures go up to 355 ms and 220 ms respectively for Logo, 347 ms and 2058 ms for NabLab, 229 ms and 242 ms for MiniJava, and 110 ms and 174 ms for ThingML.

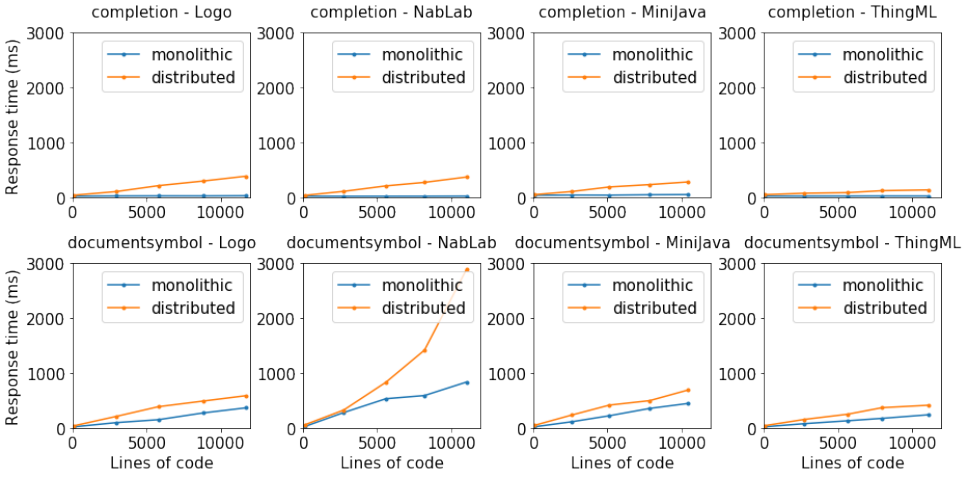


Figure 7.13 – Comparison of services browsing few model references (completion) and many model references (documentSymbol) for the languages Logo, NabLab, MiniJava and ThingML in the monolithic and the distributed architectures

7.5.3 Discussion

We modularized and distributed language services in a cluster and we compared their response times with monolithic servers implementing the same features. In this section we discuss these results to answer the research questions introduced in section 7.2.

Systematic approach to automate the modularization and individual deployment of language services (RQ1)

We present in this work a first generative approach to modularize all services of a language server in the form of deployment units as microservices, and a second generative approach in charge of establishing the communication between the various microservices corresponding to a given configuration of the expected IDE. The granularity of the IDE features in terms of the language services to be included in a given microservice, as well as the information on the possible dependencies between them, are given by a protocol specification taken as input of our overall approach. The microservices are stateless, and support custom and possibly dynamic adaptation of their configuration.

We experiment our approach on four representative EMF-based languages, namely NabLab, Logo, MiniJava and ThingML, and demonstrate the ability of our approach to be applied on all of them. We generate deployment configurators for these four languages, and select language services to be deployed, distribute them on the machines of a cluster and dynamically change the deployment configuration. For all languages, the monolithic and distributed implementations of the language server are functionally equivalent.

The results from Figures 7.8 to 7.11 show that the protocol and the communication between the microservices are only a small part of the overall overhead in the response times of the distributed features. The overhead for features *rename* and *symbol* last longer, which can be explained by the fact that these two features query the symbol index in addition to retrieving the model while the other features only send requests to retrieve the model.

The differences of response times in Figure 7.13 between the monolithic architecture and the distributed architectures can be explained by the fact that the monolithic servers are stateful and the microservices are stateless. The monolithic servers load models only once at initialization and perform model validation to find errors, a process that requires traversing all elements of the model. This means that models are loaded and all their references are resolved when the internal logic of the feature is running. In the case of microservices, models are loaded at each request and references are resolved on demand while the internal logic of the feature is running. We notice in Figure 7.13, that the differences in response time between the monolithic and distributed curves are at least equal to the loading times of the models in Figure 7.12 and

even close to the full resolution time of the models for NabLab. We also notice that the *completion* feature has less overhead than the *documentSymbol* feature. This difference is due to the lazy loading of the model since *completion* has to resolve few model object references whereas *documentSymbol* has to resolve all containment references of the model.

From these observations we conclude that the protocol for stateless microservices implementing IDE features introduces a small overhead corresponding to message exchanges. The differences of response times between the monolithic and distributed architectures are mostly due to the stateless nature of microservices that requires to load the model and to resolve the references lazily.

Distribution of the language services to improve the overall performances of the IDE (RQ2)

Our generator uses the specification of a communication protocol to generate microservices communicating by HTTP requests. Each microservice is associated with an HTTP resource, which is identified by a URL address. To send a request to a resource, a Domain Name System (DNS) must translate the destination URL into an IP address. This process abstracts the actual destination address of a request. We use HTTP to convey messages to microservices that can dynamically change their location after a reconfiguration of their deployment. Since microservices are by nature isolated from each other, HTTP communication allows the microservices to be distributed over different execution platforms.

We observed from Figure 7.7 that there is a real benefit on the response times to distribute some computationally intensive features (*e.g., compiler*), while others that are less demanding in terms of resources are better deployed locally to keep the best user experience (*e.g., completion* and *references*). The size of the program considered is also important on the result (cf. *rename*).

We conclude that there is an important benefit in modularizing the language services, and in distributing them in a relevant way such as we can optimize the response time of each feature individually, and improve the overall user experience of the IDE. As future work, we plan to use a learning model to estimate the pros and cons of the distribution of each feature according to the communications and the available execution platforms, and then to infer automatically the best configuration for a given context.

7.6 Conlusion

IDEs provide heterogeneous language services but lack the flexibility to fulfill their individual needs by leveraging the various execution platform that may be available. In this chapter, we proposed a generative approach to modularize and distribute language services. We proposed to complement language specifications with protocol specifications describing the language services and their communications. Based on these specifications, we can generate microservices implementing the language services and generate a Feature Model to drive their safe deployment on multiple execution platforms. We applied our approach to four languages and deployed their language services on the execution platforms in a cloud. We evaluated our approach by comparing language services implemented by microservices with language services implemented by monolithic servers and showed that it is beneficial to distribute computationally intensive language services on a cloud. However, our results also showed that stateless microservices may be costly due to the cost of loading the model which is not compensated by the computational power of the cloud execution platforms.

We have shown with this contribution that distributing language services across multiple execution platform brings more flexibility to language users since the configuration of the language service deployment allows to leverage available execution platforms.

PART III

Conclusion and Perspectives

CHAPTER 8

CONCLUSIONS

IDEs are central components for both language designers and language users. For language designers, IDEs are more and more expected to be extensible, making them a practical base for language workbenches. With IDEs, language designers have access to multiple language workbenches that can provide different technological stacks. Each technological stack has its own strength and multiple stacks could be relevant to implement a language. Having language implemented in multiple technological stacks is also interesting for the language user since he could manipulate language constructs, for a given activity, with the most appropriate language services provided by a particular language stack. However technological stacks are isolated and therefore it is not possible to benefit from multiple stacks neither for language designers or language users. IDEs are also central to language users for the development activities. To support these activities, IDEs aggregate a set of heterogeneous language services. In the meantime, language users have access to a variety of execution platforms, ranging from laptop to dedicated server or even cloud clusters, each one of these execution platforms having different capacities or resources (e.g., CPU, memory, network latency, ...), which can't be fully exploited because of the monolithic architecture of IDEs that implies to deploy every language services on the same execution platform.

For these reasons, we state in this thesis that the distribution of language services makes IDEs more flexible for both language designers and language users. We identify two challenges to be addressed to bring such flexibility. First, technological stacks should be interoperable to allow language designers to benefit from the specific strengths of multiple stacks when implementing languages, and must be interoperable to allow language users to switch seamlessly between language services implemented in different technological stacks. Second, language services should be modularized as independently deployable units and distributed to leverage the resources of the available execution platforms to best fit their needs according to the language user activities, which can evolve over time.

To address the first challenge, we proposed a communication bus to connect multiple technological stacks in which the same language is implemented and we defined a formalism to express the changes in the model incarnations. Equivalent model incarnations from the different technological stacks are kept synchronized by emitting their changes through the communication bus following a publish/subscribe pattern. We implemented our approach to connect multiple technological stacks as the framework PRISM, which is our prototype integrated to GEMOC [16]. We evaluated our first contribution on a Finite State Machine language implemented on the three technological stacks EMF, Rascal, and Java fluent API. A Finite State Machine was represented in the three technological stacks with their own language constructs. We were able to manipulate the same Finite State Machine through the language services of different technological stacks while keeping language constructs synchronized.

To address the second challenge, we defined a protocol specification language to allow language designers to describe language services and their relations. We proposed a microservices generator taking such protocol specification in input to automatize the modularization of language services. We also use the protocol specification as a basis for generating a feature model that is used to drive a deployment configurator in the safe distribution of the language services across the different execution platforms according to the constraints specified in the feature model. We implemented our generator as an Eclipse plugin that takes as input languages defined in the EMF ecosystem and produces microservices containing the language services. Our prototype allows us to dynamically change the deployment of the microservices in a Kubernetes cluster through to a configurator using the Kubernetes API. This second contribution was evaluated on the four languages NabLab, Logo, ThingML, and MiniJava. For each of these languages, we specified a protocol describing their language services, and based on this specification we were able to generate microservices implementing the language services and a feature model expressing the constraints between these services. We demonstrated the usefulness of the distribution by measuring for NabLab the response times of language services provided by a monolithic server deployed on the same laptop of the IDE client and language services implemented as microservices and deployed on a cloud. Our results show that the use of distant cloud servers improves the performances of computational intensive language services (*e.g.*, *rename* and *compile*), thereby compensating the overhead due to the distribution. Experiments also show the benefits of retaining local other features that should be reactive and less de-

manding in resources (*e.g., completion*). We also compared for the four languages the response times of the monolithic and distributed architectures, both deployed on a cloud, to investigate the cost of the distribution. Our results show that microservices have higher response times than monoliths due to our choice of making stateless microservices whereas monolithic servers are stateful. We found that the response times overhead of microservice is mainly due to the costly loading of models on each request and that the message exchange times are acceptable (i.e., less than 200 ms).

In conclusion, we brought with the works of this thesis more flexibility to language designers and users through the combination of multiple technological stacks for the design, implementation, and usage of languages and through the distribution of language services across multiple execution platforms. Although the works done in this thesis are the first steps towards the realization of IDEs as cloud-native applications, we believe that the full adoption of cloud computing and web technologies will open up new possibilities for software language engineering that will benefit both language designers and users.

CHAPTER 9

PERSPECTIVES

The work presented in this thesis is the first step toward distributed IDE to bring more flexibility in the design, implementation, and deployment of languages. We present afterward the perspectives opening to future works to improve the contributions (Section 9.1) and we close on the long-term perspectives (Section 9.2).

9.1 Contribution improvements

Model access Our approach to distributing language services across execution platforms to best meet their needs has been to modularize language services as stateless microservices. We identified the model load as a bottleneck that significantly increases the response time of language services. Access to the model is a concern that requires further investigation to identify the possible solutions and the trade-off to ensure acceptable response times.

Stateful language services could be one solution to avoid model load at each request. One direction to explore in order to support stateful language services is model replication with a synchronization mechanism to keep the multiple instances of a model contained in different language services up to date. However, statelessness can still be a good solution for language services that do not require the whole model to perform their internal logic. Keeping models in model repository and querying only parts of the model might be an appropriate solution for such services.

To determine which kind of access is appropriate for which service, we need a characterization of each service to identify the part of the model it uses and its frequency of use. Such characterization could be based on the knowledge of the language designer, who adds this informations into a protocol specification, which can be used as a basis for generating the appropriate code in the microservice. The application of static analysis of code on the existing framework implementing language services may be another

way to extract these characteristics, as well as the application of dynamic analysis on the running language services.

Collaborative model editing As mentioned in Section 6.4.2, our approach to metamorphic DSL does not account for concurrent edits of different incarnations of the same model. It does not account either for a possible distribution of the language services and the incarnations of the model over the network, which require dealing with consistency concerns such as the order of edits or their duplication. Nonetheless, we believe that the idea of exchanging patches would be a good fit for advanced scenarios such as collaborative and distributed editing of models by different stakeholders under different shapes.

Conflict-free Replicated Data Types (CRDT) is a set of data structures with merge operators designed for replicated data in a decentralized context. Concurrent editing of data is allowed by the properties of the merge operator (i.e., associativity, reflexivity and idempotency) which ensure that merging a set of divergent data in any order will always produce the same result. Each data replica can evolve independently and as long as every replica receives all the modifications of its siblings (possibly multiple times and in a different order), the properties of CRDT guarantee their eventual consistency. The use of the CRDT to express patches is a promising lead to be investigated for handling concurrent editions, possibly from distributed language services.

Communication protocol We have automated the modularization of the IDE by generating microservices implementing language services on the basis of a protocol specification. This protocol specification lets the language designer list the language services, describe their capabilities, and define their dependencies. This information allowed us to generate the source code of the microservices and to configure their communication channel with their dependencies.

However, we do not define the interface of the language services, and neither their behavior (i.e., the actual protocol between services) in the protocol specification. Enriching our specification DSL with such concepts could help increase the amount of code we generate in the microservice and thus reduce the amount of work remaining for the language designer. Making interfaces for language services explicit will bring the benefits of static typing, thus reducing the risk of deploying microservices that don't work.

Our protocol specification DSL makes it possible to define dependencies between language services but does not permit to define the order in which they are used. Specifying the behavior of the language services, at the scale of the whole IDE, could make it possible to generate dedicated source code in the microservice and thus reduce the risk of errors that can arise when the language designer manually implements these behaviors. One direction to be investigated is the service choreography (i.e., each service implementing a part of the global service coordination), which allows to deduce the individual behavior of services from the specification of global system behavior.

Generation of language implementations To benefit from our contribution allowing to combine technological stacks (Chapter 6), language designers have to handcraft every shape of a language. It may however be possible to automatically generate shapes of a language, either from a common language definition or from a shape to another. For instance, researchers have studied the generation of fluent APIs from BNF-like grammar definitions [111]. Automatic generation of shapes is not necessary for shape-diverse DSLs, but future research in this area would greatly ease their adoption.

In our approach to the distribution of language services (Chapter 7), we containerize microservices to deploy them in a cluster and allow the language user to use them through a web application. This implies that each machine has an environment capable of running containers, which limits the possible deployment locations. Although we can deploy microservices on the user's machine if it provides such an environment, we may want to deploy language services even closer to the user, directly into its browser. In our implementation, we generate Java microservices, but we believe that generating WebAssembly code could be a good target to deploy language services at the location closest to the user, thus avoiding the costs induced by network and containerization layers. The availability of the WebAssembly technology creates a new technical environment that facilitates the development of modules that can run both on the client-side and the server-side in an efficient manner.

9.2 Long-term perspectives

Patch formalism Implementers of a synchronization mechanism for shape-diverse DSLs may opt for the closed-world or open-world assumption. In the former, one assumes that all technological stacks are known beforehand, while in the latter new

technological stacks may be connected at any point in time, for instance using our *produce/apply* interface for patches. Although the closed-world assumption eases the definition of a common patch formalism on which all technological stacks agree, it hampers evolution and adaptability of the communication bus.

In PRISM, we opted for patches in the form of edit scripts [124] and were successfully able to bridge three distinct technological stacks relying on radically different theories. We cannot conclude however that the information contained in such patches is sufficient for any kind of abstract syntax formalism. In an open world especially, connecting new technological stacks raises the problem of patch evolution. In addition, if extra information (e.g., textual layout) has to be shared amongst various technological stacks, the patch formalism should be adapted accordingly. Patches are nonetheless central to our vision, as most other approaches (e.g., change propagation [128]) assume the existence of an underlying model that is not materialized in our case.

Challenges of internal DSLs We encountered a number of challenges when engineering the Java shape of our DSL (Section 6.5). These are mainly due to the fact that the domain-specific static semantics is lost when manipulating Java ASTs using domain-agnostic Java tooling. Besides, it may be hard to statically analyze the Java code manipulating models to account for reflexivity, string manipulation, or use of variables. Future work must investigate what are the limits imposed by internal DSLs in this context, especially regarding the absence of domain-specific static semantics.

Synchronization and impact management To synchronize the incarnations of a model for shape-diverse DSLs, we made the assumption that produced Patches are always well-formed. We require a model to be valid before emitting a Patch. However, one technological stack may emit Patches that are not valid for other technological stacks. For example, we observed that the semantic of the FSM language in the Java fluent API (Section 6.5) could not be expressed since we rely on the validation of Java program, which may lead to the production of ill-formed Patches. To handle ill-formed Patches, PRISM needs a mechanism to roll back a Patch when it cannot be applied to a notified technological stack.

Another improvement for PRISM is to explore how to make it possible for a language user to fix a produced ill-formed Patch since he may have the expertise to handle it. To assist the language user in this process, a correspondence model [37] could

be established by the language designer to link elements of the different shapes. With such a model, impacted model elements could be automatically inferred to guide the language user in fixing the ill-formed Patch.

Self adaptation for language services deployment We have presented an approach for distributing the language services of an IDE (Chapter 7). We proposed a deployment configurator, driven by a feature model, which allows selecting the language services to be deployed and the execution platforms. This configuration is performed manually by a language designer or a language user, depending on who has knowledge of which language service should be available and which execution platforms best meet the needs of each language service. However, taking such a decision may not be an easy task and may evolve, as the needs of language services depend on the available execution platforms and the current activity of the language user, both of which may change over time. To handle such a dynamic environment, an automatic configuration of the language services deployment would be more efficient than our manual approach. One approach for such automation could be techniques from the domain of self-adaptive systems. It will require to describe the resources of the execution platforms and the needs of the language services. Based on these descriptions, a self-adaptation of the deployment configuration can be performed. It consists in measuring metrics and selecting an adaptation strategy to satisfy quality of services goals.

Moreover, self-adapting system is a domain that could be useful for metamorphic DSL. We view the contribution presented in Chapter 6 as a first step towards *metamorphic DSLs*. Beyond the combination of multiple technological stacks, the notion of metamorphic DSL envisions self-adaptable languages that automatically adapt their shapes and the associated IDE according to a particular usage or task. The implementation of a language as a self-adapting system reacting to the activities of the language user by deploying language services from a particular shape accordingly could be a solution towards self-adaptability of languages.

9.3 IDE as Code

We envision the future of IDE as the *IDE as Code* which takes its inspiration from the *as code* approach that was applied for example in cloud computing through the YAML configuration files. The *as code* approach is the idea that a system could be de-

scribed with 'code', i.e., a textual representation. The *as code* brings the advantages of *code* (e.g., modularization, sharing, versioning, static analysis, automatization, etc) to the configuration of a system but also enforce a common formalism for all its concerns, that avoid the multiplication of configuration files in different languages.

Actual language protocols in IDE allow interoperability between client IDEs and language servers implementing general-purpose languages. Such protocols are fixed standards, which limit architectures to client/server and allow IDE configuration at language granularity. To use domain-specific languages, which may have their own specific language services, we need tailor-made protocols.

We see *IDE as Code* as the next step in the configuration of IDEs. Such configuration should be done through a language within language services are first class citizen. Such language should allow to describe the language services which include their interfaces, their interactions and their needs regarding their executing environment.

We think such description could ease the tasks of language designers, for example by detecting errors. A detailed description of language services could also allow the automatic generation of implementations targeting multiple execution environments and thus enriching the possibilities in the configuration of the IDE (e.g., choosing to deploy a language service implementation for a server or choosing an implementation for a web browser).

BIBLIOGRAPHY

[1] Hal Abelson, Nat Goodman, and Lee Rudolph, « Logo manual », *in*: (1974).

[2] Ace, *Ace*, 2021, URL: https://ace.c9.io/.

[3] Mathieu Acher, Benoit Combemale, and Philippe Collet, « Metamorphic domain-specific languages: A journey into the shapes of a language », *in*: *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, 2014, pp. 243–253.

[4] OSGI alliance, *OSGI*, 2021, URL: https://www.osgi.org/.

[5] Amazon, *Amazon EC2*, 2020, URL: https://aws.amazon.com/ec2/.

[6] Amazon, *AWS Lambda – Serverless Compute - Amazon Web Services*, 2020, URL: https://aws.amazon.com/lambda.

[7] Amazon, *Cloud9*, 2021, URL: http://c9.io/.

[8] Apache, *NetBeans*, 2021, URL: http://netbeans.apache.org/.

[9] Apache, *Zeppelin*, 2021, URL: https://zeppelin.apache.org/.

[10] Apple, *Xcode*, 2021, URL: https://developer.apple.com/xcode/.

[11] Ioana Baldini et al., « Serverless computing: Current trends and open problems », *in*: *Research Advances in Cloud Computing*, Springer, 2017, pp. 1–20.

[12] Ankica Barisic et al., « How to reach a usable DSL? Moving toward a Systematic Evaluation », *in*: *Electronic Communications of the EASST* 50 (2012).

[13] Francesco Basciani et al., « MDEForge: an Extensible Web-Based Modeling Platform. », *in*: *CloudMDE@ MoDELS*, 2014, pp. 66–75.

[14] Tim Berners-Lee, Roy Fielding, and Larry Masinter, *Uniform resource identifier (URI): Generic syntax*, tech. rep., 2004.

[15] Lorenzo Bettini, *Implementing domain-specific languages with Xtext and Xtend*, Packt Publishing Ltd, 2016.

[16] Erwan Bousse et al., « Execution framework of the gemoc studio (tool demo) », *in*: *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, ACM, 2016, pp. 84–89.

[17] Hendrik Bünder, « Decoupling Language and Editor-The Impact of the Language Server Protocol on Textual Domain-Specific Languages. », *in*: *MODELSWARD*, 2019, pp. 129–140.

[18] Brendan Burns et al., « Borg, omega, and kubernetes », *in*: *Queue* 14.1 (2016), pp. 70–93.

[19] Carlos Carrascal Manzanares, Jesús Sánchez Cuadrado, and Juan de Lara, « Building MDE cloud services with DISTIL », *in*: *CEUR Workshop Proceedings*, CEUR-WS, 2015.

[20] Carlos Carrascal-Manzanares, Jesús Sánchez Cuadrado, and Juan de Lara, « Building MDE cloud services with DISTIL », *in*: *International Conference on Model Driven Engineering Languages and Systems*, vol. 1563, Model-Driven Engineering on and for the Cloud, Ottawa, Canada: CEUR Workshop Proceedings, Sept. 2015, pp. 19–24, URL: https://hal.archives-ouvertes.fr/hal-01761670.

[21] Tomas Cerny, Michael J Donahoo, and Michal Trnka, « Contextual understanding of microservice architecture: current and future directions », *in*: *ACM SIGAPP Applied Computing Review* 17.4 (2018), pp. 29–45.

[22] Betty HC Cheng et al., « On the globalization of domain-specific languages », *in*: *Globalizing Domain-Specific Languages*, Springer, 2015, pp. 1–6.

[23] Codeanywhere, *Codeanywhere*, 2021, URL: https://codeanywhere.com/.

[24] CodeMirror, *CodeMirror*, 2021, URL: https://codemirror.net/.

[25] CodePen, *CodePen*, 2021, URL: https://codepen.io/.

[26] Object Computing, *Micronaut*, 2021, URL: https://micronaut.io/.

[27] Jonathan Corley, Eugene Syriani, and Huseyin Ergin, « Evaluating the cloud architecture of AToMPM », *in*: *2016 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, IEEE, 2016, pp. 339–346.

[28] Valerio Cosentino, Massimo Tisi, and Javier Luis Cánovas Izquierdo, « A model-driven approach to generate external dsls from object-oriented apis », *in*: *International Conference on Current Trends in Theory and Practice of Informatics*, Springer, 2015, pp. 423–435.

[29] Fabien Coulon et al., « Modular and Distributed IDE », *in*: *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2020, Virtual, USA: Association for Computing Machinery, 2020, pp. 270–282, ISBN: 9781450381765, DOI: 10.1145/3426425.3426947, URL: https://doi.org/10.1145/3426425.3426947.

[30] Fabien Coulon et al., « Shape-diverse DSLs: languages without borders (vision paper) », *in*: *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering*, 2018, pp. 215–219.

[31] K. Czarnecki and A. Wasowski, « Feature Diagrams and Logics: There and Back Again », *in*: *11th International Software Product Line Conference (SPLC 2007)*, Sept. 2007, pp. 23–34, DOI: 10.1109/SPLINE.2007.24.

[32] Arie Van Deursen and Paul Klint, « Little languages: little maintenance? », *in*: *Journal of Software Maintenance: Research and Practice* 10.2 (1998), pp. 75–92.

[33] Devbox, *Devbox*, 2021, URL: https://devbox.ewave.com/#/.

[34] Abhijit Dubey and Dilip Wagle, « Delivering software as a service », *in*: *The McKinsey Quarterly* 6.2007 (2007), p. 2007.

[35] Thomas F Düllmann and André van Hoorn, « Model-driven generation of microservice architectures for benchmarking performance and resilience engineering approaches », *in*: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, 2017, pp. 171–172.

[36] Thomas F. Düllmann and André van Hoorn, « Model-Driven Generation of Microservice Architectures for Benchmarking Performance and Resilience Engineering Approaches », *in*: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, ICPE '17 Companion, Association for Computing Machinery, 2017, pp. 171–172, ISBN: 9781450348997, DOI: 10.1145/3053600.3053627, URL: https://doi.org/10.1145/3053600.3053627.

[37] Sophie Ebersold, « Modelisation des systemes complexes et Points de vue: l'Ingenierie des Modeles centree utilisateur pour l'Ingenierie Systeme », PhD thesis, Universite de Toulouse, 2021.

[38] Eclipse, *Eclipse IDE*, 2021, URL: https://www.eclipse.org/ide/.

[39] Eclipse, *Eclipse Orion*, 2021, URL: https://projects.eclipse.org/projects/ecd.orion.

[40] Eclipse, *MicroProfile*, 2021, URL: https://microprofile.io/.

[41] Eclipse Foundation, *Eclipse Che | Eclipse Next-Generation IDE for developer teams*, [Online; accessed 25-February-2020], 2020, URL: https://www.eclipse.org/che/.

[42] Sebastian Erdweg et al., « Evaluating and comparing language workbenches: Existing results and benchmarks for the future », *in*: *Computer Languages, Systems & Structures* 44 (2015), pp. 24–47.

[43] Moritz Eysholdt and Heiko Behrens, « Xtext: Implement Your Language Faster than the Quick and Dirty Way », *in*: *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '10, Reno/Tahoe, Nevada, USA: Association for Computing Machinery, 2010, pp. 307–309, ISBN: 9781450302401, DOI: 10.1145/1869542.1869625, URL: https://doi.org/10.1145/1869542.1869625.

[44] Moritz Eysholdt and Heiko Behrens, « Xtext: implement your language faster than the quick and dirty way. », *in*: Jan. 2010, pp. 307–309, DOI: 10.1145/1869542.1869625.

[45] Jean-Marie Favre et al., « Empirical language analysis in software linguistics », *in*: *International Conference on Software Language Engineering*, Springer, 2010, pp. 316–326.

[46] Matthias Felleisen et al., « A programmable programming language », *in*: *Communications of the ACM* 61.3 (2018), pp. 62–71.

[47] OpenJS fondation, *Electron*, 2021, URL: https://www.electronjs.org/.

[48] Martin Fowler, « Fluent Interface.(2005) », *in*: *URL http://martinfowler. com/bliki/FluentInterface. html* (2005).

[49] Martin Fowler, *Language workbenches: The killer-app for domain specific languages*, 2005.

[50] Martin Fowler, *UML distilled: a brief guide to the standard object modeling language*, Addison-Wesley Professional, 2004.

[51] George Fylaktopoulos et al., « A distributed modular platform for the development of cloud based applications », *in*: *Future Generation Computer Systems* 78 (2018), pp. 127–141.

[52] George Fylaktopoulos et al., « CIRANO: An integrated programming environment for multi-tier cloud based applications », *in*: *Procedia Computer Science* 68 (2015), pp. 42–52.

[53] GitHub, *Atom*, 2021, URL: https://atom.io/.

[54] GitHub, *Codespaces*, 2021, URL: https://github.com/features/codespaces.

[55] Google, *Android Studio*, 2021, URL: https://developer.android.com/studio.

[56] Google, *Google App Engine*, 2020, URL: https://appengine.google.com/.

[57] Giona Granchelli et al., « Microart: A software architecture recovery tool for maintaining microservice-based systems », *in*: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, IEEE, 2017, pp. 298–302.

[58] Claudio Guidi et al., « Microservices: a Language-based Approach », *in*: *Present and Ulterior Software Engineering*, ed. by Manuel Mazzara and Bertrand Meyer, https://hal.inria.fr/hal-01635817: Springer, Nov. 2017, URL: https://hal.inria.fr/hal-01635817.

[59] Claudio Guidi et al., « Microservices: a Language-based Approach », *in*: *CoRR* abs/1704.08073 (2017), arXiv: 1704.08073, URL: http://arxiv.org/abs/1704.08073.

[60] David Harel and Bernhard Rumpe, « Meaningful modeling: what's the semantics of" semantics"? », *in*: *Computer* 37.*10* (2004), pp. 64–72.

[61] Nicolas Harrand et al., « ThingML: a language and code generation framework for heterogeneous targets », *in*: *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, 2016, pp. 125–135.

[62] Sara Hassan, Nour Ali, and Rami Bahsoon, « Microservice ambients: An architectural meta-modelling approach for microservice granularity », *in*: *2017 IEEE International Conference on Software Architecture (ICSA)*, IEEE, 2017, pp. 1–10.

[63] Sara Hassan and Rami Bahsoon, « Microservices and their design trade-offs: A self-adaptive roadmap », *in*: *2016 IEEE International Conference on Services Computing (SCC)*, IEEE, 2016, pp. 813–818.

[64] Sara Hassan, Rami Bahsoon, and Rick Kazman, « Microservice transition and its granularity problem: A systematic mapping study », *in*: *Software: Practice and Experience* 50.*9* (2020), pp. 1651–1681.

[65] Red Hat, *Quarkus*, 2021, URL: https://quarkus.io/.

[66] Jan Heering et al., « The syntax definition formalism sdf—reference manual— », *in*: *ACM Sigplan Notices* 24.*11* (1989), pp. 43–75.

[67] Jay Heiser and John Santoro, *Hype cycle for software as a service*, 2019.

[68] Soichiro Hidaka, Frédéric Jouault, and Massimo Tisi, « On Additivity in Transformation Languages », *in*: *20th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2017, Austin, TX, USA, September 17-22, 2017*, 2017, pp. 23–33, DOI: 10.1109/MODELS.2017.21, URL: https://doi.org/10.1109/MODELS.2017.21.

[69] Charles Antony Richard Hoare, « An axiomatic basis for computer programming », *in*: *Communications of the ACM* 12.*10* (1969), pp. 576–580.

[70] Christian Hofer and Klaus Ostermann, « Modular domain-specific language components in scala », *in*: *ACM SIGPLAN Notices*, vol. 46, 2, ACM, 2010, pp. 83–92.

[71] Sublime HQ, *Sublime Text*, 2021, URL: https://www.sublimetext.com/.

[72] Paul Hudak, « Building domain-specific embedded languages », *in*: *Acm computing surveys (csur)* 28.*4es* (1996), 196–es.

[73] JetBrains, *IntelliJ IDEA*, 2021, URL: https://www.jetbrains.com/fr-fr/idea/.

[74] JetBrains, *PyCharm*, 2021, URL: https://www.jetbrains.com/fr-fr/pycharm/.

[75] JSFiddle, *JSFiddle*, 2021, URL: https://jsfiddle.net/.

[76] Jupyter, *Jupyter*, 2021, URL: https://jupyter.org/.

[77] Kyo C Kang et al., *Feature-oriented domain analysis (FODA) feasibility study*, tech. rep., Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, 1990.

[78] Lennart CL Kats and Eelco Visser, « The spoofax language workbench: rules for declarative specification of languages and IDEs », *in*: *ACM sigplan notices*, vol. 45, 10, ACM, 2010, pp. 444–463.

[79] Lennart CL Kats et al., « Software development environments on the web: a research agenda », *in*: *Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software*, 2012, pp. 99–116.

[80] Sven Keidel, Wulf Pfeiffer, and Sebastian Erdweg, « The IDE Portability Problem and Its Solution in Monto », *in*: *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2016, Amsterdam, Netherlands: Association for Computing Machinery, 2016, pp. 152–162, ISBN: 9781450344470, DOI: 10.1145/2997364.2997368, URL: https://doi.org/10.1145/2997364.2997368.

[81] Anneke Kleppe, *Software language engineering: creating domain-specific languages using metamodels*, Pearson Education, 2008.

[82] Paul Klint, Ralf Lämmel, and Chris Verhoef, « Toward an engineering discipline for grammarware », *in*: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 14.3 (2005), pp. 331–380.

[83] Paul Klint and Tijs van der Storm, « Model Transformation with Immutable Data », *in*: *International Conference on Theory and Practice of Model Transformations*, Springer, 2016, pp. 19–35.

[84] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju, « EASY Meta-Programming with Rascal. Leveraging the Extract-Analyze-SYnthesize Paradigm for Meta-Programming », *in*: *Proceedings of the 3rd International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'09)*, LNCS, Braga, Portugal: Springer, 2010.

[85] Sander Klock et al., « Workload-based clustering of coherent feature sets in microservice architectures », *in*: *2017 IEEE International Conference on Software Architecture (ICSA)*, IEEE, 2017, pp. 11–20.

[86] Tomaž Kosar, Sudev Bohra, and Marjan Mernik, « Domain-specific languages: A systematic mapping study », *in*: *Information and Software Technology* 71 (2016), pp. 77–91.

[87] Tomaž Kosar et al., « Program comprehension of domain-specific and general-purpose languages: replication of a family of experiments using integrated development environments », *in*: *Empirical Software Engineering* 23.5 (2018), pp. 2734–2763.

[88] Nane Kratzke and Peter-Christian Quint, « Understanding cloud-native applications after 10 years of cloud computing-a systematic mapping study », *in*: *Journal of Systems and Software* 126 (2017), pp. 1–16.

[89] Ivan Kurtev, Jean Bézivin, and Mehmet Aksit, « Technological Spaces: An Initial Appraisal », *in*: (Jan. 2002).

[90] Ralf Lammel and Chris Verhoef, « Cracking the 500-language problem », *in*: *IEEE software* 18.6 (2001), pp. 78–88.

[91] Ralf Lämmel, *Software languages: Syntax, semantics, and metaprogramming*, Springer, 2018.

[92] Ralf Lämmel and Erik Meijer, « Mappings make data processing go'round », *in*: *International Summer School on Generative and Transformational Techniques in Software Engineering*, Springer, 2005, pp. 169–218.

[93] Benoit Lelandais, Marie-Pierre Oudot, and Benoit Combemale, « Fostering Metamodels and Grammars within a Dedicated Environment for HPC: The NabLab Environment (Tool Demo) », *in*: *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2018, Boston, MA, USA: Association for Computing Machinery, 2018, pp. 200–204, ISBN: 9781450360296, DOI: 10.1145/3276604.3276620, URL: https://doi.org/10.1145/3276604.3276620.

[94] Benoit Lelandais, Marie-Pierre Oudot, and Benoit Combemale, « Fostering metamodels and grammars within a dedicated environment for HPC: the NabLab environment (tool demo) », *in*: *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering*, 2018, pp. 200–204.

[95] James Lewis and Martin Fowler, *Microservices - a definition of this new architectural term*, 2014, URL: https://martinfowler.com/articles/microservices.html.

[96] Frank Leymann et al., « Native cloud applications: why monolithic virtualization is not their foundation », *in*: *International Conference on Cloud Computing and Services Science*, Springer, 2016, pp. 16–40.

[97] Frank Leymann et al., « Native cloud applications: Why virtual machines, images and containers miss », *in*: *Proceedings of the 6th International Conference on Cloud Computing and*, SciTePress, pp. 7–15.

[98] Miklós Maróti et al., « Next generation (meta) modeling: web-and cloud-based collaborative tool infrastructure. », *in*: *MPM@ MoDELS* 1237 (2014), pp. 41–60.

[99] Peter Mell, Tim Grance, et al., « The NIST definition of cloud computing », *in*: (2011).

[100] Dirk Merkel, « Docker: lightweight linux containers for consistent development and deployment », *in*: *Linux journal* 2014.*239* (2014), p. 2.

[101] Marjan Mernik, Jan Heering, and Anthony M Sloane, « When and how to develop domain-specific languages », *in*: *ACM computing surveys (CSUR)* 37.*4* (2005), pp. 316–344.

[102] Microsoft, *Cloud Computing Services | Microsoft Azure*, 2020, URL: https://azure.microsoft.com/.

[103] Microsoft, *Debug Adapter Protocol*, 2021, URL: https://microsoft.github.io/debug-adapter-protocol/.

[104] Microsoft, *Language Server Protocol*, 2021, URL: https://microsoft.github.io/language-server-protocol/.

[105] Microsoft, *Monaco*, 2021, URL: https://microsoft.github.io/monaco-editor/.

[106] Microsoft, *Visual Studio*, 2021, URL: https://visualstudio.microsoft.com/fr/.

[107] Microsoft, *VSCode*, 2021, URL: https://code.visualstudio.com/.

[108] Matt Morley, *JSON-RPC*, 2021, URL: https://www.jsonrpc.org/specification.

[109] Peter D Mosses, « The varieties of programming language semantics and their uses », *in*: *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, Springer, 2001, pp. 165–190.

[110] Moh Afifun Naily et al., « A framework for modelling variable microservices as software product lines », *in*: *International Conference on Software Engineering and Formal Methods*, Springer, 2017, pp. 246–261.

[111] Tomoki Nakamaru et al., « Silverchain: a fluent API generator », *in*: *ACM SIGPLAN Notices* 52.*12* (2017), pp. 199–211.

[112] Leandro Marques do Nascimento et al., « A systematic mapping study on domain-specific languages », *in*: *The Seventh International Conference on Software Engineering Advances (ICSEA 2012)*, 2012, pp. 179–187.

[113] Netflix, *Netflix*, 2020, URL: https://www.netflix.com/.

[114] Jakob Nielsen, *Usability engineering*, Morgan Kaufmann, 1994.

[115] Obeo, *Sirius Web*, 2021, URL: https://www.eclipse.org/sirius/sirius-web.html.

[116] Ernst Oberortner, Uwe Zdun, and Schahram Dustdar, « Tailoring a model-driven quality-of-service DSL for various stakeholders », *in*: *2009 ICSE Workshop on Modeling in Software Engineering*, IEEE, 2009, pp. 20–25.

[117] Deepak Puthal et al., « Cloud computing features, issues, and challenges: a big picture », *in*: *2015 International Conference on Computational Intelligence and Networks*, IEEE, 2015, pp. 116–123.

[118] Andres J Ramirez and Betty HC Cheng, « Design patterns for developing dynamically adaptive systems », *in*: *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, 2010, pp. 49–58.

[119] Andres J. Ramirez and Betty H. C. Cheng, « Design Patterns for Developing Dynamically Adaptive Systems », *in*: *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '10, Cape Town, South Africa: Association for Computing Machinery, 2010, pp. 49–58, ISBN: 9781605589718, DOI: 10.1145/1808984.1808990, URL: https://doi.org/10.1145/1808984.1808990.

[120] Repl.it, *Repl.it*, 2021, URL: https://repl.it/.

[121] Eric Roberts, « An overview of MiniJava », *in*: *ACM SIGCSE Bulletin* 33.1 (2001), pp. 1–5.

[122] Roberto Rodriguez-Echeverria et al., « An LSP infrastructure to build EMF language servers for web-deployable model editors. », *in*: *MODELS Workshops*, 2018, pp. 326–335.

[123] Roberto Rodriguez-Echeverria et al., « Towards a language server protocol infrastructure for graphical modeling », *in*: *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, 2018, pp. 370–380.

[124] Riemer van Rozen and Tijs van der Storm, « Toward live domain-specific languages: From text differencing to adapting models at run time », *in*: (Aug. 2017).

[125] Newman Sam, « Building microservices », *in*: *O'Reilly Media, Inc.* (2015).

[126] Jetbrains Scala Center, *Build Server Protocol*, 2021, URL: https://build-server-protocol.github.io/.

[127] Bran Selic, « A systematic approach to domain-specific language design using UML », *in*: *Object and Component-Oriented Real-Time Distributed Computing, 2007. ISORC'07. 10th IEEE International Symposium on*, IEEE, 2007, pp. 2–9.

[128] Oszkár Semeráth et al., « Change Propagation of View Models by Logic Synthesis using SAT solvers », *in*: *Proceedings of the 5th International Workshop on Bidirectional Transformations, Bx 2016, co-located with The European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 8, 2016.* 2016, pp. 40–44, URL: http://ceur-ws.org/Vol-1571/paper%5C_6.pdf.

[129] Sourcegraph, *Language Server*, 2021, URL: https://langserver.org/.

[130] Diomidis Spinellis, « Notable design patterns for domain-specific languages », *in*: *Journal of systems and software* 56.1 (2001), pp. 91–99.

[131] Megan Squire and Amber K Smith, « The diffusion of pastebin tools to enhance communication in floss mailing lists », *in*: *IFIP International Conference on Open Source Systems*, Springer, 2015, pp. 45–57.

[132] Dave Steinberg et al., *EMF: eclipse modeling framework*, Pearson Education, 2008.

[133] Dave Steinberg et al., *EMF: eclipse modeling framework*, Pearson Education, 2008.

[134] Eugene Syriani et al., « AToMPM: A Web-based Modeling Environment. », *in*: *Demos/Posters/StudentResearch@ MoDELS* 2013 (2013), pp. 21–25.

[135] Genc Tato et al., « ShareLatex on the Edge: Evaluation of the Hybrid Core/Edge Deployment of a Microservices-Based Application », *in*: *Proceedings of the 3rd Workshop on Middleware for Edge Clouds & Cloudlets*, MECC'18, Rennes, France: Association for Computing Machinery, 2018, pp. 8–15, ISBN: 9781450361170, DOI: 10.1145/3286685.3286687, URL: https://doi.org/10.1145/3286685.3286687.

[136] Genc Tato et al., « ShareLatex on the Edge: Evaluation of the Hybrid Core/Edge Deployment of a Microservices-based Application », *in*: *MECC 2018 - 3rd Workshop on Middleware for Edge Clouds & Cloudlets*, Rennes, France, Dec. 2018, pp. 1–6, URL: https://hal.inria.fr/hal-01942807.

[137] Andre LC Tavares and Marco Tulio Valente, « A gentle introduction to OSGi », *in*: *ACM SIGSOFT Software Engineering Notes* 33.5 (2008), pp. 1–5.

[138] Branko Terzic et al., « Development and evaluation of MicroBuilder: a Model-Driven tool for the specification of REST Microservice Software Architectures », *in*: *ENTERPRISE INFORMATION SYSTEMS* 12.8-9 (2018), pp. 1034–1057.

[139] Thomas Thüm et al., « FeatureIDE: An extensible framework for feature-oriented software development », *in*: *Science of Computer Programming* 79 (2014), pp. 70–85.

[140] Leonardo P Tizzei et al., « Using microservices and software product line engineering to support reuse of evolving multi-tenant saas », *in*: *Proceedings of the 21st International Systems and Software Product Line Conference-Volume A*, 2017, pp. 205–214.

[141] Google Trends, *Top IDE index*, 2021, URL: https://pypl.github.io/IDE.html.

[142] TypeFox, *Gitpod*, 2021, URL: https://www.gitpod.io/.

[143] TypeFox, *Theia*, 2021, URL: https://theia-ide.org/.

[144] Uber, *Uber Eats*, 2020, URL: http://ubereats.com/.

[145] Arie Van Deursen, Paul Klint, and Joost Visser, « Domain-specific languages: An annotated bibliography », *in*: *ACM Sigplan Notices* 35.6 (2000), pp. 26–36.

[146] Eelco Visser et al., « A language designer's workbench: a one-stop-shop for implementation and verification of language designs », *in*: *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, 2014, pp. 95–111.

[147] Vladimir Viyović, Mirjam Maksimović, and Branko Perisić, « Sirius: A rapid development of DSM graphical editor », *in*: *IEEE 18th International Conference on Intelligent Engineering Systems INES 2014*, IEEE, 2014, pp. 233–238.

[148] VMware, *Spring*, 2021, URL: https://spring.io/.

[149] Markus Voelter, *Generic tools, specific languages*, Citeseer, 2014.

[150] Markus Voelter, « Language and IDE Modularization and Composition with MPS », *in*: *Generative and Transformational Techniques in Software Engineering IV: International Summer School, GTTSE 2011, Braga, Portugal, July 3-9, 2011. Revised Papers*, 2013, pp. 383–430, DOI: 10.1007/978-3-642-35992-7_11".

[151] Markus Voelter and Sascha Lisson, « Supporting Diverse Notations in MPS'Projectional Editor. », *in*: *GEMOC@ MoDELS*, 2014, pp. 7–16.

[152] Markus Voelter and Vaclav Pech, « Language modularity with the MPS language workbench », *in*: *2012 34th International Conference on Software Engineering (ICSE)*, IEEE, 2012, pp. 1449–1450.

[153] Markus Voelter, Jos Warmer, and Bernd Kolb, « Projecting a modular future », *in*: *IEEE Software* 32.5 (2015), pp. 46–52.

[154] Markus Voelter et al., *DSL engineering: Designing, implementing and using domain-specific languages*, dslbook. org, 2013.

[155] Joe Weinman, « Hybrid cloud economics », *in*: *IEEE Cloud Computing* 3.1 (2016), pp. 18–22.

[156] Niklaus Wirth, « Extended backus-naur form (ebnf) », *in*: *Iso/Iec* 14977.*2996* (1996), pp. 2–1.

[157] Ling Wu et al., « CEclipse: An online IDE for programing in the cloud », *in*: *2011 IEEE World Congress on Services*, IEEE, 2011, pp. 45–52.

[158] Chengzhi Xu et al., « Caople: A programming language for microservices saas », *in*: *2016 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, IEEE, 2016, pp. 34–43.

[159] Vladimir Yussupov et al., « Pattern-based Modelling, Integration, and Deployment of Microservice Architectures », *in*: *2020 IEEE 24th International Enterprise Distributed Object Computing Conference (EDOC)*, IEEE, 2020, pp. 40–50.

[160] Qi Zhang, Lu Cheng, and Raouf Boutaba, « Cloud computing: state-of-the-art and research challenges », *in*: *Journal of internet services and applications* 1.*1* (2010), pp. 7–18.

[161] Olaf Zimmermann, « Microservices tenets », *in*: *Computer Science-Research and Development* 32.*3-4* (2017), pp. 301–310.

[162] Philip Zweihoff, Stefan Naujokat, and Bernhard Steffen, « Pyro: Generating domain-specific collaborative online modeling environments », *in*: *International Conference on Fundamental Approaches to Software Engineering*, Springer, 2019, pp. 101–115.

DOCTORAT BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

Titre : Vers des Environnements de Développement Intégrés flexibles

Mot clés : IDE, DSL Metamorphique, Microservice, Approche générative

Résumé : Les Environnements de Développement Intégrés (EDI) sont des composants centraux pour les utilisateurs de langages, qui permettent les activités de développement et fournissent un accès unifié aux services de langages. Les EDIs sont également essentiels pour les concepteurs de languages, qui s'attendent de plus en plus à ce que les EDIs soient extensibles, ce qui en fait une base pratique pour les ateliers de langages.

Bien que les concepteurs de langage aient la possibilité d'implémenter les langages dans la pile technologique de leur choix, ils manquent de flexibilité pour cette tâche. En effet, un langage peut être implémenté dans différentes piles technologiques mais les services de langages des différentes piles ne peuvent pas manipuler les mêmes constructions de langage en raison de l'isolement des piles technologiques. Une telle flexibilité est intéressante pour les concepteurs de langage, car chaque pile a ses points forts, et est intéressante pour les utilisateurs de langage car, pour une activité donnée, ils manipuleront les constructions de langage avec les services de langage les plus appropriés fournis par une pile technologique spécifique.

La tendance récente dans l'ingénierie des EDIs est d'adopter le modèle Software as a Service pour réaliser les EDIs en tant qu'applications cloud, l'objectif étant de fournir un environnement de développement sans installation via un navigateur web. Les clouds sont composés de plateformes d'exécution qui peuvent varier dans les ressources qu'elles fournissent (mémoire, CPU, etc.) et, en même temps, les EDIs sont composés de services de langage hétérogènes ayant des besoins spécifiques. Cependant, les EDIs existants basés sur le cloud ne tirent pas parti de la diversité des ressources disponibles pour répondre au mieux aux besoins de leurs services de langage car leurs architectures sont monolithiques, ce qui interdit toute flexibilité dans le déploiement des languages.

La flexibilité dans la réalisation et le déploiement des langages peut se résumer à deux défis. Premièrement, les concepteurs de langage doivent pouvoir bénéficier des atouts spécifiques des nombreuses piles technologiques possibles pour la réalisation des langages et, d'autre part, les utilisateurs de langage doivent pouvoir passer de manière transparente entre les services de langage réalisés dans différentes piles technologiques pour manipuler les mêmes constructions de langage. Deuxièmement, les services de langage doivent exploiter les plateformes d'exécution disponibles pour répondre au mieux à leurs besoins en fonction des activités de l'utilisateur du langage, qui peuvent évoluer dans le temps. À cette fin, les services de langage doivent être distribués et leur déploiement doit être configurable.

Cette thèse apporte plus de flexibilité à la fois dans l'implémentation du langage et dans le déploiement des services du langage. Pour relever le défi de l'implémentation d'un langage dans plusieurs piles technologiques, nous proposons un bus de communication basé sur un motif de type publication/souscription et un formalisme pour exprimer les changements dans les constructions du langage afin de synchroniser les piles technologiques implémentant le même langage. Nous avons appliqué cette contribution aux trois piles technologiques EMF, Rascal, et Java fluent API dans

lesquelles le langage Machine à Etats a été implémenté.

Afin de relever le défi de la distribution et de la configuration du déploiement des services de langage, nous proposons une approche générative, basée sur la spécification d'un protocole de communication, pour modulariser les services de langage et pour permettre un déploiement sûr. Pour valider la généralisation de notre deuxième contribution, nous l'avons appliquée aux langages NabLab, Logo, MiniJava et ThingML. Nous avons mesuré qu'il y a un avantage à exécuter des services intensifs en calcul en tant que microservices déployés à distance et que les microservices sans état ont une surcharge significative due au chargement de modèle.

Cette thèse a permis de tirer profit des forces de plusieurs piles technologiques pour implémenter des langages en connectant des ateliers de langages de ces piles et a permis de distribuer des services de langage sur différentes plateformes d'exécution en fonction de leurs besoins grâce à la spécification d'un protocole de communication combiné à une approche générative. Nous pensons que ce travail est un premier pas vers un EDI auto-adaptatif capable de réagir automatiquement aux changements dans les activités des utilisateurs de lanngage et aux changements dans les plateformes d'exécution disponibles.

Title: Towards flexible Integrated Development Environment

Keywords: IDE, Metamorphic DSL, Microservice, Generative approach

Abstract: Integrated Development Environments (IDE) are central components for language users to support development activities and provide unified access to language services. IDEs are essential for language designers as well, who increasingly expect IDEs to be extensible, making them a practical base for language workbenches.

Although language designers have the choice to implement languages in the technological stack of their choice, they lake of flexibility for this task. Indeed, a language can be implemented in different technological stacks but the language services of the different stacks cannot manipulate the same language constructs due to the isolation of technological stacks. Such flexibility is interesting for language designers, because each stack has its strengths, and is interesting for language users because, for a given activity, they would manipulate language constructs with the most appropriate language services provided by a specific technological stack.

The recent trend in IDE engineering is to adopt the Software as a Service model for implementing IDEs as cloud applications, the goal being to provide a development environment without installation through a web browser. Clouds are composed of execution platforms that can vary in the resources they provide (memory, CPU, etc) and at the same time IDEs are composed of heterogeneous language services with specific needs. However existing cloud IDEs do not take advantage of the diversity of available resources to best fit the needs of their language services because their architectures are monolithic, which prohibits any flexibility in language deployment.

Flexibility in the implementation and deployment of languages can be summarized as two challenges. First, language designers must be able to benefit from the specific strengths of the many possible technological stacks for implementing languages and, on the other side, language users must be able to seamlessly switch between language services implemented in different technologi-

cal stacks to manipulate the same language constructs. Second, language services must leverage available execution platforms to best fit their needs according to the language user's activities, which can evolve over time. To this end, language services have to be distributed and their deployment has to be configurable.

This thesis brings more flexibility in both language implementation and language services deployment. For the challenge of implementing a language in multiple technological stacks, we propose a communication bus based on a publish/subscribe pattern and a formalism to express changes in language constructs to synchronize technological stacks implementing the same language. We applied this contribution to the three technological stacks EMF, Rascal, and Java fluent API in which the Finite State Machine language has been implemented.

To address the challenge of distributing and configuring the deployment of language services, we propose a generative approach to modularize language services and support their safe deployment based on the specification of a communication protocol. To validate our second contribution is generalizable, we applied it to the NabLab, Logo, MiniJava, and ThingML languages. We measured that there is a benefit in running computational intensive services as remotely deployed microservices and that stateless microservices have a significant overhead due to model loading.

This thesis made it possible to take advantage of the strengths of multiple technological stacks to implement languages by connecting language workbenches and to distribute language services across different execution platforms according to their needs through the specification of communication protocol combined with a generative approach. We believe that this work is a first step towards a self-adaptive IDE capable of automatically reacting to changes in the activities of language users and in available execution platforms.