On modularity and performance of External Domain-Specific Language implementations

Manuel Leduc

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

THÈSE DE DOCTORAT DE

L'UNIVERSITE DE RENNES 1
COMUE **U**NIVERSITE **B**RETAGNE **L**OIRE

Ecole Doctorale N°601
Mathèmatique et Sciences et Technologies de l'Information et de la Communication
Spécialité : Informatique

Par

Manuel LEDUC

On modularity and performance of External Domain-Specific Language implementations

Thèse présentée et soutenue à RENNES , le 18 Décembre 2019
Unité de recherche : Equipe DiverSE, IRISA

Rapporteurs avant soutenance :
Paul Klint, Professeur, CWI, Amsterdam, Pays-Bas
Juan De Lara, Professeur, Université autonome de Madrid, Madrid, Espagne

Composition du jury :

Président : Olivier Ridoux, Professeur, Université de Rennes 1, Rennes, France
Examinateurs : Nelly Bencomo, Lecturer in Computer Science, Aston University, Birmingham, Royaume-Uni
Jean-Remy Falleri, Maître de conférences HDR, Université de Bordeaux, Bordeaux, France
Gurvan Le Guernic, Ingénieur de recherche, DGA, Rennes, France
Dir. de thèse : Benoît Combemale, Professeur, UT2J, Toulouse, France
Co-dir. de thèse : Olivier Barais, Professeur, Université de Rennes 1, Rennes, France

Invité

Thomas Degueule, Chargé de Recherche, CWI, Amsterdam, Pays-Bas

ACKNOWLEDGEMENT

First of all, I would like to thank all the members of the jury for accepting to review my work. In particular, I wish to thank Professor Juan De Lara and Professor Paul Klint for thoroughly reviewing this report.

Those three years have been a very intense and enriching experience. For that, I want to thank my advisors, Benoît Combemale and Olivier Barais. I also want to highlight the infinite energy and passion that they invest in their work. This has been an incredible thing to witness.

I also want to thank Thomas, who had the patience to write with me and to be available to answer my numerous questions. You helped me untangle the puzzle that scientific writing is.

Then, I want to thank Gurvan, who gave me an outside view of my research and helped me question my work.

Continuing, I would like to thank the members of the SWAT team. It has been a pleasure to visit your team and to collaborate on the ALE project.

And as a final note on the professional and scientific side, I want to thank the DiverSE team as a whole. I am impressed by the energy, creativity, and spirit of its members and I'm proud to have shared so many coffee breaks with you all.

Moving to a more personal note, I first want to thank the "coureurs d'extrême" running club. The weekly training sessions have been an awesome way to clear my mind after work. The great advice and cheerfulness of everybody brought me from slow-paced 10k races to ultra-trails during the span of those three years.

It goes without saying that I thank everybody from "Robidou" and my family. Your attentions have helped me through this challenge.

Finally, my last acknowledgment goes to Flora. Three wonderful years, thank you for your infinite support and encouragements, even when you were more exhausted than I was.

TABLE OF CONTENTS

INTRODUCTION EN FRANÇAIS

Contexte de recherche

Un challenge important de l'ingénierie logicielle est le développement et la maintenance de systèmes complexes. Des exemples récents de tels systèmes sont l'internet des objets ou les systèmes cyber-physiques. La conception de ces systèmes complexes implique de nombreux partie prenante, avec des points de vue variés et hétérogènes. Ce qui soulève de nombreux défis d'ingénierie. Par conséquent, le succès de la conception de systèmes complexes requière l'application de méthodes et d'outils d'ingénierie spécifiques.

Une solution dédiée au développement de systèmes complexes est l'Ingénierie Dirigée par les Modèles (IDM). Plus précisément, l'IDM promeut la réduction de la complexité accidentelle inhérente au développement de systèmes complexes en faisant le pont entre l'espace des problèmes et l'espace des solutions à l'aide de modèles. Un modèle est une représentation d'un aspect spécifique d'un système et fournis des abstractions pertinentes aux experts métiers. Même si les modèles eux même sont des vues spécifiques sur un aspect d'un système, leur représentation est par défaut générique. Toutes fois, de nombreuses études montrent que les modèles sont mieux appréhendés sous la forme de représentation spécifiques [75, 117]. Les langages (informatiques) dédiés sont une solution couramment utilisée pour la réalisation de tels représentations spécifiques. Les langages dédiés sont des langages spécialisés à un domaine d'application en particulier, en opposition avec les langages généralistes, qui ont pour but objectif d'être universels. Dans ce contexte, les langages dédiés sont alors utilisés comme moyen privilégié d'interagir avec les modèles.

Toutes fois, les langages sont des logiciels à part entière [57], ce qui implique l'application des pratiques de l'ingénierie logicielle à leur développement. Et ce à toutes les étapes de leur cycle de vie : définition des exigences, conception, développement, test, déploiement, maintenance, etc. Dans ce cadre, la discipline de l'ingénierie des langages est une sous discipline de l'ingénierie informatique qui se concentre sur la construction rationnelle et scientifique de langages. Cela à travers l'intégration de multiple disciplines tels que l'analyse de programmes, la construction de compilateurs, la transformation de logiciels ou la rétro-ingénierie.

Énoncé du problème

Les langages dédiés peuvent prendre différentes formes. On distingue deux grandes catégories de langages dédiés, les langages dédiés internes et externes. Les langages dédiés internes sont intégrés dans la syntaxe d'un langage existant (par exemple sous la forme de méthodes chainées, ou par staging [81]) et sont donc soumis à la sémantique et aux outils du langage dans lequel ils s'intègrent. Les langages dédiés externes ont quant à

eux une syntaxe et une sémantique et des outils périphériques qui leur sont propres. Dans cette thèse nous nous intéressons plus particulièrement aux langages dédiés externes car, contrairement aux langages dédiés internes, ils ne sont pas contraints par le fonctionnement d'un langage préexistant et offrent donc plus d'opportunité de personnalisation. En contrepartie, la création de langages dédiés externes requière la définition de toutes les composantes d'un langage à partir de zéro. Par conséquent la création de langages dédiés externes requière des efforts de développement beaucoup élevés et des compétences spécialisés en ingénierie des langages. De plus, les langages dédiés ont, par définition, un base d'utilisateurs restreinte car très spécialisée, en particulier en comparaison avec des langages généralistes. La combinaison de ces facteurs limite l'applicabilité des langages dédiés externes.

Les langages dédiés externes sont créés à l'aide d'environnements de développement intégré appelé language workbenches[1]. Les language workbenches assistent le développement de langage en proposant la manipulation d'abstraction pertinentes, associées avec des approches génériques ou génératives. Ces abstractions sont manipulées par les ingénieurs de langage sous la forme de métalangages, c'est-à-dire des langages spécialisés dans la définition de certains aspects des langages. Loin de seulement permettre la spécification de langages à l'aide d'abstraction dédiées, les métalangages sont aussi conçus pour répondre aux challenges inhérents à l'ingénierie des langages. Cela comprend les challenges de la réutilisation, de la modularité ou des performances, parmi tant d'autres. Enfin, les language workbenches assistent la traduction des spécifications de langages à l'aide de métalangages opérant à un haut niveau d'abstraction, vers des implémentations de services permettant la manipulation des langages par des développeurs (par exemple des interpréteurs, des éditeurs ou encore des débogueurs). Toutes fois, les abstractions dédiées aux langages qui sont au cœur des métalangages ne sont généralement pas exploitées au mieux lors de leur traduction vers les implémentations de services de langages. Par exemple, un langage spécifié de manière modulaire peut souffrir d'une absence de support de la compilation séparée, entrainant une absence de la modularité au niveau de son implémentation.

Cela force les ingénieurs de langages à être confrontés aux subtilités de bas niveau des implémentations de services de langage. Les conséquences sont soit 1) une augmentation des coûts de développement (par exemple la réécriture manuelle de fragments de code ayant des performances insuffisantes) 2) une impossibilité de réaliser certains scénarios de développement (par exemple la réutilisation de code ancien).

Question de Recherche

Quels sont les meilleures manières d'exploiter l'information disponible dans les spécifications de langages dans le but d'améliorer les propriétés non-fonctionnelles des implémentations de leurs services ?

[1] Language workbench : Banc de création de langage.

L'information inexploitée disponible au sein des spécifications de langages peut permettre l'amélioration de plusieurs aspects non-fonctionnels des implémentations de services de langages : leur réutilisabilité et leurs performances. Par conséquent, nous tirons les deux challenges ci-dessous de notre question de recherche :

> **Challenge #1 :** Quels sont les patrons d'implémentation de services de langage qui permettent l'amélioration de leur réutilisabilité ? Quelle part de l'information disponible dans les spécifications de langages peut être exploitée pour la traduction des spécifications vers de tels patrons d'implémentation ?

> **Challenge #2 :** Quels sont les patrons d'implémentation de services de langage qui permettent l'amélioration de leurs performances ? Quelle part de l'information disponible dans les spécifications de langages peut être exploitée pour la traduction des spécifications vers de tels patrons d'implémentation ?

Contributions

Nous adressons les deux challenges identifiés ci-dessous à travers deux contributions. Pour répondre au **challenge #1**, il est important d'équiper les ingénieurs de langages avec des solutions autorisant la réutilisation opportuniste (par exemple par extension ou composition) de langages dédiés existant afin d'en définir de nouveaux. Les approches existantes de réutilisation de langages requièrent de l'anticipation, demandent l'usage de fonctionnalités avancée rarement disponible dans les langages populaires, ou ne sont pas applicable dans le contexte de l'ingénierie des langages dédiés basés sur des modèles. Pour répondre à la question de la réutilisabilité dans ce contexte, nous proposons le patron d'implémentation REVISITOR en tant que réponse à la question de la réutilisabilité des langages. Le REVISITOR autorise l'extensibilité à la fois syntaxique et sémantique des langages de langages dédiés basés sur des modèles, supporte la compilation incrémentale et ne requière pas d'anticipation.

Les approches récentes dans ce domaine sont nombreuses mais souffrent habituellement de deux problèmes principaux : soit elles ne supportent par la composition modulaire de langages au niveau de leurs spécifications et de leurs implémentations, soit elles demandent des connaissant pointues sur ds paradigmes de développement avancés. Ces problèmes limitent leur adoption généralisée dans l'industrie. Non introduisons donc une approche non intrusive de développement modulaire de langages, proposant la définition de modules de langages avec une interface explicite qui peuvent être composés modulairement, à la fois au niveau de leurs spécifications et de leurs implémentations.

Nous évaluons dans un premier temps le patron d'implémentation REVISITOR dans le cadre de l'extension de langages [104]. Ensuite, nous évaluons le REVISITOR dans le cadre de la composition de langages [103].

Ensuite, pour répondre au challenge #2, nous proposons une approche pour optimiser automatiquement les performances à l'exécution des interpréteurs de langages dédiés. Dans de nombreux domaines les performances sont essentielles (par exemple pour le calcul scientifique ou à hautes performances) et l'adoption des langages dédiés est freinée par leur performance parfois insuffisante. Cela force les ingénieurs à optimiser manuellement et intrusivement les implémentations de langages dérivées des spécifications. Dans cette seconde contribution, nous proposons l'exploitation systématique des informations fournies par les abstractions disponibles dans les métalangages utilisés pour la spécification des langages afin d'améliorer automatiquement leurs performances. Nous réalisons notre approche par-dessus un cadriciel de modélisation industriel appelé Eclipse Modeling Framework (EMF) [150]. Pour cela nous complétons la chaine de compilation d'EMF avec des optimisations spécifiques aux langages dédiés. Un des points importants de notre approche est qu'elle ne demande pas aux développeurs de changer leurs habitudes de développement ou d'assimiler de nouveaux concepts. Notre approche a donc le double bénéfice d'isoler les développeurs des détails de bas niveau nécessaires à sa réalisation et d'être directement applicable sur du code existant, sans modification préalable.

L'outillage qui entoure la réalisation d'outils qui répondent aux deux challenges développés dans cette thèse sont réalisé à l'aide du cadriciel EMF. Nous réutilisons le métalangage Ecore pour la définition de la syntaxe des langages et un langage d'action, appelé ALE pour la définition de la sémantique des langages. Nous utilisons ces deux métalangages pour la définition des langages utilisés pour l'évaluation de nos approches.

Applications : Implémentations et Évaluations

Afin de valider et d'évaluer nos contributions, nous utilisons Ecore et ALE pour le développement de plusieurs langages. Premièrement, afin d'évaluer la capacité de nos outils à autoriser l'extension de langages, nous avons réimplémenté le langage de diagrammes d'activité initialement proposé dans le cadre du TTC'15[2]. Initialement monolithique, nous validons notre approche par la modularisation par extension de ce langage.

De plus, nous évaluons l'impact du patron d'implémentation REVISITOR en comparant notre implémentation basée sur le patron d'implémentation REVISITOR aux performances de plusieurs implémentations du langage de diagrammes d'activité, chacune basée sur un patron d'implémentation issu de l'état de l'art. Cette évaluation nous permet de conclure que l'impact du patron d'implémentation REVISITOR sur les performances est modéré tout en proposant une extensibilité accrue.

Ensuite, dans le but d'évaluer les capacités de réutilisation par composition de notre approche, nous avons réimplémenté un langage dédié à la définition de systèmes de l'in-

[2]Tool Transformation Contest 2015 [111]

ternet des objets, initialement proposé par Degueule et al. [41]. L'implémentation initiale, bien que modulaire dans ses spécifications, ne supporte pas la compilation indépendante de son implémentation. À l'inverse, notre implémentation suivant notre approche supporte la compilation indépendante des implémentations de modules de langage.

Enfin, dans le but de valider notre approche d'optimisation des performances de langages, nous avons implémenté quatre langages : un sous-ensemble de Java appelé MiniJava [136], a un petit langage fonctionnel nommé Boa et inspiré d'OCaml, un langage de définition de systèmes de machines à états finis et une copie du langage éducatif Logo. Ensuite, nous avons évalué le gain de performance induit par notre approche en mesurant les performances de nos implémentations en comparaison de techniques d'implémentations issues de l'état de l'art.

Ces mesures de performance nous permettent de conclure que notre approche permet l'obtention d'un gain de performance intéressant sans aucun effort de développement et sans demander de connaissances supplémentaires aux ingénieurs de langages. Additionnellement, notre approche est aussi applicable de manière non intrusive sur des langages existants.

CHAPTER 1

INTRODUCTION

In this chapter, we first place this thesis in its research context (Section 1.1) and detail the challenges we address (Section 1.2). Then, we present the scientific contributions (Section 1.3), and the applications we built to validate them (Section 1.4). Finally, we present the organization and the reading flows (Section 1.5) of this manuscript. A list of publications produced in the context of this thesis is also available (Section 1.6).

1.1 Research Context

An important challenge in software engineering is the development and maintenance of complex systems. They involve many components that interact with each other, leading to dependencies and interactions that lead to behaviors that are difficult to reason about. Recent examples of complex systems are the Internet of Things or cyber-physical systems. The design of such complex systems exceeds the cognitive and intellectual capacities of a single actor. Consequently, their design must involve many stakeholders with diverse and heterogeneous points of view, to address the multiple components and aspects of such complex systems. The multiplication of the number of actors, views, and components needed to define complex systems leads to many engineering challenges. Therefore, the success of such engineering work requires the application of dedicated methodologies and tools.

One solution dedicated to the challenge of developing complex systems is Model-Driven Engineering (MDE) [143]. More precisely, MDE promotes the mitigation of the accidental complexity inherent to the development of complex systems by bridging the gap between the problems and solutions spaces through the use of models. A model is a representation of a particular aspect of a system and provides relevant abstractions to the domain experts. Even if models themselves are specific views of the system, their representations are usually generic by default. However, multiple studies show that models are better expressed using domain-specific representations [75, 117]. Such representations are usually realized using Domain-Specific Languages (DSLs). That is to say, software languages dedicated to a particular application domain. In this context, the dedicated means of interaction with the models is then DSLs.

However, since "Software languages are software too" [57], this implies the application of Software Engineering (SE) practices to the development of software languages, at every step of software languages lifecycle: requirements, design, testing, deployment, evolution, maintenance. In this context, the field of Software Language Engineering (SLE) is the sub-discipline of SE that focuses on the rational and scientific construction of software

languages — including DSLs — through the integration of various disciplines such as program analysis, compiler construction, software transformation, and reverse engineering.

1.2 Problem Statement

Because of the diversity of the contexts in which DSLs are used, they come in many shapes and forms. We distinguish two main categories of DSLs, Internal, and External DSLs. Internal DSLs[1] are embedded in the syntax of existing languages (e.g., fluent APIs, staging [81]). Consequently, their embedding in an existing language makes them dependent on the syntax, the semantics, and the tooling of this language. Conversely, External DSLs provide their own syntax and semantics. This absence of dependency on an existing language's syntax and semantics requires the definition of dedicated tool support to accompany the use of External DSLs.

In this thesis, we focus on the engineering of External DSL because, unlike Internal DSLs, they are not constrained by the characteristics of a pre-existing language. Therefore, External DSLs offer more opportunities for customization, which is necessary to meet the diversity of uses that exist in the context of complex systems.

In counterpart, the creation of External DSLs requires to fully define the syntax, the semantics, and the tooling. This task is extremely costly and requires specific engineering knowledge that is not very widespread. Additionally, by definition, a DSL has a narrow user-base of experts of the application domain targeted by the DSL, this is especially true in comparison to popular General-Purpose Languages (GPLs). The combination of these factors limits the applicability of External DSLs, and dedicated solutions must be developed to mitigate such limitations.

External DSLs are created using specialized Integrated Development Environments (IDEs), called language workbenches. Language workbenches assist in the development of DSLs by offering useful language abstractions associated with generic or generative approaches. Language engineers manipulate these abstractions through metalanguages; in other words, languages specialized for the definition of specific aspects of languages. Far from solely allowing the *specification* of languages using dedicated abstractions, metalanguages are also designed to address the challenges inherent to the engineering of languages. This includes the challenges of reuse, modularity, or performances, to name a few. Finally, language workbenches assist in the transformation of language specifications to implementations of services supporting the use of software languages (e.g., interpreters, editors, debuggers).

However, the language-specific abstractions at the core of metalanguages are often not exploited at their best for the implementation of the tool support essential to the use of software languages by the language engineers. For instance, independently defined language modules can suffer from the absence of a separate compilation of their implementations,

[1] Internal DSLs are also sometimes called Embedded, but the term is ambiguous [114] and will not be used in this document

leading to the loss of modularity of the language services implementation.

This forces language engineers to be confronted to the low-level intricacies of language service implementations, which are either 1) costly to address in term of development time (e.g., the manual definition of glue between modules, or the re-implementation of slow language implementations) 2) preventing the realization of useful engineering scenarios (e.g., the reuse of legacy artifacts).

Research Question

How to best exploit the information available in the metalanguage abstractions to improve the non-functional properties of software language services implementation?

The unexploited information available in language specification can be employed to improve the non-functional properties of software language implementations. In this thesis, we focus on two non-functional properties: reusability and performance. Consequently, we draw the two challenges below from our research question:

Challenge #1 What are the relevant language implementation patterns to improve the reusability of software language implementations? Which information available in language specifications is useful to the translation of software language specification to these patterns?

Challenge #2 What are the relevant language implementation patterns to improve the performances of software language implementations? Which information available in language specifications is useful to the translation of software language specification to these patterns?

1.3 Contributions

We tackle the aforementioned challenges through two contributions. In the context of the reuse of software language implementations (**challenge #1**), it is important to provide language engineering facilities for opportunistic reuse (e.g., extension and customization) of existing DSLs to ease the definition of new ones. Current approaches to language reuse either require anticipating reuse, make use of advanced features that are not widely available in programming languages, or are not directly applicable in the context of model-based DSLs. To do so, we propose a new language implementation pattern, named REVISITOR, that enables the independent extensibility of the syntax and semantics of metamodel-based DSLs with incremental compilation and without anticipation.

Recent approaches in this area are plentiful but usually suffer from two main problems: either they do not support modular language composition both at the level of language specifications and the level of language service implementations, or they require advanced knowledge of specific paradigms which hampers wide adoption in the industry. We introduce an approach to the modular development of language modules with well-defined interfaces that can be composed modularly at the level of language specifications and the level of language services implementation. We first demonstrate the REVISITOR implementation pattern in the context of software language extension [104]. Then, we demonstrate the REVISITOR implementation pattern in the context of software language composition [103].

Then, to address the challenge of the performances of software language implementations (**challenge #2**), we propose an approach to optimize the runtime performances of DSL interpreter implementations automatically. In many domains where performance is key (e.g., scientific and high-performance computing), the implementations of DSLs interpreters suffer from performance issues. These forces language engineers to handcraft ad-hoc optimizations in the generated interpreters code. In this contribution, we propose to systematically exploit the information available in the language specifications to derive optimized language interpreters. We implement our approach on top of an industrial standard (i.e., Eclipse Modeling Framework (EMF) [150]) by complementing its existing compilation chain with specific optimizations. A key benefit of our approach is that it leverages existing language specifications and does not require additional development effort from language engineers who remain oblivious of low-level intricacies while preserving the tool support accompanying External DSLs.

Our approaches have been implemented on top of the EMF ecosystem. We reuse the Ecore metalanguage to define language's abstract syntaxes and build two-compilers for ALE, a metalanguage dedicated to the definition of operational semantics on top of Ecore metamodels.

1.4 Implementations and Evaluations

To evaluate and validate our contributions, we use Ecore and ALE for the development of several languages. First, to evaluate the aptitude of our tools to allow the extensions of languages, we reimplemented the fUML language initially proposed in the context of the Tool Transformation Contest of 2015.[2] Initially monolithic, we validated our approach by the modularization of both its specification and implementation.

Then, to validate the composition capabilities of our approach, we reimplemented an existing DSL for the Internet of Things, initially proposed by Degueule et al. [41]. The initial implementation is modular at the specification level but does not support the separate compilation of its language services implementation. Instead, we propose an approach that, although also modular at the specification level, supports the separate compilation of

[2]Tool Transformation Contest, 2015: http://www.transformation-tool-contest.eu/2015/

the language services implementation.

Finally, to validate our contribution to the automatic optimization of the performance of software language implementations (**challenge #2**), we implemented four languages: a subset of Java [136], a small functional language inspired by OCaml called Boa, a system of Finite State Machine (FSM) language and the educational Logo language. Then, we evaluate the performance gain allowed by our approach in comparison to standard language implementation approaches.

These experimentations allow us to conclude that our approach provides interesting performance speedups without additional development effort and without requiring additional knowledge from language engineers. Besides, our approach is also seamlessly applicable to existing EMF-based languages.

1.5 Outline

The content of this thesis is structured into four parts. A graphical view of the structure of this document is presented in Figure 1.1.

The first part (Part I) is composed of two chapters and introduces the preliminaries that lead to this thesis' contributions. Chapter 2 introduces the general background required for the understanding of the remainder of this thesis. This chapter presents current practices in SLE, with a focus on the engineering of DSLs in an MDE context. Chapter 3 reviews the state of the art of the two aspects of DSLs addressed in this thesis: reuse and runtime performances.

The second part (Part II) is composed of two chapters and presents our contributions. The first chapter addresses the reusability (**challenge #1**), and the second chapter addresses runtime performances (**challenge #2**). Chapter 4 presents our work on the safe and modular reuse of software language modules. Finally, Chapter 5 presents our work on optimizing compilers for DSLs interpreter implementations.

Alternative reading orders

While this document is written to be read linearly, we propose alternative reading orders for the various interested readers. Readers wishing to improve their understanding of the subject of MDE and SLE can read Chapters 2 and 3 independently of the remainder of the rest of the document, as a reference of the global context of the model-based specification of DSLs, and the state of the art of DSLs reuse and runtime performances.
Additionally, readers interested solely in language reuse can skip Chapter 5 and Chapter 8, whereas readers interested only in language runtime performance can skip Chapter 4 and Chapter 7.

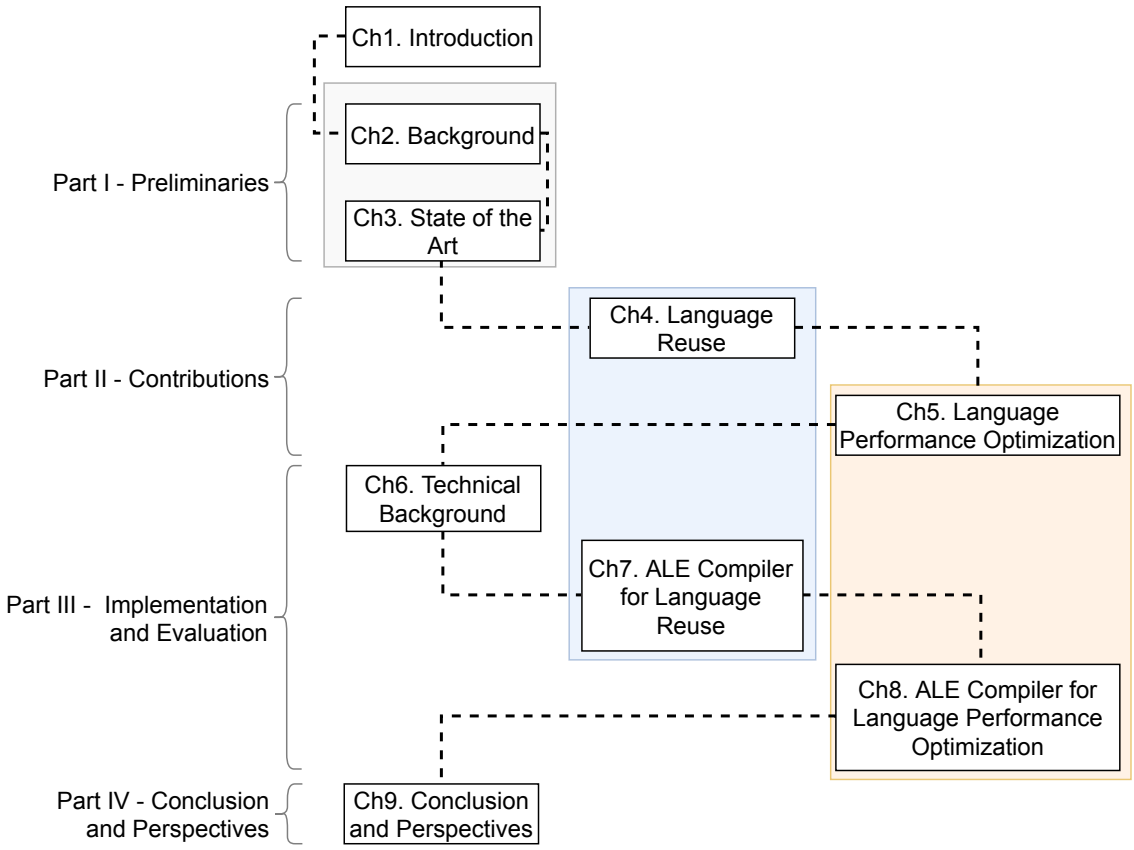


Figure 1.1 – Outline of this thesis. Chapters are expected to be read from top to bottom, following the dashed lines.

The third part (Part III) contains three chapters and details the implementation and evaluation of our contributions. First, Chapter 6 presents the technical background required for the understanding of the remainder of the chapters of this part. Then, Chapter 7 presents the implementation and evaluation of our approach to language reuse (**challenge #1**). Finally, Chapter 8 presents the implementation and evaluation of our approach to the automatic optimization of the runtime performance of DSLs (**challenge #2**).

The last part of this thesis (Part IV) contains a single chapter (Chapter 9) that summarizes our contributions and presents the line of works resulting from the contributions of this thesis.

1.6 List of Publications

The list of publications achieved during the development of this thesis is presented below and separated between the accepted publications strictly related to this thesis (Section 1.6.1), the publications currently under minor revision (Section 1.6.2), and other publications (Section 1.6.3).

1.6.1 Main Publications

The Software Language Extension Problem (Experts voice) Manuel Leduc, Thomas Degueule, Eric Van Wyk, Benoît Combemale. In International Journal on Software and Systems Modeling (SoSyM), 2020 [105].

Modular Language Composition for the Masses (Conference paper — *distinguished artifact award*) Manuel Leduc, Thomas Degueule, Benoît Combemale. In Proceedings of the 11th International Conference on Software Language Engineering (SLE), 2018 [103].

Modular Language Composition for the Masses (Poster) Manuel Leduc, Thomas Degueule, Benoît Combemale. Presented during the 11th International Conference on Software Language Engineering (SLE), 2018.

Revisiting Visitors for Modular Extension of Executable DSMLs (Conference Paper) Manuel Leduc, Thomas Degueule, Benoît Combemale, Tijs van der Storm, Olivier Barais. In Proceedings of the 20th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS), 2017 [104].

1.6.2 Publications Under Minor Revision

Automatic generation of Truffle-based interpreters for Domain-Specific Languages (Conference Paper) Manuel Leduc, Gwendal Jouneaux, Thomas Degueule, Gurvan Le

Guernic, Olivier Barais, Benoît Combemale. Submitted to the 16th European Conference on Modelling Foundations and Applications (ECMFA), 2020.

1.6.3 Other Publications

Concern-Oriented Language Development (COLD): Fostering Reuse in Language Engineering (Journal Paper) Benoît Combemale, Andreas Wortmann, Erwan Bousse, Gunter Mussbacher, Jean-Marc Jezequel, Jörg Kienzle, Manuel Leduc, Matthias Schöttle, Misha Strittmatter, Olivier Barais, Philippe Collet, Robert Heinrich, Sébastien Mosser, Tanja Mayerhofer, Thomas Degueule, Walter Cazzola. In Computer Languages, Systems and Structures (COMLAN), 2018 [35].

SLEBOK: The Software Language Engineering Body of Knowledge (Dagstuhl Seminar 17342) (Report) Benoît Combemale, Ralf Lämmel, and Eric Van Wyk. Edited by Manuel Leduc. In Dagstuhl Reports 7.8, 2018 [33].

KevoreeJS: Enabling Dynamic Software Reconfigurations in the Browser (Conference Paper) Maxime Tricoire, Olivier Barais, Manuel Leduc, François Fouquet, Gerson Sunyé, Brice Morin, Johann Bourcier, Grégory Nain, Ludovic Mouline. In Proceeding of the 19th International ACM Sigsoft Symposium on Component-Based Software Engineering (CBSE), 2016 [156].

PART I

Preliminaries

CHAPTER 2

BACKGROUND

In this chapter, we introduce the theoretical and technical background used in this thesis. The presentation of these preliminary concepts eases the understanding of the State of the Art (Chapter 3). First, we introduce Model-Driven Engineering (Section 2.1). Then, we present an overview of the notions related to Software Languages (Section 2.2). Finally, we focus on Software Language Engineering (Section 2.3) and present its application in the context of Model-Driven Engineering.

2.1 Model-Driven Engineering

Model-Driven Engineering (MDE) is a development paradigm that aims at mitigating the accidental complexity inherent to the development of complex systems through the use of models. While the term is overloaded and used with slightly different meanings in many scientific and engineering communities, we base our work on the following definition, quoting France et al. [62]:

> "A model is an abstraction of some aspect of a system. The system described by a model may or may not exist at the time the model is created. Models are created to serve particular purposes, for example, to present a human-understandable description of some aspect of a system or to present information in a form that can be mechanically analyzed."

MDE aim at taming the development of complex systems in two ways [34]:

- **Separation of concerns:** Multiple models are used to describe a system, each describing a specific aspect of the system. Hence, the system's aspects are defined by relevant abstractions, notations, and modeling tools. Consequently, domain experts can focus on independent aspects of complex systems (e.g., security, environmental impact).
- **Raising the level of abstraction:** MDE helps to bridge the gap between high-level and low-level concepts. To do so, MDE promotes the automatic code generation from models to software artifacts (e.g., documentation, tests, components).

The MDE methodology reasons at four levels (cf. Figure 2.1). The description given in the Engineering Modeling Languages book [34] inspires the following description of the four levels, initially drawn from the Model-Driven Architecture (MDA) approach [56]. The real world (i.e., the modeled system) is presented at the right (M0 level). Models representing this system correspond to the M1 level. Metamodels for the definition of these

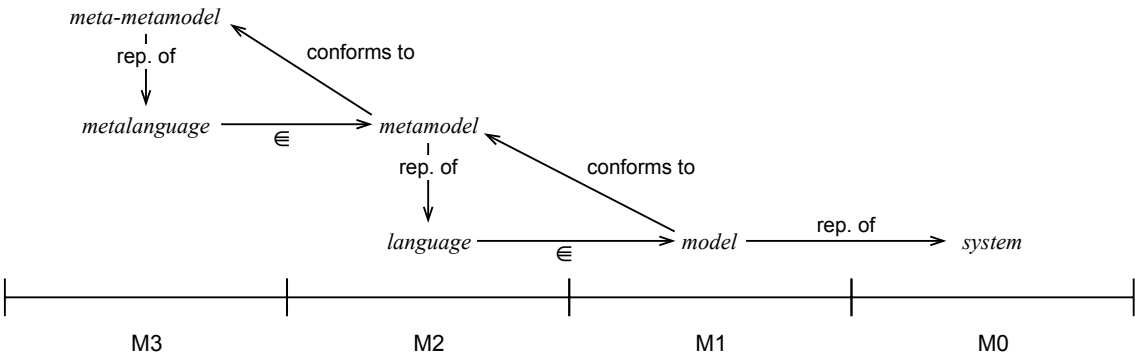


Figure 2.1 – Summary of MDE levels and their relations to models and languages [78].

models (e.g., the UML metamodel) correspond to the M2 level. Finally, meta-metamodels are at the left (M3 level). Each level corresponds to a particular use/purpose of models (M1 for modeling systems, M2 for modeling languages, M3 for modeling metalanguages).

It should be noted that such a hierarchy arrangement of models according to their particular purposes is not specific to MDA and has been used in other areas of computer science. For instance, we see such hierarchy in other *technological spaces* [96] such as grammarware in the context of grammars.

Metamodels Metamodels are at the core of MDE [5] and materialize the knowledge of an application domain. Metamodels conform to meta-metamodels, and a meta-metamodel is a representation of a metalanguage. Metalanguages are used by language engineers for the specification of languages.

Language Engineer

A language engineer exploit the metalanguages and their corresponding language workbenches (M3) to specify languages and build their corresponding tooling (M2).

Many metamodeling formalism exist, such as entity-relationship metalanguages (e.g., AToM^3 [101]) or object-oriented metalanguages (e.g., Meta-Object Facility (MOF) [127]). MOF is the current industry standard for the definition of models and is used for the remainder of this thesis. EMF [150] is a framework dedicated to the definition and manipulation of models and is aligned with MOF. The MOF formalism freely inspires the illustrative examples that follow.

Figure 2.2 presents an excerpt of a self-descriptive meta-metamodel (i.e., sufficient to model itself). Using this meta-metamodel, we are also going to define the metamodel of Petri nets (Figure 2.3). The concept of `Package` represents the containment of the classes common to a metamodel. The concept of `Class` is qualified by a `name`. A `Class`

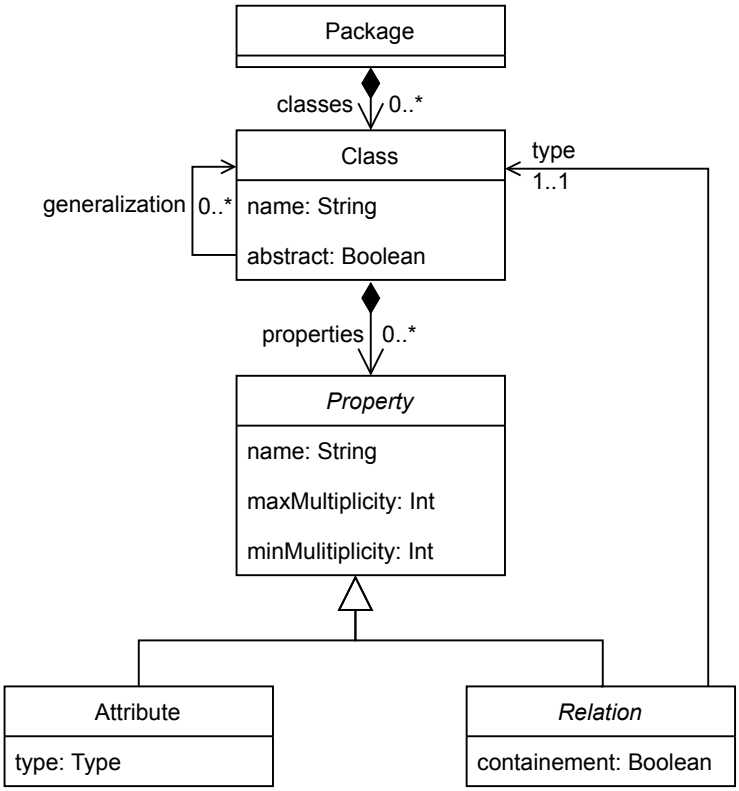


Figure 2.2 – Extract of self-descriptive meta-metamodel.

can eventually be abstract, can have a list of classes from which it generalizes and a list of `properties` containing classes of type `Property`. `Property` is an abstract class qualified by a `name`, minimal and maximal multiplicities (i.e., `minMultiplicity` and `maxMultiplicity`). Two classes inherit (i.e., generalize) from `Property`: `Attribute` and `Relation`. The concept of `Attribute` is qualified by all the properties of `Property` by inheritance and by a `type` attribute (e.g., String, Boolean, Integer). In the same way, the concept of `Relation` is qualified by all the properties of `Property`, can eventually be a `containment`, and has a relation named `type` that holds a reference to `Class`. The notion of containment is conceptually equivalent to UML's composition relations [1].

Figure 2.3 illustrates the use of a meta-metamodel to define a metamodel. Our metamodel example is based on Petri nets, a mathematical modeling language often used for the description of distributed systems. A Petri net is composed of nodes and arcs. A node can be either a transition or a place. This is modeled by the inheritance of the Transition and Place classes to the abstract Node class. An oriented arc connects an incoming node to an outcoming node (i.e., *from* and *to* relations). Finally, an arc is qualified by its weight of type integer. The graphical representation of the metamodel follows UML's class diagram notation.

In addition to the conformance relation between a model and a metamodel, complementary well-formedness rules are sometimes required. These rules express static constraints that cannot be expressed using metamodeling formalisms. In the context of MOF, well-formedness rules are defined using Object Constraint Language (OCL) [126]. As an example, in a Petri net an arc can only connect a transition to a place, or a place to a transition, but should not connect two nodes of the same type (e.g., a transition to another transition). One can observe that the Petri net metamodel presented here does not enforce such constraint. In this case, an OCL rule can allow the static validation of the well-formedness of Petri net models. An example of such OCL rule is presented in Listing 2.1 and validates if the two nodes of an arc are of two different types. Similarly, the weight of the arcs of a Petri net must be a strictly positive integer. Another OCL rule can be used to validate this constraint.

Models Models conform to metamodels, and metamodels are representations of languages. Languages allow the specification of models by domain experts. One can observe that the same set of relationships is repeated between levels M3 and M2 as well as between levels M2 and M1.

Figure 2.4 presents an object diagram of an illustrative example of a Petri net that can

```
1 context Arc
2 inv arcsTypes: (self.from.oclIsKindOf(Place) and
      self.to.oclIsKindOf(Transition)) or (self.from.oclIsKindOf(Transition) and
      self.to.oclIsKindOf(Place))
```

Listing 2.1 – Example of well-formedness rule for Petri net

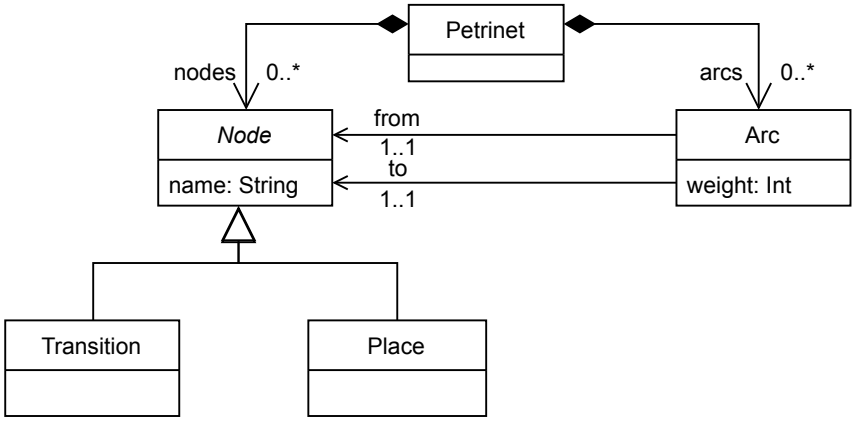


Figure 2.3 – Illustrative Petri net metamodel conforming to the meta-metamodel of Figure 2.2.

be modeled from the Petri net metamodel of Figure 2.3. It is composed of two transitions – T1 and T2 – and four places – from P1 to P4. Six arcs connect the places and transitions. P1 is connected to T1, and T1 is connected to P2 and P3. P2 and P3 are connected to T2. Finally, T2 is connected to P4. This representation of a Petri net, while accurate and valid, is somewhat technical and does not represent the usual way of representing Petri nets. Indeed, they are usually represented graphically using circles for the places, rectangles for the transitions, and arrows between nodes for the arcs. Consequently, it is not straightforward to reason on the modeled Petri net from its generic representation. We will show in Section 2.3.1.b how languages can be exploited to provide domain-specific representations of models.

Domain Expert

A domain expert exploit the languages and their corresponding tooling (M2) to define models (M1) of the systems of interest (M0).

In this thesis, we focus on the specification of metamodels using metalanguages, in the objective of defining languages. In the next sections, we present the notion of software languages (Section 2.2) and their engineering (Section 2.3).

2.2 Software Languages

Software languages are artificial languages that can be defined with software [98]. They hold an important role in computer science and SE, highlighted by the increasing number of newly created software languages [97]. Software languages can be classified according

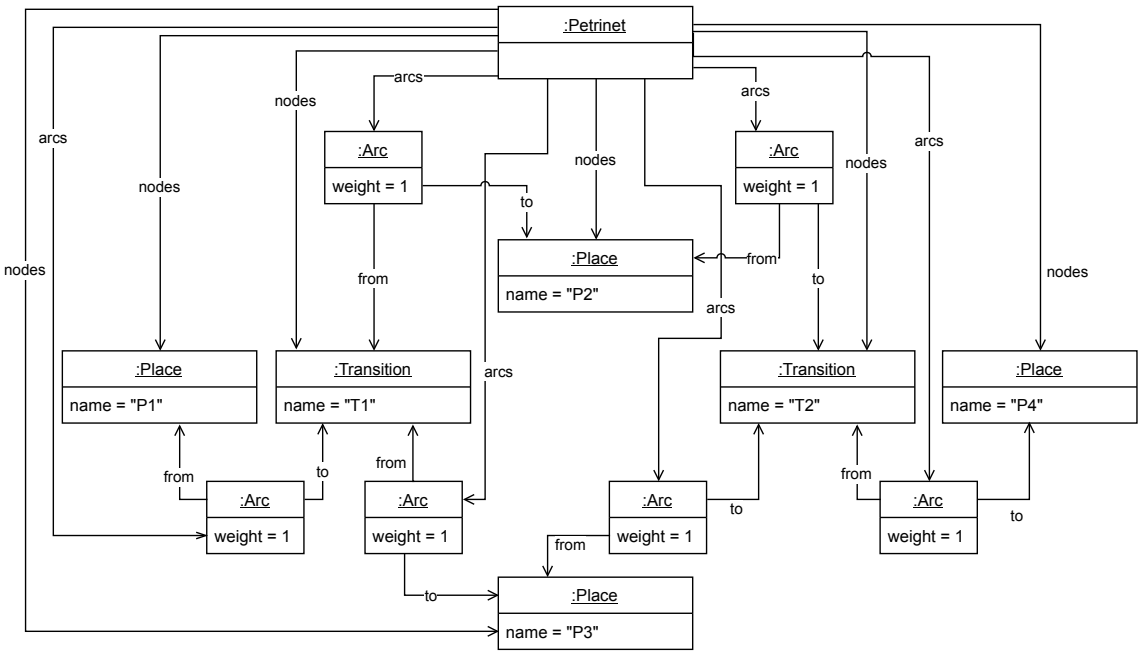


Figure 2.4 – A model conforming to the Petri net metamodel of Figure 2.3.

to the intended application domain of the language. We present a non-exhaustive but illustrative selection of application domains and associated software languages:

- configuration formats (e.g., json, and yaml).
- markup (e.g., html, and markdown).
- data query (e.g., SQL).
- security (e.g., ATSyRa [133]).

Another orthogonal axis of classification for software languages is the distinction between General-Purpose Languages (GPLs) and Domain-Specific Languages (DSLs) [44]. GPLs are not attached to solve the problems of a specific application domain and intend to be universal (e.g., Java, UML), whereas DSLs are specialized to solve the problems of a well-identified application domain (e.g., SQL, awk). In practice, the border between DSL and GPL is blurry, and some software languages initially designed to be domain-specific are now considered as general-purpose (e.g., Fortran or Lisp) [44].

The main benefit of DSLs is their ability to allow the expression of problems at the level of abstraction of the problem domain [43, 44]. Additionally, DSLs are a way to reify the knowledge of an application domain on software and to make this knowledge explicitly available to users. Similarly, the expression of problems at the application domain level avoids the loss of knowledge when applied to lower levels of abstraction. Additionally, reasoning at the level of abstraction of the problem simplifies the development of peripheral tools such as tests, validators, model checkers, or optimizers. For instance, it is easier to identify an unreachable state in a state machine program defined using a dedicated formalism compared to the same analysis applied on a similar program defined

using a GPL. Besides, there is some evidence that DSLs allow their users to be more productive [92], but it is fair to observe the lack of a strong consensus on the topic [8].

However, DSLs are not as widely adopted as it could be expected from its promises. In practice, DSLs are costly to develop and maintain. Indeed, DSL development is a complex task that requires specialized knowledge and tools. Also, linking concepts between languages is notoriously challenging [28], and the accumulation of small languages caused by the adoption of DSLs can lead to compatibility and integration issues that increase the cost of DSLs.

From a design point of view, languages are composed of two fundamental concerns: the syntax and the semantics. These terms are inspired by the natural language vocabulary and hold similar meanings.[1]

The syntax governs the form and notation of a language, and the semantics defines the meaning associated with its syntax. In the context of software languages, we distinguish between the concrete syntax and the abstract syntax. The concrete syntax can be textual or graphical, and represent the visual notation manipulated and observed by the language's users. The abstract syntax is the minimal representation of the syntax of a software language, containing only purely structural information, and striped from concepts needed only for the language's visual representation. Figure 2.5 presents those concerns and their relations, annotated with the common approaches used for the specifications, described in Section 2.3.1. The core of a language is conceptually constituted of an abstract syntax, complemented with concrete syntaxes and semantics. The concrete syntaxes and semantics are decoupled and can evolve independently. Degueule et al. relaxed the constraints of the abstract syntax and propose an approach where semantics can be modularly applied on multiple abstract syntaxes [40, 41].

It should be noted that the conceptual relation between language concerns does not constrain the order of definition of their specifications, that often co-evolve in practice. The two main language technological spaces are named grammar-first and model-first, where respectively a textual concrete syntax is first defined and an abstract syntax is — eventually implicitly — derived from the former, or the abstract syntax is defined first in the form of a metamodel, and a concrete syntax is specified in a second step.

2.3 Software Language Engineering

Software languages face the same development challenges as other software (e.g., maintenance, test, evolution). Addressing these challenges implies the application of Software Engineering (SE) practices to the development of software languages at every step of the software language lifecycle: requirements, design, testing, deployment, evolution, maintenance. Software Language Engineering (SLE) is a sub-discipline of SE dedicated to the engineering of languages. SLE abstract away from specific kinds of languages, as presented in Section 2.2, but focuses on the engineering aspect of language construction.

[1] We acknowledge that we exclude the pragmatics from this analogy.

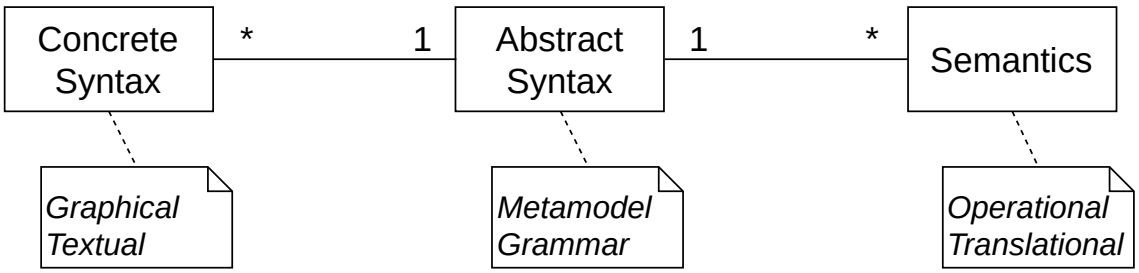


Figure 2.5 – Relationship and cardinality of software language concerns.

This involves the application of rational and scientific methods to produce the best engineering results. Concretely, SLE applies the methods of various disciplines such as program analysis, compiler construction, software transformation, and reverse-engineering.

In the context of SLE, languages are developed using specialized languages, dedicated to the specification of the various aspects of languages. Language specifications are then translated to language services, for instance, editors, interpreters, or validators. Languages used for the specification of languages are called metalanguages and often take the form of DSLs applied to the specification of specific aspects of languages. For instance, the grammar of a textual language can be specified using metalanguages with formalism close to the Extended Backus-Naur Form ((E)BNF) notation [175] (e.g., Xtext [55], ANTLR [129] or SDF [69]). The translation of language specification to language services can follow different patterns, usually by compilation to a GPLs, or by interpretation (e.g., Spoofax [85], Rascal [89]).

A large part of the SLE research effort focuses on the study of the relevant metalanguages and their transformation into language services implementation. In the following section, we present the concepts used for the specification of languages (Section 2.3.1), the patterns used for the implementation of language services (Section 2.3.2), and the language workbenches that support DSLs development (Section 2.3.3).

2.3.1 Domain-Specific Language Specification

DSL specifications are a cornerstone of SLE and are used as high-level formalism providing the relevant abstraction to describe languages. From those specifications, a large diversity of language service implementations can be derived. Usually, the main artifact is a language interpreter, complemented with peripheral tools such as language editor services, debuggers, profilers, documentations, or tests.

Dedicated metalanguages exist for every layer of languages, from the concrete syntax and the semantics to more specific layers such as type systems, scopes, or tests. In the next sections we focus on the three central layers of languages: abstract syntax (Section 2.3.1.a), concrete syntax (Section 2.3.1.b), and semantics (Section 2.3.1.c).

Specifications

The term *specification* is greatly overloaded in computer science and often includes *descriptive* specifications, eventually defined using natural languages. In this thesis, we focus on *prescriptive* specifications that can be systematically manipulated by software to produce useful results.

2.3.1.a Abstract Syntax

The abstract syntax of a language is a data structure that represents the logical representation of its concepts and their relations, without reference to its concrete representation (i.e., its concrete syntax, presented in Section 2.3.1.b). For instance `1 + 2 * 3` and `add(1, mult(2,3))` are different notations for equivalent arithmetic expression.

Abstract syntaxes are an essential part of language specifications, allowing the automated processing of language concepts without relying on its concrete syntax, usually presented in the form of an Abstract Syntax Tree (AST).

Abstract syntax formalisms intend to define the set of valid AST instances of the language. The three most widely used formalisms are metamodels, (context-free) grammars, and Abstract Data Types (ADTs). Many works study these formalisms and present the commonalities and conceptual relations between them [3, 12, 95]. The notions of metamodeling are already presented in Section 2.1. Consequently, the remainder of this section focuses on grammars and Abstract Data Types (ADTs).

Grammars are historically used to formally specify the abstract syntax and concrete syntax of textual languages [7], but their use has been extended to other aspects such as editor support (e.g., code outline), parser, or error reporting. More precisely, a grammar is a way to define a data-structure and its notation in a single formalism. The most current example of grammar is (E)BNF-like notations, but data format definition (e.g., XML's DTD or JSON schema) can be seen as a way to define a grammar on top of general-purpose data formats.

ADTs are inspired by the mathematical notion of algebraic structure and are defined by the set of possible values and the set of operations that can be applied to the values. In practice, and ADT has a name, encapsulates its concrete implementation, and provides a set of operations (e.g., create, read, combine) [36].

Figure 2.6 shows two illustrative definitions of the abstract syntaxes using a grammar formalism (Figure 2.6a) and an ADT (Figure 2.6b). These abstract syntax definitions use, respectively Antlr4 [129] and Scala [122]. The grammar formalism defines the abstract syntax and the concrete syntax in the same formalism, whereas an ADT represents purely the concepts of the application domain using classes. The ADT instead allows the explicit definition of the relations between the language's concepts, whereas the grammar formalism defines only identifiers (i.e., ID tokens), and an additional analysis phase is required to proceed to the identification of relations between the language's concepts.

The term grammarware is introduced by Klint et al. [88] to designate the technical space of grammars and grammar-dependent softwares. Similarly, the term modelware is used to designate the technical space of models and model-dependent softwares.

2.3.1.b Concrete Syntax

The concrete syntax of a language represents the interface with which users of the language interact, and may be either textual or graphical. Textual languages are represented as a structured sequence of characters, whereas graphical languages have their own ad-hoc data-structures that are presented to the user through graphical interfaces.

Textual concrete syntaxes are defined in terms of production rules, often using variations of the (E)BNF notation [174], that defines rules that specify the — possibly infinite — set of valid sequences of characters of the language. A parser generator is a tool that generates a parser implementation from language specifications.

Graphical concrete syntaxes are usually represented to the user using geometric shapes (e.g., box and arrows) annotated with text, with which users can interact mainly using the mouse or a touch screen [144]. Moreover, some graphical concrete syntax formalisms promote the use of mathematical or tabular notations [166]. Well-known examples of language with a graphical syntax are UML or Scratch [135].

Figure 2.7 shows two illustrative concrete syntaxes for the Petri net use case presented in Section 2.1. Both are logically equivalent to the object diagram of Figure 2.4, but instead of proposing a generic view, they offer domain-specific representations of the model. On the left (Figure 2.7a), is shown a graphical concrete syntax that matches the usual representation of Petri nets in the literature. On the right (Figure 2.7b), a textual concrete syntax of the same Petri net model is proposed. The domain-specific graphical representation of the Petri net model uses its traditional representation, using circle, rectangle, and arrows. Instead, the textual representation uses common language design idioms, such as a scoping based on nested brackets and keywords to identify concepts (e.g., petrinet, place, transition, arc).

2.3.1.c Semantics

The semantics is the part of the language that gives meaning to its concepts. It is realized by the interpretation or compilation of the abstract syntax to obtain a result. Regardless of the mean of execution of the semantics, the results can either be side effects (e.g., text printed on the standard output) or software artifacts (e.g., documentation, logs or binary executables).

The dynamic semantics defines the runtime behavior of the language and often have a notion of state or evolution over time. There exists three main kinds of dynamic semantics: Axiomatic, Operational, and Translational. They are not exclusive, and a language can have multiple cohabiting dynamic semantics of each kind, for instance, for performance or portability reasons. While they are all presented below, the remaining of this thesis focus on operational semantics.

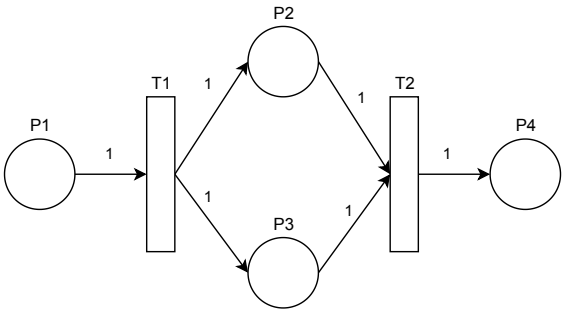
```
grammar Petrinet;

petrinet  : 'petrinet' '{' (node|arc)* '}' ;

node: transition|place;

transition: 'transition' ID;
place: 'place' ID;

arc: 'arc' 'from' ID 'to' ID 'weight' INT;

ID : [a-z]+ ;
INT : [0-9]+ ;
WS : [ \t\r\n]+ → skip ;
```

(a) Illustrative grammar definition for the Petri net abstract syntax of Figure 2.3 using Antlr4.

```
case class Petrinet(nodes: List[Node], arcs: List[Arc])
abstract class Node
case class Transition(name: String) extends Node
case class Place(name: String) extends Node
case class Arc(from: Node, to: Node, weight: Int)
```

(b) Illustrative ADT definition for the Petri net abstract syntax of Figure 2.3 using Scala.

Figure 2.6 – Two illustrative abstract syntaxes using grammars and ADT for the Petri net use case presented in Section 2.1.



(a) Graphical representation of the Petri net instance of Figure 2.4.

```
petrinet {
  place P1
  transition T1
  place P2
  place P3
  transition T2
  place P4

  arc from P1 to T1 weight 1
  arc from T1 to P2 weight 1
  arc from T1 to P3 weight 1
  arc from P2 to T2 weight 1
  arc from P3 to T2 weight 1
  arc from T2 to P4 weight 1
}
```

(b) Textual representation of the Petri net instance of Figure 2.4

Figure 2.7 – Two illustrative concrete syntaxes for the Petri net use case presented in Section 2.1.

Axiomatic Semantics is an approach that relies on mathematical logic, e.g., Hoare logic. An axiomatic semantics defines the behavior of programs by defining assertions that always hold. The intended use of axiomatic semantics is to reason in terms of properties satisfied by a program, for instance, its termination or memory access safety.

Operational Semantics defines how a program is executed by describing transition functions between program states. Operational semantics are divided into two kinds, small-step operational semantics define the execution as a sequence of small state transformations, whereas big-step operational semantics describes an execution as a single big transformation step — from the initial state to the final state — composed of sub-transformation steps.

Translational Semantics defines programs in terms of their outputs, often in the form of a translation from an input language to an output language (e.g., compilers). The term denotational semantics can also be found in the literature whenever the constructs of the translation are mathematical objects.

2.3.2 Domain-Specific Language Services Implementation

DSLs are specified at high levels of abstraction using metalanguages. To be usable by domain experts, DSLs are translated into language services. For instance: interpreters, debuggers, or editors. These implementations rely on regular and well-documented implementation patterns.

Such implementation patterns come with non-functional properties that impact significantly the scenarios offered to language engineers and domain experts, for instance, in terms of modularity, performance, or tool support.

The translation from highly abstract metalanguages to language services implementation is a complex task that involves compilers construction knowledge in addition to the understanding of the expected shape of the implementations. In the context of SLE and MDE, this translation is realized using model transformation techniques.

The patterns involved in the implementation of language aspects are a deeply studied topic that aims at defining reproducible implementation patterns with well-defined and documented properties. Hills et al. propose a comparison of two implementation patterns in terms of maintainability and efficiency [72]. Spinellis [147] propose a classification of language implementation patterns, later extended by Mernik et al. [115].

2.3.3 Language Workbenches

The term *language workbench* was introduced by Martin Fowler in 2005 [61], but the idea is not new and can be traced back to the CENTAUR system [17], in the late eighties. A language workbench is an IDE dedicated to the creation of languages. They usually support a predefined set of metalanguages and provide tools supporting the engineering

tasks involved during the development life-cycle of languages. Examples of recent language workbenches are LISA [116], Melange [41], MontiCore [93], Neverlang [157], Rascal [89], or Spoofax [85], to name a few. Erdweg et al. compared ten language workbenches and their features [51]. Coulon et al. propose an approach to synchronize models or programs between multiple language workbenches, to bridges the gap between language workbenches and benefit from their combined strengths [38].

2.4 Summary

In this chapter, we introduced MDE as a relevant approach for the definition of complex systems. We also presented DSLs as a solution to raise modeling tools to the level of abstraction at which domain experts reason when modeling complex systems. Finally, we show how the development of DSLs benefits from the practice of SLE. In the next chapter, we detail the current state of the art of reuse and performances in the context of the engineering for software languages for MDE.

CHAPTER 3

STATE OF THE ART

In this chapter, we first define a scope for our state of the art (Section 3.1). Then, we explore the state of the art of Software Language Engineering regarding reuse in software language engineering (Section 3.2) and automatic optimization of language runtime performances (Section 3.3) exhaustively. Finally, we explore and compare the contributions to tool support for the development of languages (Section 3.4). To do so, we review the body of work that contributes to those concerns. We close this chapter by presenting the conclusions we can draw from this study (Section 3.5).

In Section 1.2, we identified two challenges drawn from the need for deeper exploitation of information available in the abstractions used for the specification of DSLs. **Challenge #1** aims at improving the reusability of software language implementations, and **challenge #2** aims at improving the runtime performance of DSLs.

In order to support the integration of DSLs in an MDE context, both challenges have the additional requirement to be compatible with the tools support, which is essential to the work of language engineers. Indeed, the work of language engineers is supported by tools and methods that focus on abstractions specific to the definition of languages. However, improving the non-functional properties of language implementation is often realized by introducing new abstractions on the tools that support the definition of languages. We claim that the optimization of the non-functional properties of language implementation can instead be improved by deeper exploitation of the existing abstraction manipulated by language engineers.

The objectives and motivations underlying these challenges are not new, and a tremendous amount of work is already available to support them. Section 3.1 defines further the three concerns that help us scope our state of the art. The first concern relates to language reuse in the context of SLE (Section 3.2). Then, the second concern relates to the automatic optimization of language runtime performances (Section 3.3). Finally, the third concern relates to the tools supporting the development of DSLs (Section 3.4).

3.1 Context and Scope

In this section, we present the concerns that help us scope our state of the art, and that led to the identification of the contributions presented in Part II. Figure 3.1 presents the three concerns we follow to explore this state of the art.

We select two concerns derived from our research question: improving the reusability of DSLs implementations and improving their runtime performances. Also, placing ourselves

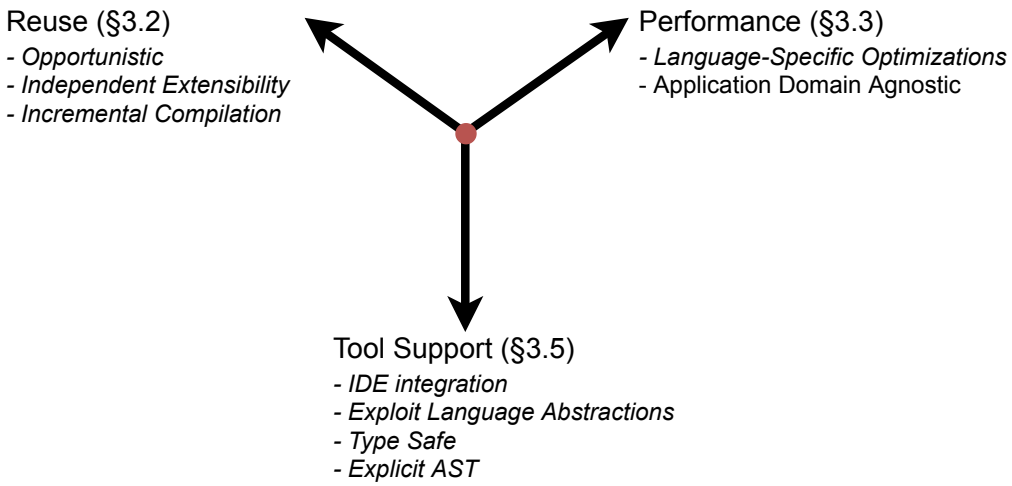


Figure 3.1 – Presentation of the concerns of the State of the Art.

in the context of the development of complex systems, supported by MDE, we aim at preserving the use of existing high-level metalanguages and their tool support.

From these observations, we selected the three following concerns: reuse, performance, and tool support detailed below. Each concern presents the dimensions we deem of interest to qualify them.

Reuse To evaluate if an approach to language development promotes reusability is not a trivial task. We focus on three dimensions to qualify language reuse: opportunistic reuse, independent extensibility, and incremental compilation. Those dimensions are drawn from the Language Extension Problem, presented in Section 4.1.

The Language Extension Problem is inspired by the seminal Expression Problem initially proposed by Walder [169], and later explored by many [155, 180]. More specifically, the Language Extension Problem lifts the Expression Problem at the level of software languages.

Opportunistic Reuse In the spirit of information hiding and encapsulation, the development of language must not necessarily be done with reuse in mind; in other words, reuse should not have to be anticipated. At the level of language service implementations, that also implies the use of implementation patterns that adapt modularily to existing legacy artifacts. In other words, without the need to regenerate or edit existing code to proceed to the composition.

Independent Extensibility It should be possible to extend languages independently in terms of syntax and in terms of semantics. New *syntactic* variants should be easily

adapted to handle existing semantics, and new *semantic* variants should handle pre-existing syntaxes. It should be possible to extend languages in a non-linear way, allowing the composition of independent extensions together.

Incremental Compilation Existing implementations of the syntax and semantics of language modules should not be modified, duplicated, or recompiled. Whole language compilation would require access to the source code of the base language — which might not be available — and would incur a non-linear performance penalty when compiling extensions.

Performance We focus on two dimensions to qualify the performance concern: language-specific optimization, and application domain agnostic.

Language-Specific Optimization We already stated that software languages are software too [57]. Consequently, general-purpose performance optimizations apply to software language implementations. However, additional optimizations can be applied to language implementations by taking into account the language-specific concepts manipulated on the language specifications. Indeed, reasoning at the level of language specifications allows the application of more relevant optimizations on the language implementations. Hence, achieving better runtime performance compared to general-purpose optimizations.

Application Domain Agnostic Whereas performance optimizations should take into account the specificities of languages, optimizations must not anticipate the application domain of languages. In other words, optimizations should be universal and should not anticipate specific application domains. With this dimension, we promote the applicability of performance optimization techniques to more and possibly unanticipated application domains.

Tool Support The work of software engineers is best done with the support of the relevant tools, and the introduction of additional reusability or performance should not interfere with the existing tool support of languages. While our goal is to improve the non-functional properties of language implementation, we consider that this should not be achieved at the cost of more restrictive or more complex language development tools and methods. Consequently, we selected four dimensions that help us qualify tool support in the context of MDE: IDE Integration, Exploit Language Abstractions, Type Safe, and Explicit AST.

IDE Integration The tool support of language development should be integrated into a language workbench that supports the tools that reflect the methodology supporting existing language development paradigms. Additionally, switching paradigms is costly and slow and eventually breaks the compatibility with legacy language specifications.

Exploit Language Abstractions The integration of non-functional properties on language implementation should not require the introduction of specific abstractions (e.g., patterns or annotations) at the level of language specifications. In other words, it should be done based solely on abstraction dedicated to the definition of languages, allowing language engineers to focus on the functional aspect of languages.

Type Safe We aim at maintaining the static type safety usually found in popular modeling workbenches such as EMF.

Explicit AST We place our works in a modelware approach. That entails additional properties required for the compliance to existing metamodeling tools. Metamodels are typed structure, guaranteeing the well-formedness of their models, hence the typing relation between models, metamodels, and semantics must be enforced and verifiable.

3.2 Reuse in Software Language Engineering

Languages are complex artifacts that require specialized development skills. Moreover, the same development process typically repeats itself from scratch for every new DSL. In other words, reuse rarely occurs in practice during the development of DSLs. Moreover, to foster their adoption, the productivity benefits of DSLs must offset the initial effort invested in their creation. Therefore, researchers have provided tools and techniques to assist language engineers in the development of new DSLs by reusing — part of – legacy DSLs.

DSLs are, by their very nature, meant to be tailored to a particular domain of application. Although this may suggest that there are few opportunities for reuse from one DSL to another, it is not uncommon to see different DSLs share recurring paradigms (e.g., state-transition, workflows, actions, and queries, units) [132]. Over the years, researchers have proposed many approaches seeking to improve language reuse. We give a brief overview of these in the following sections. Many of these techniques ultimately materialize as features of a language workbench, as presented in Section 2.3.3.

One important notion related to the reuse of languages is their variability. Svahnberg et al. [153] distinguish two types of variability, open and closed variability that encompass the two main views on reuse and are defined in those terms: “[...] a variation point is open for adding new variants or for removing old ones. During all other phases, it is not possible to change the set of available variants, and then the variation point is closed”. We discuss later how each type of variability impacts the reusability of languages.

Specification techniques for DSLs reuse are diverse, and this diversity is reflected by the richness of the contributions to the topic. From attribute grammars [179] to Parsing Expression Grammars [60] and scannerless parsing [163], the literature is rich in reuse techniques for grammars. The same can be said of the metamodeling world, where the problem of language reuse often boils down to the problem of metamodel reuse [49]. In

both cases, solutions either require the definition of explicit language modules that have built-in modularity mechanisms (e.g., extension points) or propose reuse operators that operate on existing languages to build new ones [41].

On the semantics side, formal approaches have been modularized (e.g., modular structural operational semantics [118], modular denotational semantics [106]) as well as their concrete realization e.g., in K [141], Redex [58], or DynSem [161].

The question of language reuse is also studied at the language implementation level. Researchers have thus developed reuse-oriented design patterns to implement such concrete artifacts modularly (e.g., [66, 104]). While most of these seek to reuse languages that are developed within the same technological space or that run on the same virtual machine, other authors tackle the problem of composing heterogeneous languages [9]. Recent advances in projectional editing also pave the way for tool-supported language composition [164].

A recent advance in SLE is the notion of language product lines [112, 94]. Language product lines are derived from software product lines [134]. A software product line can be defined as a set of software that shares a common core of functionalities while presenting different features dedicated to the adaptation of the core to specific contexts. Language product lines are then product lines where the products are languages. Engineering a language product line requires expressing the variability of such a family of languages. Typically, features of the variability model correspond to language features. Language features are reusable pieces of language that can be composed to derive a new specialized language variant from the family, according to a particular configuration. Language product lines typically make explicit the *closed* variability of a family of languages: the set of features of a language can be tailored, but it is hard to further adapt the resulting language to a new context of use.

Finally, the question of composition is not restricted to language engineering. Ferré et al. [59] address the construction of customized logics by the composition of independently defined logic functor, than can be considered as a form of modules. Their approach distinguishes between the Programmer that defines the logic functors, and the Application Designer that uses the functors to defines a customized logic for specific domains. This is comparable to the relation between Language Engineers and Domain Experts presented in our work.

In the remainder of this section, we study the approaches that address the reusability of languages: first at the syntactic level (Section 3.2.1), then at the semantic level (Section 3.2.2).

3.2.1 Syntax Reuse

As explained in Section 2.3.1.a, two main abstract syntax formalisms exist: metamodels and (context-free) grammars. The question of reuse is studied for both alternatives and resulted in different, yet complementary, contributions.

Syntax Reuse and Grammarware The reuse of grammar is a topic that raised much interest over the years and is known to be challenging for two reasons: parsing performance and ambiguity resolution. Indeed, the composition or extension of grammars often leads to exponential combinatorial complexity or ambiguities. The initial research challenge regarding reuse in grammarware was the exploration of parser implementation patterns that support efficient and modular parsing. However, parsing implementations, even following well-defined patterns, are notoriously difficult to write and maintain. Hence, a large body of work focuses on the definition of high-level formalisms that allow the specification of modular grammars while ensuring essential properties such as debugability, error reporting, safe and modular reuse, or separate compilation. Schwerdfeger et al. present a summary of parsing techniques and their limitations regarding their extension and composition [145]. Of course, many works offer approaches to mitigate such limitations.

Kaminski et al. [84] propose an approach that allows the static identification of composition conflicts of attribute grammar and the automatic (i.e., without expert knowledge from the part of the user that proceed to the choice and composition of the composed languages modules) composition of extensions over a base language.

Butting et al. [22] propose an approach to the systematic composition of software language syntaxes by allowing the definition of under-specified grammars with abstract production rules. This modular definition of grammar allows closed-world reasoning through the definition of language families using language product lines [183, 94].

Xtext offers a bridge between the grammarware and modelware technological spaces by synchronizing a metamodel from a type-annotated (E)BNF notation that relies on the LL(*) parser generator Antlr 4 [130], and vice-versa. Xtext supports only the linear extension of grammars by overloading or addition of production rule. Hence, it does not allow the modular composition of independently developed grammars.

Object Grammars [151] tackles the bidirectional mapping between graph-based object models and BNF grammars. Hence, bridging the gap between grammarware and modelware. In addition, Object Grammars also address the question of reusability through the questions of composability and extensibility.

Syntax Reuse and Modelware In the context of modelware, the syntax is represented by a metamodel. We saw in Section 2.1 that metamodels are also models. Hence, the question of syntax reuse can be reduced to the question of model reuse. The question of metamodel reuse has been introduced early [102], and lead to research efforts dedicated to a better understanding of metamodels and their properties. Emerson et al. [49] identify the need for a tool-supported approach of MDE, in particular for tasks related to reuse. Similarly, Vallecillo [159] motivates the use of DSL composition to mitigate the complexity inherent to the development of complex systems using models.

The effort toward syntax reuse on modelware can be classified into two categories. On the one hand, approaches that promote the definition of modules with a strict notion of encapsulation. First, the keywords-based modularization [31] supports the construction of

DSLs through the definition of DSLs modules and their composition. The abstract syntax is defined in an object-oriented model, conceptually close to a metamodel, and the semantics is defined using a translational semantics formalism. Then, Zivkovic et al. [182] introduce the notion of language modules with explicit interfaces and emphasize the interest of information hiding and its impact on module composition. Finally, Wende et al. [171] propose a role-based approach to the definition of language modules, integrated into the LanGems language workbench [46].

On the other hand, we can find approaches that define new kinds of relationships between modules. First, Meta Programming System (MPS) [77] is a projectional editor in which users directly manipulate the AST through graphical projections of the AST in the editor. Projectional editors can mimic the look and feel of textual editors by fine-tuned interaction with the graphical projection of characters streams. MetaMod [152] is built on top of MPS and defines a metamodel formalism dedicated to the definition of modular metamodels. Modularity specific concepts are introduced in an ad-hoc metalanguage formalism build to improve reusability in modeling. Steel et al. propose the notion of model typing [149], later extended to model subtyping by Guy et al. [67]. Both works formalize the notion of typed relation between models to improve their safe and flexible reuse. Degueule et al. exploit the notion of interfaces and study the specificities of its application on languages [40]. The notion of language interface is applied as a first-class entity in the Melange language workbench [41], which allows the composition of language modules while preserving the compatibility with pre-existing tool support. Melange none the less requires access to language module specification and does not provide support for incremental compilation. MetaDepth [100] introduces the notion of *a-posteriori typing* that allows the definition of a typing relationship between two modeling levels at use time instead of definition time. This flexibility in typing opens the opportunity to reuse metamodels in unanticipated contexts. Finally, Cuadrado et al. [39] propose an approach to the reuse of metamodels in unanticipated contexts by defining an open Meta-Object Protocol mechanism [86, 87] for metamodels. That way, metamodel semantics can be extended modularly.

3.2.2 Semantics Reuse

The idea of reusing language specification is a topic well-studied for the definition of language semantics too. Similarly to the syntax, the reuse of semantics has been explored at the level of language specifications as well as language services implementation.

Starting with language specifications, the time-honored Structural Operation Semantics (SOS) notation, used to define operational semantics formally, have been modularized through many incremental modifications of its notation [118, 119]. The notion of reusable components of semantics specification is proposed on top of these notations [29]. Those components are called *funcons* and are intend to be defined "once and for all" and be highly reusable. While *funcons* are dedicated to being reusable for the specification of language semantics, *funcons* are composed of the specification of their syntax and semantics, which

are built to be composed with all the other *funcons*. Experiments have been conducted to port the funcons to the K framework [120] and Spoofax [14], allowing the specification of executable and reusable language interpreters modules.

More generally, semantics formalisms have the expressiveness to abstract away the modularity-specific details and provide type checking mechanisms to ensure the well-formedness of modules composition. Kermeta [79] allows the specification of operational semantics over metamodels, following an open-class mechanism [32].

Kaminski et al. [84] extends the Expression Problem and introduce an additional constraint, the automatic composition of language extensions. No "glue code", i.e., code dedicated solely to the interconnection of extensions, must be required from the user to compose a collection of language extensions. Consequently, language extensions can be composed safely by users, even without knowledge of language engineering.

3.2.2.a Interpreter-Based Implementations Reuse

At the level of language semantics implementation, the operational semantics of languages is often implemented in the form of interpreters [64]. Many variations of the interpreter pattern have been proposed to address the problem of semantics implementation reuse.

In the functional programming paradigm, many approaches to modular interpreters have been proposed. This includes tagless interpreters [24], monad transformers [107], lightweight modular staging [139], Cake Pattern [74], or Data Types à la Carte [154]. All those approaches share the capability to define extremely reusable language modules. However, they all rely on advanced type-system mechanisms that are not found in mainstream object-oriented languages, which hampers their larger adoption.

In the object-oriented paradigm, many works address the implementation of modular interpreters. First, Object Algebras [124] bridges the gap between functional and object-oriented styles by allowing the definition of tagless-like interpreters, where the structure of programs is defined in the form of abstract operations. However, unlike tagless interpreters, Object Algebras does not require advanced type systems features such as F-bounded quantification [23]. Multiple contributions extend over the Object Algebras. Oliveira et al. [142] apply Object Algebras to the feature-oriented definition of language modules. Inostroza et al. [76] define reusable modular languages on top of Object Algebras, by lifting existing semantics in unanticipated contexts. Finally, Zhang et al. [181] propose an approach that allows the definition of extensible visitors on explicit metamodels using mainstream object-oriented features and code generation from annotated Object Algebra.

3.2.2.b Visitor-Based Implementations Reuse

The Visitor pattern [64] is probably the most used object-oriented implementation pattern for language semantics but suffers from a lack of modularity when new syntactic variants are introduced. Many approaches aim at mitigating the limitations of the Visitor pattern. Oliveria proposes two approaches to implement modular visitor components [123]

but requires the use of features specific to the Scala language and not available in mainstream object-oriented languages. Wang et al. [170] propose a variation of the Visitor pattern by exploiting the covariant type refinement of return types, available in most mainstream object-oriented languages, to implement modular visitor. However, this approach is limited in its applicability because it forces reuse to be anticipated, contradicting to *Opportunistic Reuse* dimension. Finally, Torgersen proposes four solutions to the Expression Problem [155] in the form of variations of the usage of genericity in Object-Oriented programming. Nonetheless, the applicability of those solutions requires anticipation during the definition of the reused module.

3.3 Runtime Performance Optimization in Software Language Engineering

In this section, we first describe the main concepts and research related to the runtime platforms underlying the execution of languages. Then, we explore the contributions related to the optimization of the languages themselves.

Performances of compilers and interpreters [54] have been deeply studied, and compiled programs are usually more efficient, especially when the compiler performs optimizations that benefit from platform-specific performance features (e.g., according to a given processor architecture). Additionally, interpreted programs pay the price of the interpretation overhead. This overhead is due to the execution of the interpreter itself, in addition to the program. With regard to performance optimization, one interesting advantage of interpreted languages over compiled ones is their ability to take into account their runtime information (e.g., input parameters or runtime data). Such information is either unavailable at compilation time, or difficult to infer and can help interpreters to take performance-related decisions during program execution.

At the frontier of compilation and interpretation stands the Just-In-Time (JIT) compilation [6]. The JIT performs the compilation of fragments of a program during its interpretation. The objective of such a mechanism is to: identify the relevant parts of the program that would benefit the most from the compilation; then, proceed to their compilation in parallel to the interpretation. This JIT compilation process itself has a price in terms of performance. This price is expected to be compensated by the speedup induced by the substitution to a faster compiled version. Conceptually, JIT and static compilation — also called Ahead Of Time (AOT) compilation — are similar, but are constrained differently. When AOT lacks knowledge of the runtime context, JIT is instead constrained by its own execution cost.

The lightweight modular staging [139] approach addresses the performance challenge by introducing staging information in interpreter implementations in the form of typing information. The compiler can then infer from the types which expressions can be directly interpreted during the compilation, reducing the amount of interpretations steps and indirections required during the execution of the compiled interpreter. LMS has been

applied for the implementation of multiple high-performance embedded DSLs, for instance, an SQL interpreter [138], and a regular expressions parser [140].

To the best of our knowledge, only two approaches address the language-agnostic performance optimizations: tracing and partial evaluation. Only one implementation of each approach exists, respectively RPython [15, 16] and Truffle [178, 176]. Smarr et al. compare their performances by implementing SOM [68], a subset Smalltalk dedicated to teaching and research on virtual machines, once on each platform, and running performance benchmarks. This kind of approach has the great advantage of offering more accurate runtime optimizations by reasoning in terms of language concepts. Nevertheless, writing language implementations using such approaches requires advanced language programming skills in addition to the understanding of the advanced concepts related to tracing or partial evaluation.

Vergu et al. [162] address the optimization of the performance of the metainterpreter of DynSem, a metalanguage integrated into the Spoofax language workbench. By operating at the level of metainterpreters, Vergu et al. are application domain agnostic. Nonetheless, this approach does not allow the introduction of language-specific optimizations prior to its execution.

Recently, Shaikhha et al. [146] propose a performance-oriented approach to the definition of Internal DSLs. They offer a set of dedicated scala annotations (e.g., compiler or reflected types) to language engineers that assist the compiler in its analysis of the DSL specification. This approach is based on the explicit definition of performance concepts by language engineers and requires the introduction of dedicated concepts on top of language specifications.

Another less explored research direction is to trade accuracy for performance, namely the field of approximate computing. Such an approach has been applied to various optimization goals, such as energy consumption [70], and of course, runtime performance [137]. However, efforts are focused on specific application domains and are not applicable to more general application contexts.

Languages can also be optimized by introducing domain-specific optimization to existing languages. This vision if often realized by the definition of a base GPL and an extension mechanism that allows the modular introduction of domain-specific optimization. That way, language engineers can adapt base GPLs to specific contexts. Contributions to this vision have been realized on top of Haskell's compiler [80], extensible compiler frameworks [121], and Attribute Grammars [48].

In summary, language runtime performance optimization requires development efforts and performance-specific knowledge from language engineers. Furthermore, many approaches break the compatibility with the existing tool support of languages or are not applicable on the context of MDE due to their lack of support for the mutability of the AST.

3.4 Tool Support in Software Language Engineering

In the previous sections, we have listed the approaches that contribute to the reusability (Section 3.2) and the performance optimization of languages (Section 3.3). In this section, we study how dedicated language engineering tools support language development. More precisely, we list language workbenches and study how they support the development of languages. In addition, we study how the optimization of non-functional properties impacts the tool support of those language workbenches. To do so, we study the existing language workbenches dedicated to the definition of External DSLs and details their properties regarding the selected dimensions.

The GEMOC Studio The GEMOC Studio is a language workbench built on top of Eclipse and EMF. It targets both language engineers and domain experts. It focuses on the automatic and generative production of services supporting the development and use of languages (e.g., debuggers, executions traces). Ecore is used for the specification of abstract syntaxes and Kermeta [79] for the modular specification of operational semantics on the Ecore metamodels. Melange [41] is integrated into the GEMOC studio and builds on top of these specifications and provides a metalanguage to specify the modular composition of language modules. Méndez-Acuña et al. provide solutions for the top-down and bottom-up manipulation of language product lines using the GEMOC Studio [113]. Current reuse approaches based on the GEMOC studio either presume a close-world, which contradicts the *non-anticipation* or does not support incremental compilation at the implementation level.

LISA LISA is a language workbench that supports the definition of textual DSLs using attribute grammars. From attribute grammars specification, LISA derives compilers or interpreters and a set of language-specific tools (e.g., editors, visualizers). LISA supports multiple inheritance of attribute grammars [116], allowing incremental language development through various composition operators [114]. However, LISA fails to support modularity at the implementation level, which prevents the opportunistic reuse of existing languages.

Monticore Monticore [93] is a language workbench dedicated to the definition of textual DSLs using a Grammar-based metalanguage that mixes an (E)BNF notation with concepts required to define abstract syntax in the form of arbitrary graphs of objects, similar to metamodels. Monticore allows the construction of languages by reuse through multiple inheritance and embedding [71]. Recent contributions on the definition of the notion of language module interface have been integrated into Monticore through the definition of partially defined language modules, allowing the safe composition of independently defined language modules [22, 21]. However, Moticore requires the manipulation of paradigms specific to its approach, contradicting the *Exploit Language Abstraction* dimension.

MPS The Meta Programming System (MPS)[1] is a language workbench developed by JetBrains and based on projectional editing. MPS has been notably exploited for the definition of a C development environment [167], or the definition of a modular, reusable, and extensible expression language [165]. MPS offers dedicated metalanguages for each aspect of language definition (e.g., syntax, projectional rules, types-system). Being projectional, MPS circumvents the issues related to the composition of concrete syntaxes.

Neverlang Neverlang [158] is a language workbench that promotes the idea of feature-oriented language development, in which languages can be built by composition of small reusable fragments, called *slices*, that embody the syntax and semantics of a language feature. The syntax is specified using a context-free attributed grammar formalism, and the semantics is specified using a translational semantics formalism. Slices can be compiled separately. Artifact resulting from slices compilation can then be reused independently of their specifications [27]. Such modularity has been applied to the idea of Language-Oriented Programming (LOP) to grow a family of Javascript-like languages for teaching purposes [25]. However, Neverlang requires the manipulation of specific concepts, out of the scope of the usual paradigms manipulated by software engineers, contradicting the *Exploit Language Abstractions* dimension.

Rascal Rascal is a metalanguage dedicated to the analysis and transformation of source code [89]. Additionally, it can be used as a language workbench integrated into the Eclipse IDE. The extreme expressiveness of Rascal allows the definition of the concrete syntax, abstract syntax, and semantics of languages in a single metalanguage. Rascal supports the generation of Eclipse plug-ins that act as tool support for DSLs (e.g., text editors, web integration, IDE outlines) from their specifications. The modularity of Rascal has been evaluated [11], by growing the Oberon-0 language [173] through multiple modular extensions. Rascal breaks the *Exploit Language Abstraction* dimension by offering advanced, yet powerful, metaprogramming concepts to the user, out of the scope of the usual paradigms exploited in the context of SLE.

Spoofax Spoofax is a language workbench built on top of the Eclipse IDE that provides a set of interconnected domain-specific metalanguages, each dedicated to the specification of an aspect of languages [85]. The SDF3 metalanguage is dedicated to the specification of concrete syntaxes using a context-free grammar formalism. ATerm is used for the specification of the abstract syntax, in the form of abstract data types. Language semantics are specified using Stratego [20], or more recently using DynSem [161]. Spoofax supports language extension and composition but requires anticipation in the design of the reuse language modules.

[1] MPS: https://www.jetbrains.com/mps/

Xtext Xtext is a language workbench that allows the definition of the abstract and concrete syntax of languages using a unified metalanguage. It mixes a (E)BNF notation annotated with type information, allowing the derivation of an EMF metamodel additionally to the generation of a parser from the grammar. An alternative mode can be chosen, where the metamodel already exists. In this case, the syntax is type-checked against the metamodel structure. Xtext relies on Antlr for the generation of the parser implementation. Xtext allows the modular extension of the abstract syntax by a single inheritance mechanism, allowing the overloading of existing rules or the introduction of new rules. While modular at the specification level, the compilation of the extensions involve the full recompilation of the parser implementation. Xtext semantics are realized by interpretation of Ecore models. Xtext uses Xtend[2] for the specification of language semantics. Xtend supports a dispatch mechanism, allowing a Visitor-like mechanism directly in the language syntax. It also has a native template system, easing the specification of translational semantics. Xtext is seamlessly integrated into the Eclipse IDE and supports the integration of many editing services (e.g., editor, jump to the declaration, outlines, error reporting...) and is based on mainstream object-oriented mechanisms, but fail at supporting proper language reuse because of its single inheritance mechanism.

3.5 Advancing the State of the Art

Table 3.1 and Table 3.2 summarize the concerns referenced in this study and their alignment with the dimensions we identified for each concern. Table 3.1 shows the approaches related to the concerns of language reuse and tool support. Approaches focusing mainly on the reusability of the syntax are qualified as out of scope regarding semantics extensibility. Approaches based on interpretation of language specification are qualified as out of scope regarding incremental compilation. Table 3.2 shows the approaches related to the concerns of language performance and tool support. We can observe that for each combination of axes, no existing approach fulfill the selected dimensions entirely. Approaches that rely on extensible compilers are not included as they require compiler construction skills that are outside of the scope of this thesis.

In summary, most of the approaches we introduced in this chapter support the definition of reusable language modules, or the definition of efficient languages but requires software engineers to learn new paradigms, and does not rely on existing language abstractions. Similarly, tool-supported approaches to language development fail on the axes of reuse or performance. The dimensions related to the reuse concern are often not reached, often due to a lack of support for incremental compilation at the implementation level. Regarding the performance concern, whereas many contributions target the runtime performance of internal DSLs, very few works target the automatic optimization of external DSL.

Finally, no existing contributions address the challenge of improving the tool-supported development of languages while ensuring their reusability and optimizing their runtime

[2]Xtend is itself developed using Xtext.

Table 3.1 – Summary of language reuse approaches and their tool support.
● = supported, ◐ = partially supported, ○ = not supported, / = out of scope

Approaches	Performance				Tool Support			
	Opportunistic	Independent Extensibility		Incremental Compilation	IDE Integration	Exploit Language Abstractions	Type Safe	Explicit AST
		Syntax	Semantics					
Interpreter [64]	◐	●	○	◐	○	○	◐	●
Visitor [64]	○	○	●	◐	○	○	◐	●
Object Algebras [124]	●	●	●	●	○	○	●	○
Kermeta [79]	●	●	●	○	●	●	●	●
Trivially [170]	●	●	●	●	○	○	●	●
AbelC (Kaminski et al.) [84]	◐	●	●	○	○	●	●	○
MontiCore [93]	●	●	●	○	●	○	●	●
Xtext [55]	●	●	/	●	●	●	●	●
Antlr 4 [130]	●	●	/	○	●	●	●	●
Object Grammars [151]	●	●	●	/	●	○	●	●
MPS [77]	●	●	●	●	●	◐	●	●
Metamod [152]	○	●	●	●	●	○	●	●
Melange [41]	●	●	●	○	●	●	●	●
MetaDepth [100]	●	●	●	/	●	○	●	●
tagless interpreters [24]	●	●	●	/	○	○	●	○
monad transformers [107]	●	●	●	/	○	○	●	○
lightweight modular staging [139]	●	●	●	●	○	○	●	●
cake pattern [74]	●	●	●	●	○	○	●	○
datatype à la carte [154]	●	●	●	○	○	○	●	●
EVF [181]	○	●	●	●	●	○	●	●
Oliveria's visitors [123]	●	●	●	●	○	○	●	●
Torgersen [155]	○	●	●	●	○	●	●	●
LISA [116]	●	●	●	○	●	●	●	●
Neverlang [158]	●	●	●	●	◐	○	●	●
Rascal [89]	●	●	●	/	●	○	●	●
Spoofax [168]	●	●	●	●	●	○	●	●

Table 3.2 – Summary of language performance optimization approaches and their tool support.
● = supported, ◐ = partially supported, ○ = not supported.

Approaches	Performance		Tool Support			
	Langage-specific optimizations	Application Domain Agnostic	IDE Integration	Exploit Language Abstractions	Type Safe	Explicit AST
Lightweight Modular Staging [139]	●	●	○	○	●	●
RPython [15, 16]	●	●	○	○	○	●
Partial Evaluation [178, 176]	●	●	○	○	○	●
Dynsem + Truffle (spoofax) [162]	○	●	●	●	●	●
Shaikhha [146]	●	●	◐	○	○	●

performances.

In the next part (Part II), we present new contributions that address the identified limitations by providing approaches to reuse and performance optimization of languages while providing tool support for software engineers through their integration in language workbenches.

PART II

Contributions

CHAPTER 4

LANGUAGE REUSE

In this chapter, we develop our contribution to the reusability of DSLs. We first introduce the Language Extension Problem (LEP), inspired by the Expression Problem (EP), and detail its implications to language reuse (Section 4.1). Then, we present two motivating examples and an overview of our approach to language reuse (Section 4.2). Finally, we present in detail our approach for language reuse (Section 4.3). We close this chapter drawing some conclusions on the proposed contribution (Section 4.4). This chapter is partially based on our MODELS'17 [104] and SLE'18 [103] publications.

4.1 The *Language Extension Problem*

With the advent of language workbenches, the problem of modular language extension has garnered considerable interest from the research community in the past decade. This problem informally refers to the capability of extending the syntax and semantics of an existing language while reusing its specification (e.g., grammars, semantic inference rules) and implementation (e.g., parsers, interpreters). Various authors have attempted to formalize this problem (e.g., [50]) but the lack of a clear definition makes it hard to evaluate and compare the strengths and weaknesses of existing solutions w.r.t. a common, well-defined framework. This section is an attempt to define language extensibility in the form of a well-defined problem.

4.1.1 From the *Expression Problem* to the *Language Extension Problem*

Philip Wadler coined the term "*Expression Problem*" to name a well-known problem in the programming languages community and this name has been in common use for more than two decades [169]. As Oliveira and Cook put it [124]:

> The "expression problem" (EP) is now a classical problem in programming languages. It refers to the difficulty of writing data abstractions that can be easily extended with both new operations and new data variants.

Over time, the EP has made it possible to structure the discussions around the capabilities of different programming paradigms and languages regarding data types extensibility using a common set of constraints that candidate solutions should address. There are different variations of the EP, but its canonical definition includes the following constraints [180]:

Extensibility in both dimensions: It should be possible to add new data variants and adapt existing operations accordingly. Furthermore, it should be possible to introduce new operations.

Strong static type safety: It should be impossible to apply an operation to a data variant that the operation cannot handle.

No modification or duplication: Existing code should neither be modified nor duplicated.

Separate compilation: Compiling datatype extensions or adding new operations should not encompass re-type-checking the original datatype or existing operations.

Independent extensibility: It should be possible to combine independently developed extensions so that they can be used jointly.

There is a striking parallel between the problem of modular language extension and the *Expression Problem*. As a matter of fact, most approaches to modular language extension end up discussing and addressing the EP in some way [84, 104]. However, in the context of Software Language Engineering (SLE), "data variants" are groups of syntactic categories and their constructors and "operations" over these data variants define their semantics. Due to the ambiguity in the name *Expression Problem*, in which "expression" may refer to a *language* of expressions, one might naively think that the EP and the problem of modular language extension are equivalent.

In this section, we demonstrate that instantiating the EP in the context of SLE requires a reformulation and refinement of the existing constraints of the EP and to introduce new ones, leading to a new problem: the *Language Extension Problem* (LEP). **While the EP is merely a programming problem concerning programmers and focusing on the extensibility of a single datatype, the LEP is a Software Language Engineering (SLE) problem concerning language engineers and considering the extensibility of languages (i.e., group of datatypes representing the language constructs). The LEP must also account for engineering practices that are specific to software languages such as the use of language workbenches, the duality of language specifications and implementations, and the specificities of syntax definition.** As extending a group of datatypes entails extending the datatypes it is composed of, in many cases solving the LEP (in the large) entails solving the EP (in the small). We identify what is the meaning of the two extension axes in this context and what is the set of constraints that must be used to assess the success of a given solution. Naturally, many partial solutions to the LEP already exist in the literature, scattered from programming language theory (e.g., modular visitor components [123], Revisitors [104], Recaf [13]), to language workbenches (e.g., Rascal [89], Melange [41], Silver [179, 83]). We purposely limit ourselves to the definition of the LEP, and leave to future work the positioning of existing solutions w.r.t. the constraints we list.

4.1.2 The *Language Extension Problem*

The *Expression Problem* has been initially introduced in the context of datatype extension and composition, hence presented in terms of *datatypes* and *functions* over the datatypes. Conversely, the *Language Extension Problem* focuses on language extension and composition, presented in terms of, respectively, *syntax* in the form of multiple syntactic categories and constructors for each category, and *semantics* over that syntax. In the following, we introduce the *Language Extension Problem* by paraphrasing the original definition of the *Expression Problem* by Wadler [169], but lifting the vocabulary from datatypes to languages.

> The *Language Extension Problem* (LEP) is a new name for an old problem. The goal is to define a family of languages, where one can add a new language to the family by adding new syntax (i.e., new constructors for existing syntactic categories as well as new categories) and also new semantics over existing and new syntax, while conforming to constraints similar to those in the *Expression Problem* but specialized to language extension.

As an example, consider a language family regarding state machines which starts from a core language over simple finite state machines with a simple pretty-printing semantics and constructs a new language by adding syntax to specify hierarchical state machines and a new semantics to evaluate state machines given an input sequence.

According to this characterization of the LEP, we now review the constraints initially identified in the EP, and express them in the context of SLE for the LEP:

Extensibility in both dimensions: It should be possible to extend the syntax and adapt existing semantics accordingly. Furthermore, it should be possible to introduce new semantics on top of existing syntax.

Strong static type safety: All semantics should be defined for all syntax.

No modification or duplication: Existing language specifications and implementations should neither be modified nor duplicated.

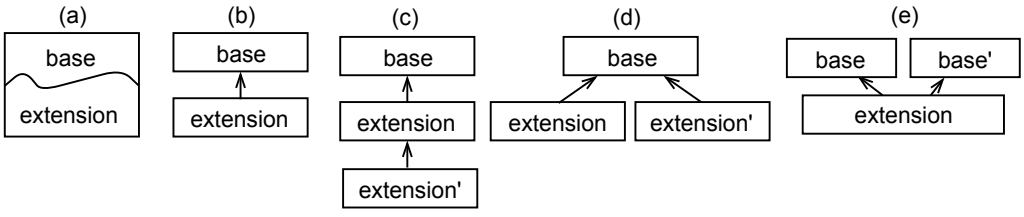


Figure 4.1 – Approaches for language extension, applicable at the specification and implementation levels. (a) mix up the extension into the base language, while (b)-(e) keep them separated and use explicit operators (e.g., references, static/dynamic introduction) or glue code.

Separate compilation: Compiling a new language (e.g., syntactic extension or new semantics) should not encompass re-compiling the original syntax or semantics.

Independent extensibility: It should be possible to combine and use jointly language extensions (syntax or semantics) independently developed.

Moreover, the distinction between the specification and the implementation of new engineered languages raises a new concern regarding **automatic composition** [84]. Indeed, "glue code", i.e., code dedicated solely to the interconnection of extensions, must be limited or avoided from the user's point of view to compose a collection of language extensions.

4.1.3 The Language Extension Problem in Practice

Numerous approaches have been explored in the past decade to address specific scenarios of language extension, either at the specification level (e.g., [22, 42, 52, 83, 103, 114, 157, 164, 168, 180]) or at the implementation level (e.g., [72, 155, 180, 181]). The specification level is based on meta-languages that provide the relevant abstractions, often with limited and domain-specific expressiveness. The specification level is then turned into an implementation thanks to compilation or interpretation, often by targeting general-purpose programming languages and following language implementation patterns specific to each approach.

While all those approaches are heterogeneous and conceptually operate at different levels, they share common extension mechanisms which are summarized in the five approaches depicted in Figure 4.1 [50].

Complying exhaustively to the identified constraints is extremely challenging, and trade-offs must be considered for a given context. We present a selection of scenarios illustrating such trade-offs. First, the constraint of separate compilation usually impacts other non-functional properties such as performance, readability, and accidental complexity (e.g., large and complex glue code, unclear modules dependencies). Consequently, it can be worthwhile to relax the separate compilation constraint in order to comply with other non-functional properties. Second, various actors can be responsible for language extension. They each come with very different skills, ranging from SLE experts with a deep understanding of languages and language workbenches internals, to end users with minimal knowledge of software development. While the former are capable of performing composition using complex handwritten glue specifications, the latter will typically require fully automatic composition approaches. Finally, the boundaries of a language family are important to consider. Two statuses can be considered, closed (i.e., all its languages are known) and open (i.e., new languages can be added organically). Indeed, in the context of closed families, the compatibility of the extensions can be checked in advance and conforming to the type safe constraints is not an issue. On the contrary, in the context of open language families, restrictive type systems can lead to difficulty or impossibility to extend languages in unanticipated contexts.

The constraints of the *Language Extension Problem* define a framework for comparing language extension approaches. It is worth noting that conforming or not to some of the

constraints is often the consequence of interesting language design choices, relative to some specific scenarios.

4.1.4 Wrap up of the *Language Extension Problem*

In this section, we have described the *Language Extension Problem*, a lift of the *Expression Problem* at the language level. We lift the constraints drawn from the *Expression Problem* to the context of software language engineering, and introduced an additional constraint specific to this context. Through the *Language Extension Problem*, we hope to provide a framework to reason on language extension and its challenges and help the comparison of existing and future SLE contributions.

4.2 Reusable Languages Motivation and Overview

In this section, we first present two motivational scenarios, addressing language extension (Section 4.2.1) and language composition (Section 4.2.2). Then, we introduce the notion of language module interfaces Section 4.2.3, and define the requirements that allow us to evaluate language reuse Section 4.2.4. Section 4.2.5 presents the application of our approach to language extension, and Section 4.2.6 presents the application of our approach to language composition. Finally, we discuss the scope and limitation of our approach Section 4.2.7.

4.2.1 Reuse of a Finite State Machine Language by Extension

Just as any software, DSL implementations are bound to evolve to meet new requirements of language users or the specificities of a new domain of application. One solution to do so is to reuse an existing language module and extend it to introduce the required specificities. This situation is highlighted by the idiomatic FSM language example depicted in Figure 4.2. The figure presents three variants of the language: a simple FSM modeling language, later extended to implement its operational semantics (represented as a set of methods woven on the corresponding concepts), and then extended to enable the expression of complex guards on the transitions. The latter requires to override part of the semantics of the xFSM language (namely the `step` method on `Transition`), to take into account the newly-introduced guards.

This refinement of a language by the successive introduction of more specific layers is close to the notion of mixin layers [99]. However, this approach does not address the question of reuse at the implementation level. Melange [41] also addresses the same kind of scenario but rely and subtype polymorphism instead of parametric polymorphism [40].

When facing such kind of language engineering scenarios, language engineers must be provided with appropriate tools to extend existing languages, customize their semantics, and combine such extensions. A solution to these problems in the context of model-driven engineering must satisfy a set of situational constraints: it must operate on an *explicit* and

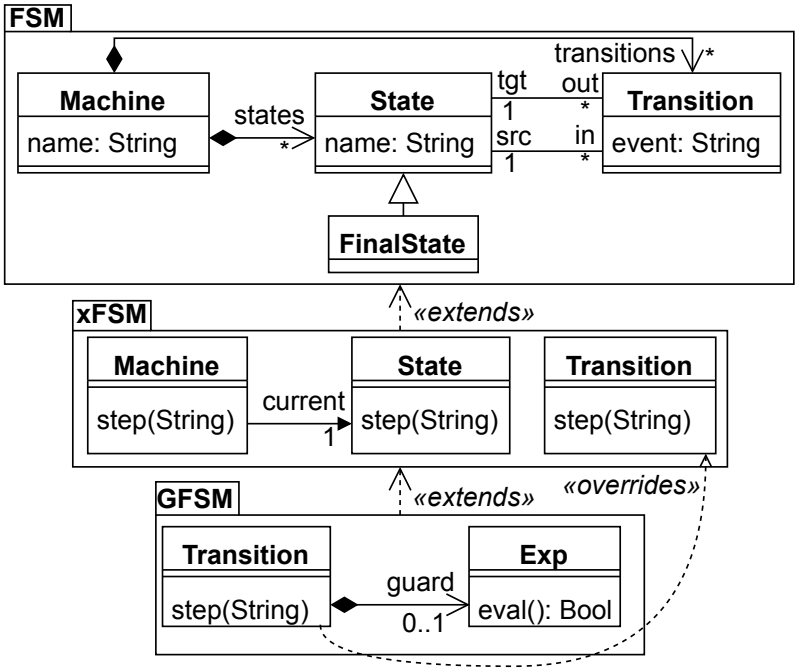


Figure 4.2 – Extensions of an FSM language

mutable AST, whose structure is prescribed by a metamodel, and maintain the *static type safety* usually found in popular modeling workbenches such as EMF.

4.2.2 Reuse of a Finite State Machine Language by Composition

We saw in the previous section the specialization of an FSM to a specific application domain by extension. In the case of language extension, a single language module is reused and specialized for a new application domain. Another interesting use-case is the reuse of independently developed languages by composition.

Let us consider the motivating example presented in Figure 4.3 depicting the metamodel of a simple FSM language module, quite similar to the FSM presented in the previous section (Figure 4.2). The main difference being the introduction of guards and actions on the transitions.

From a language engineer's point of view, multiple expression languages can be good candidates to express the guards. Similarly, multiple action languages are adapted for the expression of actions. Rather than defining new guard and action languages from scratch, including their syntax, semantics, and tooling, it would be handy to reuse and plug existing languages that provide these functionalities into the base FSM language. For instance, OCL [30] for the guards and Xbase [47] for the actions. This would allow domain experts to build FSM models by combining the expressiveness of the base FSM language with the expressiveness of dedicated expression and action languages, including their tool support.

The question that naturally arises is: how to express the *required interface* of the

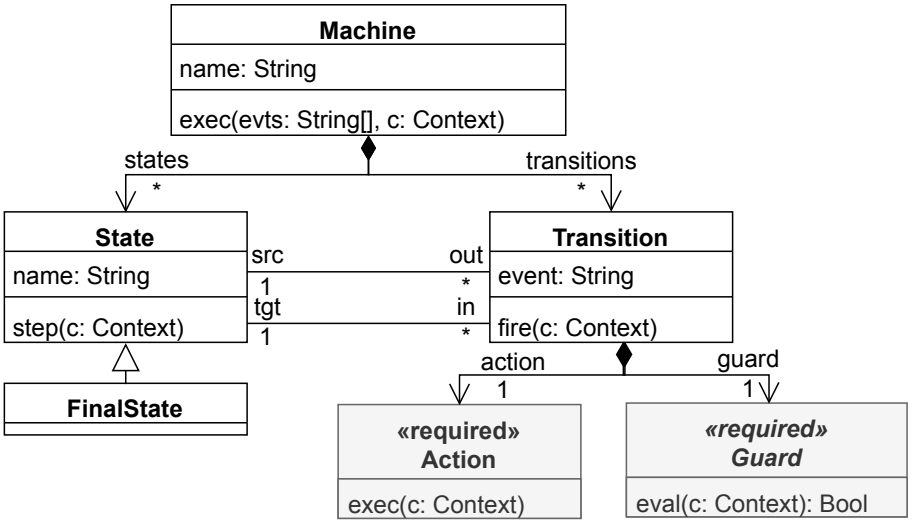


Figure 4.3 – An FSM language module with an explicit reuse interface.

FSM language? The notion of language interface, in general, is the subject of ongoing research [40]. In this section, we are specifically interested in the interfaces required for language composition. From the FSM's standpoint, a guard is merely "an expression whose evaluation returns a Boolean." That is, the signature of its evaluation function is $eval : Context \rightarrow Bool$. The internals of the expression language employed, e.g., its syntax (the set of Boolean operators it offers) and semantics (how they are evaluated) do not matter. In Figure 4.3, the `Guard` and `Action` construct annotated with `«required»` denote the required interface expected by the base FSM language. The key idea here is that most of the language's semantics can be implemented independently from the syntax and semantics of guards and actions. Knowing that actions and guards can be evaluated with their respective evaluation function is sufficient to express the semantics of transitions — only the *signature* of their evaluation function is needed. In pseudocode, the semantics of the `fire` evaluation function of transitions could be written as follows:

```
1 fire(Transition t, Context ctx) {
2   if (t.guard.eval(ctx))
3     t.action.exec(ctx);
4 }
```

An interpreter for the evaluation semantics of the FSM language can be type-checked and compiled independently; but to be run, it needs concrete implementations of the execution functions of guards and actions. These will be provided later by other language modules at composition time.

4.2.3 On Language Module Interfaces

A language module interface should expose the information needed to (i) use and (ii) compose a module [35]. Using a language first involves producing a conforming model.

The structural information, in the form of a metamodel, must thus be part of the module's interface. Then, execution functions are invoked on the different model elements. The set of execution functions, linked to the corresponding domain constructs, must also be part of the module's interface. In contrast, details of the syntax of the required constructs, as well as the implementation of their execution functions can be encapsulated behind the interface, and it should not be necessary to inspect them in order to use or compose a language module.

Another design choice to be considered by language engineers is the boundaries of a module. While we do not enforce strict rules for the definition of language module boundaries, we suggest following the well-known modularity principle of package cohesion which, in this context, states that (i) domain constructs that are commonly used together should belong to the same module, and (ii) a module should not have more than one reason to change. For instance, when designing the FSM module of Figure 4.3, the language engineer may wonder whether guards should be included as part of the language module or as part of its required interface. The `Guard` construct is clearly needed, but its concrete realization is subject to discussion. First, the implementation of the various constructs enabling the expression of guards is complex and will probably overtake the complexity of the rest of the module if it would be included. Besides, there are already existing expression languages that could fulfill this functionality and are evolving at their own pace, independently from the FSM module. Hence, the `Guard` construct is defined as `«required»` and is expected to be provided later by another language module through composition.

4.2.4 Language Reuse Requirements

From the introduction, motivating examples, and the notion of language module interface, we derive a list of seven requirements that must be addressed for the definition of reusable language modules.

One can observe that such requirements are very similar to the LEP constraints presented in Section 4.1. This is not fortuitous, but we list again the minimum constraints needed in the context of language extension and composition.

Independent Extensibility (R1) It should be possible to extend languages both in terms of syntax and in terms of semantics. New syntactic variants should be easily adapted to handle the existing semantics, and new semantic variants should handle pre-existing syntax modules. It should be possible to extend languages in a non-linear way, allowing to compose independent extensions together.

Incremental Compilation (R2) Language modules should be type-checked and compiled separately, and should not have to be edited or recompiled to be reused with other language modules. Whole language compilation would require access to the source code of the modules (which might not be available) and would incur a non-linear performance penalty when compiling languages build through reuse. Furthermore, language modules

should not make any assumption on the way they will be reused, i.e., they should not *anticipate* reuse.

Opportunistic reuse (R3) It should be possible to reuse existing implementations of languages without anticipation. A pattern relying on anticipation would require refactoring current modeling frameworks (e.g., EMF) to regenerate existing code, for instance, to insert `accept` methods to support the Visitor pattern. This would prevent the applicability of the solution to legacy artifacts generated from widely-used modeling frameworks and complicate the work of language engineers.

Module Encapsulation (R4) Language modules should be composed without having to inspect their internal implementation. In other words, the information exposed in the interfaces of language modules should be sufficient to enable the type safe composition of language modules.

Explicit Required Interfaces (R5) Required interfaces of language modules should explicitly state the requirements a module has towards other modules. Knowing the interfaces only should be sufficient to state the validity of the composition of different modules. A composition is valid if the requirements expressed by the required interface of a module are fulfilled by other modules, and if it can be ensured that the generated implementation will compile. Checking the validity of the composition of modules should be possible at the level of the metalanguage used for the definition of language modules, without requiring code generation.

Module Substitutability (R6) Two language modules providing constructs that match the same interface should be substitutable to one another in the context where this interface is required. From the requiring module's point of view, the choice of a particular language module should be transparent. Substitution of a language module by another should not require any modification of the language module which depends on it.

Non-intrusivity (R7) The definition of language modules satisfying the requirements above should not disrupt widespread language engineering processes, such as abstract syntax definition in the form of object-oriented metamodels. It should not require a new paradigm for the specification of language modules to be broadly applicable to mainstream language engineering technologies and to foster its adoption.

4.2.5 Language Extension Overview

Figure 4.4 shows a high-level visualization of our approach to language extension. At the specification level, a DSL is specified through a classical metamodeling process: an abstract syntax defined by a metamodel (an Ecore model [150] in our case), complemented

with an operational semantics which defines both the execution data through an additional metamodel, and the execution functions that are weaved across the metamodel in the form of operations that manipulate the execution data. Any action language can be used for defining the execution functions according to the modeling framework employed. We use ALE (*Action Language for Ecore*), a simple imperative Java-based action language for Ecore that uses static introduction to weave the execution functions in corresponding Ecore classes. Whereas we will not detail further ALE in this part, it is used extensively to evaluate our contributions. Chapter 6 proposes an exhaustive description of ALE features.

The explicit specification of a DSL enables its safe design and reuse. First, it supports specifying both the syntax and semantics in a uniform way while making it independent of the complex implementation details required for supporting advanced reuse, extension, and customization. Second, the DSL specification makes explicit the concept of language as a first-class entity, before it gets diluted at the implementation level. This concept of language is used for checking the safe reuse and manipulation of a given DSL thanks to a dedicated type groups checker [42].

DSL specifications are compiled to Java, following the REVISITOR language implementation pattern, which we discuss in detail in Section 4.3.1. The abstract syntax (possibly extended with runtime data) is compiled to regular Java classes using the EMF compiler from Ecore to Java (gray arrow in Figure 4.4). This standard compilation chain is then seamlessly complemented by compiling ALE semantics specifications to REVISITOR artifacts (black arrow in Figure 4.4). The compilation of a DSL specification is incremental thanks to the REVISITOR pattern; classes representing base syntax or behavior do not have to be recompiled.

4.2.6 Language Composition Overview

Figure 4.5 gives a high-level overview of our approach to language composition. At the specification level, a language module is expressed following a standard metamodeling process. More precisely, we use the same language definition approach as for the extensible language modules approach, presented in the previous section (Section 4.2.5).

As mentioned in Section 4.2.2, language modules may expose a required interface that materializes their requirements towards other modules. To express these interfaces, we rely on the built-in annotation mechanism of Ecore to enable language engineers to add a `«required»` annotation on classes of the metamodel that constitute the required interface. The execution functions woven on such constructs consist of signatures only. Such functions does not have a concrete implementation. The exact implementation of these functions is expected to be defined in the language modules that realize the composition.

The very same metalanguages are used to express how to compose two modules. To specify that a required construct is realized by an external construct in another module, we employ a simple delegation pattern between the two constructs: the required metaclass is extended by a new metaclass, in a new metamodel, that holds a reference towards the

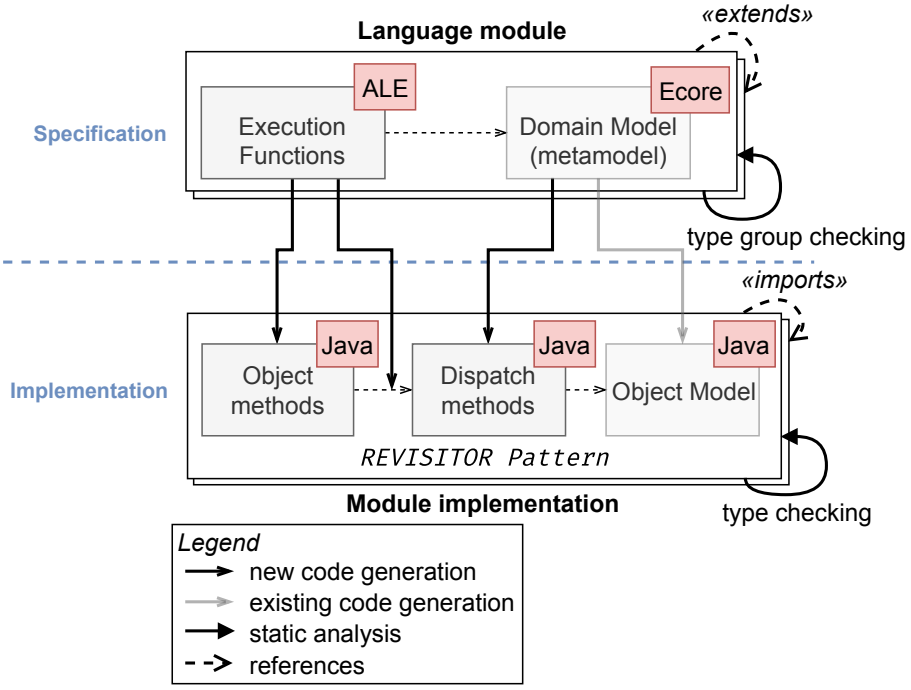


Figure 4.4 – Approach overview of language modules extension.

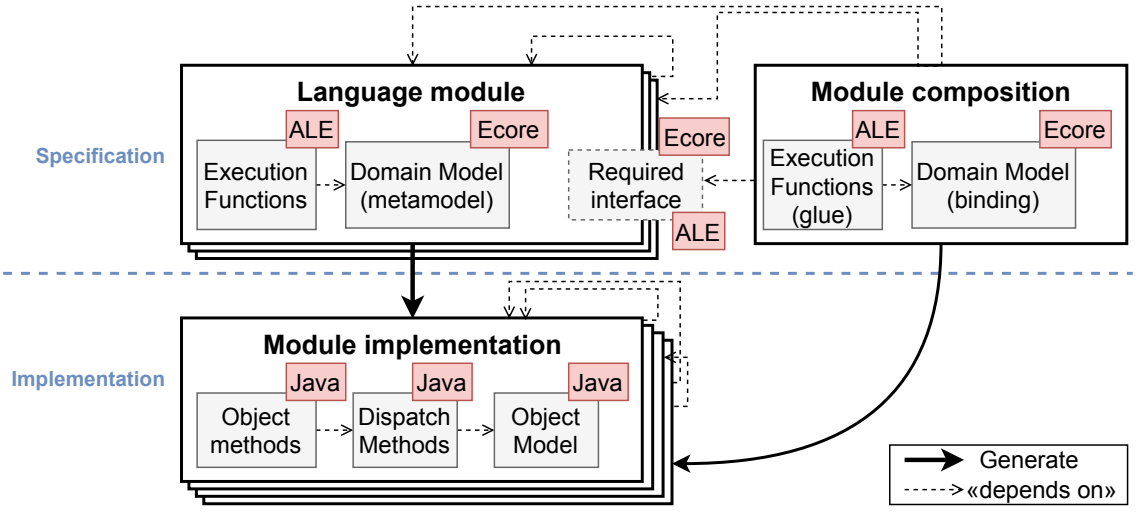


Figure 4.5 – Approach overview of language modules composition.

external construct. A new Ecore metamodel is created to bind all constructs of the required interface of a module in this way. The glue between the signatures of the execution functions of required constructs and the implementation of these execution functions in another module is expressed in ALE itself. The metaclass that holds the delegate reference implements the required signature in ALE; its body expresses how to glue together the two modules semantically. Concretely, this means that the skills required to define new modules are the very same as those required to express how to compose these modules.

Language modules are compiled to Java code using two separate compilers: the built-in Java compiler of EMF that compiles Ecore metamodels to a set of Java interfaces and classes, and our own compiler of the ALE metalanguage that generates a set of Java interfaces following the pattern introduced in Section 4.3. Language modules can be type-checked and compiled independently of each other. The very same compilation chain is reused to compile the specification of the composition of two language modules. From a description of the bindings between two modules in the form of an Ecore metamodel and the glue between their semantics in the form of ALE execution functions, a separate set of Java interfaces that composes the two modules is generated.

In the next section, we dive into the implementation pattern itself and highlight how it enables modular reuse of such language modules.

4.2.7 Modular Language Reuse Discussion

It is important to observe that our approach focuses on the modular reuse of abstract syntaxes and semantics, and does not address the modular reuse of the concrete syntax of software languages.

Many works already address the question of the modular reuse of concrete syntaxes, as presented in Section 3.2.1. In addition, many approaches to the definition and composition of language modules address the question of the concrete syntax [22, 26, 84, 125, 11, 164].

However, those approaches are either outside of the scope of this thesis or do not comply with the requirements we defined to qualify modular approaches to language reuse (Section 4.2.4). For instance, Xtext does not support the independent extensibility of its grammars. Hence, its integration with our approach would limit the possible composition scenarios.

Nonetheless, proposing modular language modules that cover all the concerns of languages while preserving non-functional properties such as modularity or performance in language implementations is an important challenge for the future of SLE.

4.3 Reusable Language Implementation

In this section, we first introduce the REVISITOR implementation pattern (Section 4.3.1). Then, we present how we applied this implementation pattern to modular language extension (Section 4.3.2). Finally, we present how we extended our use of the REVISITOR implementation pattern to modular language composition (Section 4.3.3).

4.3.1 The REVISITOR Pattern

The REVISITOR pattern is a language implementation pattern that reconciles the modular extensibility offered by Object Algebras [124] with the requirements of having an explicit, metamodel-based abstract syntax to describe graph-structured, mutable models. While the REVISITOR pattern itself is independent of a particular programming language, we present it in plain Java 8 code, using interfaces to leverage multiple inheritance. For this section, metamodels are assumed to be defined by plain Java classes; Section 4.3.2 discusses how the pattern can be applied in the context of EMF.

4.3.1.a REVISITOR Interfaces

REVISITOR interfaces (*IRevisitor* in Figure 4.6) are generic abstract factory interfaces declaring factory methods corresponding to AST constructs. Like an Object Algebra interface, a REVISITOR interface declares an extensible mapping from syntactic objects to semantic objects, captured by generic type parameters of the interface. Concrete operations are then defined by implementing the interface, thereby mapping a syntactic structure to a semantic structure which can be used to perform the operation.

An example of the REVISITOR interface for the FSM language of Figure 4.2 is shown in Listing 4.1. A REVISITOR interface defines generic, abstract factory methods for each syntactic concept of the metamodel where each factory method has a single parameter which represents the corresponding concept (the $c_1, ..., c_n$ methods in Figure 4.6). In this case, there are four such methods: for Machine, State, FinalState, and Transition. Each factory method is declared as returning a generic type parameter which will be instantiated by concrete REVISITORs.

In addition to the abstract factory methods, a REVISITOR interface implements concrete methods for dispatching from an actual model object to the corresponding factory method. By convention, these methods are named $, and there is a $-method for every concept in the metamodel. In the FSM example, the dispatching methods for Machine, FinalState, and Transition simply call the respective factory method. In the case of State, however, the $-method uses runtime type-checks to dispatch to the most specific factory method.

Note that the REVISITOR interface is *generic*: whatever the factory methods return is as of yet unspecified. For every concept in the metamodel, there is a corresponding, distinct type parameter representing a possible semantics for that syntactic concept. Inheritance in the metamodel is expressed by additional bounds on the type parameter. For instance, the type parameter `F` is bounded by `S` because FinalState is a subclass of State. This ensures that syntactic subtypes map to semantic subtypes and that the $-methods can return more specific semantic objects for more specific syntactic types.[1]

[1] In the specific case of Java, multiple bounds on a single type parameter are not allowed, which prevents the use of this technique in case of multiple inheritance. We discuss workarounds in the context of EMF in Section 4.3.1.c.

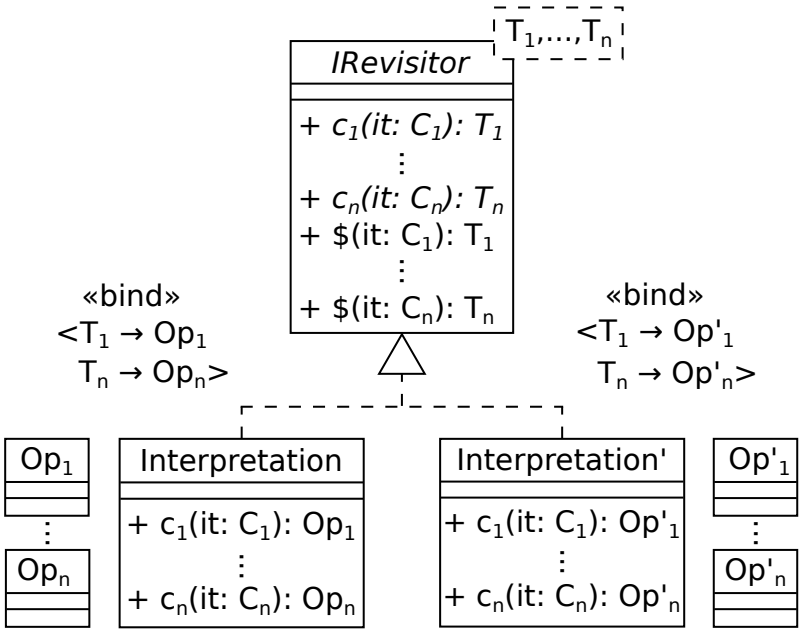


Figure 4.6 – The REVISITOR Pattern maps syntactic objects of types $C_1, ..., C_n$ to semantic objects of types $Op_1, ..., Op_n$. Different implementations of the REVISITOR interface of a language lead to different interpretations. Note that the $-methods are implemented in *IRevisitor* and reused for all interpretations.

4.3.1.b REVISITOR Implementations

A REVISITOR interface defines the basic infrastructure to map a model into some semantics. Concrete semantics of a language is defined by implementing this interface, and explicitly invoking the $-methods when a model element needs to be mapped to its semantic object. Implementing a REVISITOR interface thus defines a case-based mapping from syntactic model objects to corresponding semantic objects, where the mapping is executed lazily, and explicitly, through invocations of the $-methods.

Listing 4.2 shows an excerpt of a concrete REVISITOR that defines a pretty-printer for the FSM language. The `Pr` interface defines the type of semantic denotations that the FSM model will be mapped to ($Op_1, ..., Op_n$ in Figure 4.6). The actual printing semantics is then defined as a concrete interface, binding the type parameters of `FsmAlg` to `Pr`. The default methods override the generic factory methods, returning `Pr` objects (here, using Java 8 closure notation) to print each model element. Note that whenever a factory method navigates the argument model (e.g., `State it`), and requires its corresponding semantics, the $-method is used to obtain it. In Listing 4.2 this is shown on line 6, where the machine's states are printed by first invoking $ on the state elements, and then invoking `print`.

The printing of a machine's states also shows why the type bound on the generic type parameters (cf. `F extends S` in Listing 4.1) is required. The collection of states in a machine abstracts over the difference between ordinary states and final states. As a result,

```
1  interface FsmAlg<M, S, F extends S, T> {
2    M machine(Machine it);
3    S state(State it);
4    F finalState(FinalState it);
5    T transition(Transition it);
6
7    default M $(Machine it)    { return machine(it);    }
8    default F $(FinalState it) { return finalState(it); }
9    default T $(Transition it) { return transition(it); }
10   default S $(State it) {
11     if (it instanceof FinalState) return finalState((FinalState) it);
12     return state(it);
13   }
14 }
```

Listing 4.1 – REVISITOR interface for the FSM language depicted in Figure 4.2

the $-method on State objects needs to return the most general semantic type (i.e., `S` for states). Yet, final states might require specialized semantics, and hence a more specific semantic type. The type bound ensures that this type will indeed be a subtype of the semantic type of ordinary states so that the abstraction over the syntactic type carries over to abstraction over the semantic types. In the example, both State and FinalState are mapped to `Pr`, so the bound is trivially satisfied. However, it would be possible to let the `finalState` method return an object of a type that is more specific than `Pr`; in either case, the `$(s).print()` call on line 6 of Listing 4.2 is valid.

The use of `Pr`-closures in the example is not essential to the approach. If more than one method is needed in the semantic type, separate classes can be defined external to the REVISITOR; the factory methods will simply instantiate them passing a reference to the REVISITOR (in order to be able to call the $-methods) and the actual model element to the constructor.

The following code shows how the print semantics is used on an actual model:

```
Machine fsm = ... // load a model conforming to FSM
Pr p = new PrintFsm(){}.$(fsm);
System.out.println(p.print());
```

The model is first loaded into an object structure conforming to the metamodel (i.e., whose root element is of type `Machine`). The concrete REVISITOR is then instantiated, and the model is passed to $. The result is a `Pr` object which is then printed.

4.3.1.c Multiple Inheritance

The REVISITOR pattern as presented in Section 4.3.1 uses type bounds on type parameters to allow syntactic subclasses to be mapped to semantic subclasses. Unfortunately, some languages (e.g., Java) do not support multiple abstract type bounds on type parameters (`Foo<A, B, C extends A & B>`, for instance, would be rejected). As a result, multiple

```
interface Pr { String print(); }

interface PrintFsm extends FsmAlg<Pr, Pr, Pr, Pr> {
  default Pr machine(Machine it) {
    return () → it.states.stream()
      .map(s → $(s).print())
      .collect(Collectors.joining("\n"));
  }

  default Pr state(State it) { ... /* omitted for brevity */ }
  default Pr finalState(FinalState it) {
    return () → "*" + state(it).print();
  }
  default Pr transition(Transition it) {
    return () → it.event + " ⇒ " + it.tgt.name;
  }
}
```

Listing 4.2 – A REVISITOR implementation for FSM implementing a pretty-printer

inheritance used in the metamodel cannot be directly represented. The workaround in this context is not to introduce one factory method per class in the metamodel, but one per class-superclass pair, returning the semantic type as expected from the context where the $-method is invoked.

As an example, consider a metamodel which contains three concepts, A, B, and C, where C extends both A and B. The generated REVISITOR interface would then be as shown in Listing 4.3. The type parameter `CT` has no type bounds here. Instead, two additional type parameters are used to model the semantics of `C` in the context of a particular parent class. Both $-methods for `A` and `B` include runtime type-checks for model elements of type `C`, and delegate to the specific factory methods, `c_as_a`, and `c_as_b`, respectively.

4.3.2 Modular Extension with REVISITORS

We now discuss how the REVISITOR pattern provides modular and independent extensibility (R1) on both dimensions (syntax and semantics), with incremental compilation (R2) and without requiring anticipation (R3). In particular, we discuss the following extension scenarios: semantic extension (provide a new interpretation of a language), syntactic extension (extend a language with new syntactic concepts), and independent extension (combine two separate languages into one language).

4.3.2.a Semantic Extension

Semantic extension consists of providing a different implementation of the REVISITOR interface of a language. In the example of FSMs, for instance, a different semantics could

```
interface ABC<AT, BT, CT, CT_A extends AT, CT_B extends BT> {
  AT a(A a);
  BT b(B b);
  CT c(C c);
  CT_A c_as_a(C c);
  CT_B c_as_b(C c);

  default AT $(A it) {
    if (it instanceof C) return c_as_a(it);
      return a(it);
  }

  default BT $(B it) {
    if (it instanceof C) return c_as_b(it);
    return b(it);
  }

  default C $(C it) { return c(it); }
}
```

Listing 4.3 – Multiple inheritance in REVISITOR interfaces

be executing FSMs. Listing 4.4 shows the skeleton code of an execution semantics for FSMs.

```
interface St { void step(String ch); }

interface ExecFsm extends FsmAlg<St, St, St, St> {
  default St machine(Machine it) { return ch → { ... }; }
  default St state(State it )    { ... }
  ...
}
```

Listing 4.4 – Executing Finite State Machines

The `St` interface captures the semantic type of each model element; in this case, it represents a simple `step` method that receives a character. Then the `FsmAlg` interface is implemented using `St` to bind all type parameters. The concrete factory methods provide semantic interpretation for each syntactic type. Note that this definition is fully modular: no existing code needs to be changed or duplicated.

4.3.2.b Syntactic Extension

Extending the syntax of a language presupposes that its metamodel is extended with new concepts. Suppose the FSM language is extended with a new kind of transition, called `TimedTransition`, with an additional integer attribute `time`. To support the new

construct in the definition of FSM semantics, the REVISITOR interface `FsmAlg` is extended as `TimedFsmAlg` as shown in Listing 4.5. The new type parameter `TT` (extending the transition parameter `T`) will represent the semantics of timed transitions.

```
1  interface TimedFsmAlg<M, S, F extends S, T, TT extends T>
2      extends FsmAlg<M, S, F, T> {
3
4    TT timedTransition(TimedTransition it);
5
6    default TT $(TimedTransition it) {
7      return timedTransition(it);
8    }
9
10   @Override
11   default T $(Transition it) {
12     if (it instanceof TimedTransition)
13       return timedTransition((TimedTransition) it);
14     return transition(it);
15   }
16 }
```

Listing 4.5 – Extending FsmAlg to support timed transitions

The interface defines a factory method for timed transitions (`timedTransition`) and a corresponding $-method. Furthermore, since the inheritance hierarchy has changed, the $-method for `Transition` is overridden to deal with the new subconcept.

Given this new REVISITOR interface, we can now incrementally define the printing semantics for FSMs containing timed transitions, reusing the existing `PrintFsm` code. This is shown in Listing 4.6: the interface `PrintTimedFsm` extends the `PrintFsm` interface for the existing semantics, and the `TimedFsmAlg` interface to provide semantics for the new construct. The latter is achieved by defining `timedTransition` in terms of the `Pr` interface. Note how the ordinary transition semantics is reused by invoking the `transition` method directly within the body of the closure (Line 4).

```
1  interface PrintTimedFsm extends PrintFsm, TimedFsmAlg<Pr, Pr, Pr, Pr, Pr> {
2    default Pr timedTransition(TimedTransition it) {
3      return () → it.time + "@" +
4        transition(it).print();
5    }
6  }
```

Listing 4.6 – Printing timed transitions

4.3.2.c Independent Extensiblity

The extensibility scenarios presented up to now can all be characterized as forms of *linear* extension: a single abstract or concrete REVISITOR interface is extended or specialized. Independent extensibility allows multiple language components to be extended at once through multiple inheritance.

Listing 4.7 shows skeleton code illustrating independent extensibility in the context of the FSM example. A visual illustration of the extension relations is shown in Figure 4.7. In this case, a new variant of the FSM language is defined that features guarded transitions; this language is captured by the REVISITOR interface `GuardedAlg`. Guard conditions are represented by an independently developed expression language (`ExpAlg`). The evaluation of expression is defined as `EvalExp`.

The `GuardedAlg` then combines both `FsmAlg` and `ExpAlg`, and extends the combination of these two languages with the guarded transition concept (`Guarded`). Additionally, it defines a dispatch method for `Guarded` transitions and overrides the dispatch method for `Transition` because the inheritance hierarchy has changed.

Finally, `ExecGuarded` defines the execution semantics of the combined language, reusing the execution semantics of base FSMs (`ExecFsm`) and the evaluation of expressions (`EvalExp`). The semantics of the new language construct is defined by implementing the `guarded` factory method. Within the closure returned by this method, the $-method is used to obtain the semantics of the guard (inherited from `EvalExp`), and the base `trans` method is reused to obtain ordinary transition behavior after the guard has evaluated to true.

4.3.3 Modular Composition with REVISITORS

In this section, we describe how to derive modular language module implementations from specifications of language modules as described in Section 4.2.6, and exploiting the independent extensibility presented in the previous section (Section 4.3.2.c).

The modular implementation pattern we propose relies on two main ideas that make it intuitive. First, it leverages two well-known concepts of object-oriented programming: inheritance and the delegation pattern. Second, the same pattern and compilation scheme is employed to implement both the language modules themselves and the specification of the

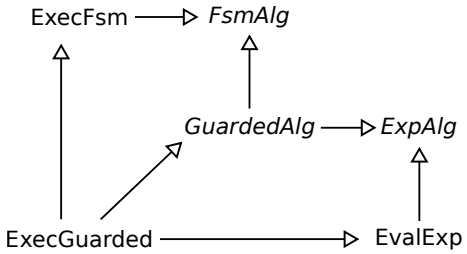


Figure 4.7 – Independent extension of FSMs and expressions, via guarded transitions, reusing existing execution semantics

```
// finite state machine execution
interface ExecFsm extends FsmAlg<St, St, St, St> { ... }

// expression language
interface ExpAlg<E, ...> { ... }

// semantic type for expression evaluation
interface Ev { Object eval(); }

// expression evaluator
interface EvalExp extends ExpAlg<Ev, ...> { ... }

// "metamodel" extension
class Guarded extends Transition {
  Exp guard;
}

// FSM + expression + guarded transitions as glue
interface GuardedAlg<E, ..., M, S, F extends S, T, G extends T>
    extends FsmAlg<M, S, F, T>, ExpAlg<E, ...> {

  G guarded(Guarded it);

  @Override
  default T $(Transition it) { ... }
  default G $(Guarded it)    { ... }
}

// guarded FSM execution reusing ExecFsm and EvalExp
interface ExecGuarded extends GuardedAlg<Ev, ..., St, St, St, St, St>,
    ExecFsm, EvalExp {

  default St guarded(Guarded it) {
    return ch -> {
      Object gv = $(it.guard).eval();
      if (gv.equals(true))
      trans(it).step(ch);
    };
  }
}
```

Listing 4.7 – Independent extensibility: combining FSMs and expressions through guarded transitions.

composition between them.

Our pattern extends the REVISITOR pattern that was used in earlier work to support modular and independent extension of the syntax and semantics of DSLs [104]. In this section, we describe how we extend it to account for required interfaces and go beyond strict extension to support arbitrary composition of language modules. Our extensions retain the desirable properties of the REVISITOR pattern: the syntax and semantics of language modules can be independently extended in a modular and type safe way, without requiring anticipation.

4.3.3.a Language Module Implementation

As we have shown in Section 4.3.2, it is possible to automatically generate a REVISITOR interface from the same metamodel, which specifies an extensible mapping from syntactic objects to semantic objects, captured by generic type parameters of the interface [104].

We extend the REVISITOR implementation pattern to account for `«required»` constructs. We enable language engineers to annotate certain elements of the metamodel with a `«required»` annotation, using the native EMF annotation mechanism [150]. `«required»` classes must be declared abstract, as they cannot be instantiated in the current language module without being bound first. Annotating a class with `«required»` is a simple language module interface documentation which is both understandable by humans, who can quickly understand if a language is fully defined and what are its extension points, and by computers which can exploit them to check interfaces and bindings at the specification level automatically.

A first part of the pattern, the *semantic mapping*, specifies a mapping from metaclasses to abstract execution functions and is implemented by a REVISITOR interface. In a standard REVISITOR interface, each metaclass leads to the introduction of (i) an abstract factory method and (ii) a dispatch method that dynamically dispatches from static metamodel types to the appropriate factory method according to the runtime type of the argument. As the `«required»` classes are not meant to be fully implemented in the current module, the generated REVISITOR includes an (abstract) dispatch method but skips the generation of a factory method for the `«required»` classes. The generation of abstract factory methods is postponed until concrete implementations of `«required»` classes are known, i.e., until the requirements expressed by `«required»` classes are fulfilled by one or several other modules at composition time.

Listing 4.8 depicts an excerpt of the REVISITOR interface generated from the FSM metamodel of Figure 4.3.[2]

The `GFSMRev` REVISITOR interface declares one generic type parameter per class in the metamodel (including the `«required»` ones) and one factory method per non-`«required»` class. Consequently, there is no factory method for `Action` and `Guard`. Finally, the REVISITOR interface declares one dispatch method (named $) per class in the metamodel. The $-methods define a case-based mapping from syntactic constructs to corresponding

[2] In the listings, `...` depicts peripheral code left out for the sake of clarity.

```
interface GFSMRev<M, T, A, ...> {
  M machine(Machine it);
  T transition(Transition it);
  ... // No factory methods for Action and Guard

  default M $(Machine it) { return machine(it); }
  default T $(Transition it) { return transition(it); }
  // Abstract dispatchs
  A $(Action it);
  G $(Guard it);
  ...
}
```

Listing 4.8 – REVISITOR interface for the FSM language module depicted in Figure 4.3

semantic objects, where the mapping is executed lazily, and explicitly, through invocations of the $-methods. As concrete implementations of `Action` are not known yet, its dispatch method is left abstract.

A second part of the pattern, the *semantic interface*, is realized by a set of Java interfaces — one per metaclass in the module — that define the signatures of the execution functions of the constructs included in a module. The signatures are then mapped to the appropriate constructs through the definition of a *concrete semantic mapping*, a Java interface that inherits from the REVISITOR interface and binds every generic type parameter to the corresponding Java interface.

Listing 4.9 presents a pretty-printing semantic interface for the FSM of Figure 4.3. The `IPrint` Java interface defines the signature of a `print()` method which returns a `String`. The `PrintGFSMRev` interface inherits from `GFSMRev` and binds each of its generic type parameters to the `IPrint` interface. So, every construct of the module, as defined in its metamodel, are mapped to the `print()` execution function through invocations of the $-methods. At this point, the whole public interface of the language module is defined, without any concrete implementation of the execution functions yet.

```
interface IPrint { String print(); }
interface PrintGFSMRev extends GFSMRev<IPrint, IPrint, IPrint, ...> {}
```

Listing 4.9 – Semantic interface and semantic mapping for a pretty-printer of the FSM language module depicted in Figure 4.3

Finally, the *semantic implementation* is realized by a Java interface that inherits from the concrete semantic mapping and implements the factory methods to provide the implementation of execution functions. Each implementation is realized by returning instances of the semantic interfaces corresponding to the bindings defined in the semantic mapping.

Listing 4.10 depicts the pretty-printing semantic implementation for the FSM module of Figure 4.3. A new `ImplPrintGFSMRev` interface is defined, extending `PrintGFSMRev` with concrete implementations of the factory methods using anonymous classes that give the semantics of every non-«`required`» construct.

```
interface ImplPrintGFSMRev extends PrintGFSMRev {
  default IPrint machine(Machine it) {
    return () → "machine " + it.name + "\n" +
      it.states.stream().map(s → $(s).print()) + "\n" +
      it.trans.stream().map(t → $(t).print());
  }

  default IPrint trans(Trans it) {
    return () → it.event +
      "[" + $(it.guard).print() + "]" +
      " / " + $(it.action).print();
  }
  ...
}
```

Listing 4.10 – Implementation of a pretty-printer for the FSM language depicted in Figure 4.3

It is important to note that the semantics of every non-«required» construct can already be implemented, even though the concrete syntax and semantics of «required» constructs are not known yet. For instance, in Listing 4.10, printing a transition consists of printing its event, the associated guard, and the associated action. Invoking the $-methods on `Guard` and `Action` returns the semantic interfaces of guards and transitions, which have been mapped to `IPrint` in Listing 4.9 and allow to type-check and compile the FSM module independently. One can observe that Listing 4.10 is very close to the definition of the FSM pretty-printer of Listing 4.2. What distinguishes them is their instantiability. Indeed, in the context of Listing 4.2, the language implementation is complete and can be used directly for the interpretation of FSM models. Conversely, in the context of Listing 4.10, the concrete implementations of `print()` for `Guard` and `Action` is left abstract. Consequently, `ImplPrintGFSMRev` cannot be instantiated and executed in its current state. It must be composed in order to form a fully defined and instantiable language semantics.

4.3.3.b Composition of Language Modules

As mentioned earlier, the composition of two language modules is realized by a language module itself. In this scenario, the metamodel of this new module binds the «required» constructs of a requiring module to the concrete constructs of one or several providing modules. The operational semantics of this new module specifies how the signatures of the execution functions of the «required» constructs are bound to concrete execution functions of the providing modules, possibly with some glue in-between. In this section, we first present the details of the syntactic bindings between constructs. Then, we present the details of the semantic gluing between execution functions.

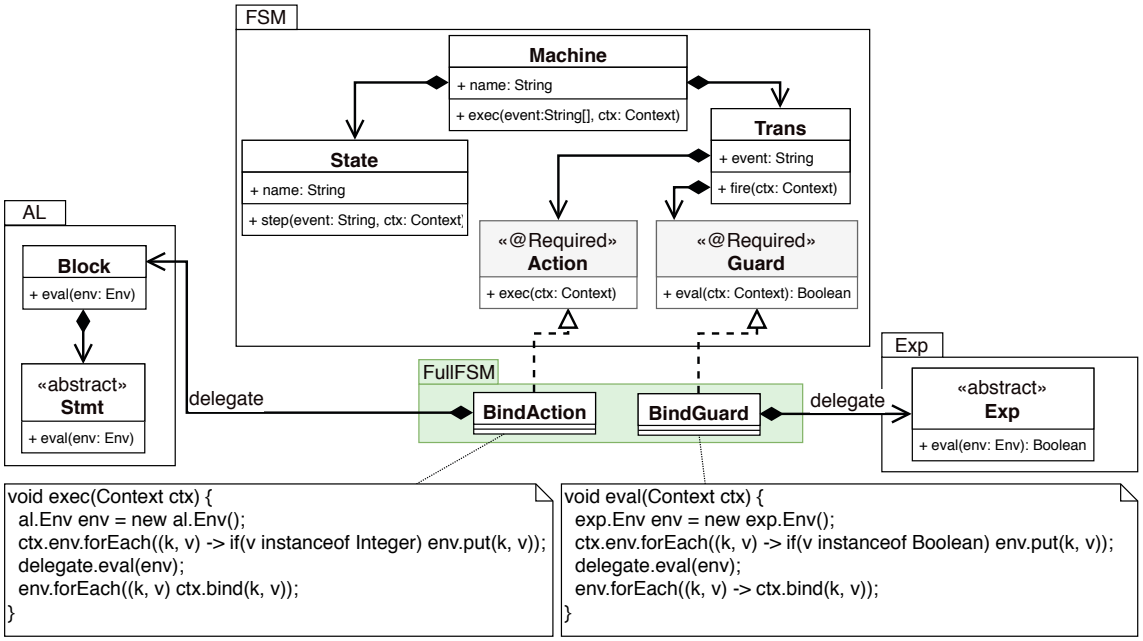


Figure 4.8 – Composing the FSM language module with an action language module and an expression language module. For the sake of conciseness, only excerpts of these modules are depicted here

Metamodel Composition The first step in composing two language modules is to compose the metamodels defining their abstract syntax. Composing the metamodels can be done regardless of the semantics of the composed language modules, i.e., two language modules can be mapped syntactically without having to consider their semantics.

Metamodel composition is realized by reusing the built-in inheritance mechanism of Ecore and the syntactic extension mechanism provided by the REVISITOR implementation pattern. A new metamodel is created containing one *Bind* metaclass per `«required»` construct. Each *Bind* metaclass inherits from a `«required»` construct and holds a single reference `delegate` to the construct that fulfills the interface in the providing module, following the well-known object-oriented delegation pattern. This way, the binding mechanism is modular and does not require any modification in either of the two composed modules.

Figure 4.8 illustrates the composition of the FSM language module to an Action Language module and an Expression Language module. The `FullFSM` metamodel specifies the bindings between constructs of the three modules. Two bindings are defined for the `«required»` classes `Action` and `Guard`, respectively to the `Block` and `Exp` classes, materialized by the `BindAction` and `BindGuard` metaclasses. Following the same generation process as introduced in Section 4.3.3.a, this leads to the generation of a new REVISITOR interface which inherits from the REVISITOR interfaces of all composed language modules and specifies factory and dispatch methods for the *Bind* metaclasses. The `FullFSMRev` interface

depicted in Listing 4.11 is the REVISITOR interface generated from the `FullFSM` metamodel shown in Figure 4.8. As the concrete types of `Action` and `Guard` are now known, the `FullFSMRev` gives concrete implementations for the dispatch methods that were left open in Listing 4.8.

```
interface FullFSMRev< ... , ActionT, ... ,
    BindActionT extends ActionT, ... >
      extends FSMRev< ... , ActionT, ... >,
      ALRev< ... >,
      ExpRev< ... > {

  BindActionT bindAction(BindAction it);
  ...

  default ActionT $(Action it) { return bindAction((BindAction) it); }
  default BindActionT $(BindAction it) { return bindAction(it); }
  ...
}
```

Listing 4.11 – REVISITOR interface generated from the `FullFSM` metamodel depicted in Figure 4.8

Semantic Interface Composition Once «`required`» constructs are bound to concrete constructs syntactically, their semantics must be bridged. In the FSM example, the three modules support variable binding and manipulation. As they have been defined independently, however, the stores that hold variables and their values do not match: the Action Language module uses the `al.Env` store which holds integer variables, the Expression Language module uses the `exp.Env` store which holds Boolean variables, and the FSM language module uses its own `fsm.Ctx`. It is nonetheless essential that the variables declared in the FSM can be manipulated by guards and actions. In the FSM example, the semantic glue thus mainly consists of the translation of variables from one store to the other and the invocation of the appropriate execution functions.

The two boxes at the bottom of Figure 4.8 depict a possible glue between these modules. As shown in the bottom left, executing an action requires to extract the integer variables declared by the FSM from the `fsm.Ctx` store and to provide them to the actions using their own `al.Env` store. Then, invoking the `eval()` execution function of `Block` is done through the `delegate` reference hold by `BindAction`, passing the appropriate store. Finally, the local `fsm.Ctx` store of the FSM is updated back to account for possible updates of the values by the actions. As shown in the bottom right, a similar glue is defined between `Guard` and `Exp`, this time passing the Boolean variables around.

Listing 4.12 depicts how the glue is implemented following our implementation pattern. The glue is specified as the implementation of the semantics of the *Bind* metaclasses introduced in Section 4.3.3.b. The `FullFSMEvalRev` interface inherits from the REVISITOR

interface of the composed module shown in Listing 4.11 and implements the execution functions of `BindAction` and `BindGuard`. These execution functions are the glue itself. For instance, the implementation of the factory method for `BindAction` is the implementation in Java of the glue depicted in Figure 4.8.

```
interface FullFSMEvalRev
    extends FullFSMRev< ... , EvalBindAction, ... >,
      FSMEvalRev,
      ALEvalRev,
      ExpEvalRev {

  default EvalBindAction bindAction(BindAction it) {
    return (ctx) → {
      al.Env env = new al.Env();
      ctx.env.forEach((k, v) → if(v instanceof Integer) env.put(k, v));
      delegate.eval(env);
      env.forEach((k, v) → ctx.bind(k, v));
    }
  }
  ...
}
```

Listing 4.12 – Semantic implementation generated from the glue depicted in Figure 4.8

In conclusion, the composition of two language modules is realized through syntactic bindings and semantic glue. This composition is implemented by a language module itself and follows the exact same implementation pattern as that of a language module. The semantic mappings, semantic interfaces, concrete semantic mappings, and semantic implementations, along with syntactic bindings and semantic glue, can all be written, type-checked, and compiled separately.

4.4 Conclusion on Language Reuse

In this chapter, we propose the REVISITOR language implementation pattern, which brings modular extensibility, composition, and customization to the definition of DSLs. The pattern can be seen as a variant of Object Algebras [124], allowing seamless application in the context of MDE, where explicit abstract syntax structures and mutability of models are prevalent.

The REVISITOR pattern can further be used as a compilation target for high-level specification languages, thus bringing separate compilation to model-based semantics specification. The specification level eases the definition of DSLs in a uniform object-oriented way and offers advanced type checking to ensure safe definition and manipulation of DSLs.

We show how REVISITORS facilitate modular extension of both syntax and semantics

when applied directly in Java, and that our implementation fulfills our requirements: independent extensibility (R1), incremental compilation (R2), and opportunistic reuse (R3).

Additionally, we have presented a modular implementation pattern for the definition and composition of language modules, fulfilling the module encapsulation requirements (R4). Language modules are equipped with well-defined required interfaces that enable encapsulation and information hiding, and make explicit the requirements a module has towards other modules (R5). When equipped with an explicit syntax and semantic interface, modules can be substituted modularly by any alternative module conforming to the same interface (R6). Finally, Language modules can be composed safely and modularly (i.e., with separate compilation and without anticipation), and without having to dive into their internal implementations. Our approach is integrated into ALE, a high-level object-oriented language dedicated to the definition of operational semantics on top of Ecore metamodels (R7).

CHAPTER 5

LANGUAGE PERFORMANCE OPTIMIZATION

In this chapter, we develop our contribution to the optimization of DSLs implementations. First, we introduce the problems related to performance in the context of DSLs (Section 5.1). Then, we present some background (Section 5.2), complementing the general background of Chapter 2, and we discuss the current approach to DSLs design and implementation in the context of language performance (Section 5.3). Next, we present our approach to the automatic optimization of DSLs (Section 5.4). Finally, we draw some conclusions on the benefits of our contribution (Section 5.5).

5.1 Introduction to Language Performances

The main objective of language workbenches is to support domain experts by reifying concepts dedicated to a given application domain to ease the development of future complex systems in this particular domain. They rely on generative and generic approaches to produce language services implementation from language specification. By their very nature, such approaches hamper the incorporation of language-specific optimizations, making the resulting language runtimes much less efficient than the optimized runtimes of general-purpose languages (e.g., just-in-time (JIT) compilation in the current JDK or V8 JS engine). Indeed, DSL runtime generators apply the same generic patterns for the generated code of every DSL. Different generators may apply different patterns, but a given generator always applies the same patterns, which prevents specific optimizations tailored to the specificities of a particular application domain or execution environment. For instance, from any given metamodel, EMF always derives visitor-like Java classes based on runtime-type inspection.

Recently, various execution frameworks have been proposed to support the definition of languages over the JVM and assist in the generation and optimization of interpreters based on JIT compilation – a process that transforms frequently used interpreted code pieces to machine code during execution. Truffle [172] relies on the Partial Evaluation capabilities provided by the GraalVM [128] to realize such optimizations. Truffle offers facilities to complement an initial DSL interpreter implementation with patterns and annotations to benefit from specific runtime optimizations. Performance gains reported in the literature are significant [177]. However, efficiently using these frameworks requires strong expertise in language development and the intricacies of the framework itself, which industrial DSL designers often lack.

In addition, we do so without breaking the compatibility with other tools surrounding the DSLs (e.g., editors, debuggers). We introduce a systematic approach to generate optimized Truffle-based language interpreters from model-based DSLs specifications automatically, inducing a complementary speedup on top of language interpreters based on the Interpreter pattern. Our systematic approach exploits high-level information contained in language specifications to drive the application of Truffle-based optimizations. We also propose an implementation of our approach integrated with the compilation chain of EMF, enabling its application to many already existing and future DSLs.

In summary, we propose and evaluate an approach to the automatic optimization of model-based interpreter performance.

We evaluate our approach on a representative set of four languages and eight conforming programs (from programming languages to modeling languages to "end-user" languages, from arithmetic-intensive to structure-intensive, from recursive style to iterative style). Our experiments demonstrate the benefits of our approach in all cases. The average speedup is of x1.14, ranging from x1.07 to x1.26.

To summarize, the approach proposed in this chapter enables language designers to automatically obtain efficient language interpreters while remaining oblivious of the technical details of the interpreter optimizations.

5.2 Background

In this section, we complement the background on language engineering presented in Chapter 2 with notions specific to language runtime and performance. We first introduce Graal and the Graal Virtual Machine (GraalVM) in Section 5.2.1, and the Truffle language implementation framework in Section 5.2.2.

5.2.1 Graal and GraalVM

GraalVM is a Java Virtual Machine (JVM) resulting from an internal project of Oracle [176] and has shown promising performance improvement results. GraalVM is a universal virtual machine targeting the execution of arbitrary languages (e.g., JavaScript, Python, Ruby, R, or LLVM). It includes a new high-performance JIT compiler, called Graal, which produces native code from Java bytecode. It is used as a replacement for the JIT of the HotSpot VM in GraalVM. Graal is also used as an ahead-of-time compiler by the Substrate VM[1], allowing the compilation of Java bytecode to native machine code outside of a virtual machine.

[1]Substrate VM documentation: https://www.graalvm.org/docs/reference-manual/aot-compilation/

5.2.2 Truffle

Truffle [172] is a framework designed to ease the development of efficient interpreters on the JVM. Several works demonstrate the speedup offered by Truffle-based interpreters [148, 109]. This makes it interesting to study in the context of DSL performance. Truffle provides a set of classes, annotations, and built-in operations in order to produce efficient Java implementations of interpreters following the Interpreter pattern. The Interpreter pattern is additionally decorated with annotations that assist in the definition of efficient language implementations.

At runtime, Truffle relies on Partial Evaluation [108, 63]. This consists of the combination of an interpreter with its program to produce an optimized interpreter, specialized for this given program. While the optimizations are processed by Graal, Truffle provides the expressiveness to define language-level information that assists Graal to apply language-specific optimizations.

In our context, Partial Evaluation works by combining a method of an interpreter with data (i.e., parts of a program) to produce an optimized Graal Intermediate Representation (IR). The Partial Evaluation process allows the application of various optimizations such as constant folding, indirect to direct call substitution, or dead code elimination.

During Partial Evaluation, Truffle can make optimistic assumptions (e.g., a variable is never null, a method always returns true), and propagate such decisions (e.g., removing an unreachable `else` branch) in the resulting optimized machine code. If runtime data later contradict future executions of the optimized code (e.g., the variable is finally not-null), Truffle can *deoptimize* the code and goes back to using the interpreter version of the code.

Without constraints, Truffle explores the execution graph eagerly during Partial Evaluation. Consequently, this process might lead to code explosion [177], failing due to the production of too large specialized interpreters (i.e., too large to be compiled). This is why Truffle requires the definition of explicit boundaries in order to prevent such undesirable behaviors. Würthinger et al. [177] shared their experience building interpreters with Truffle. They tried to define boundaries automatically but did not find a suitable automated solution in the context of the abstractions proposed by Java.

5.3 DSL Design and Implementation

As presented in Section 2.2 the definition of a DSL encompasses the definition of its abstract syntax and semantics. The abstract syntax specifies the domain concepts and their relations. In the modeling world, it is typically defined by a metamodel. Object-Oriented formalisms such as Ecore, represent language concepts as a set of metaclasses and their relations. The semantics of a DSL assign meaning to its constructs. In order to support the operational execution of the conforming models, it is typically defined by an interpreter. It implements its operational semantics in the form of a transition function over model states. In the modeling world, it is defined using an action language that extends the language concepts with operations. In this chapter, we use ALE, presented in Chapter 6.

The operational semantics defines the evolution of the state of the execution of a program. This evolution is realized by the modification of the model by the operational semantics at runtime. The set of concepts modified during the execution is called the *Execution metamodel* [18].

One solution to make metamodels and operational semantics executable is to compile them to general-purpose languages. In this context, their implementations follow well-defined implementation patterns [124, 160, 53, 178, 2], the most common being the Interpreter and Visitor patterns [64].

While functionally equivalent, both patterns offer different advantages, notably regarding their modularity, but also regarding their performances. By default, EMF provides the generation of a variation of the Visitor pattern, named the Switch pattern. Consequently, this implementation pattern is found in EMF-based language implementations and is used as a reference in this chapter.

In the remainder of this chapter, we study our approach to improve further the performances of the implementation of languages specified using model-oriented language specifications, as presented in this section.

5.4 Automatic Generation of Truffle-based Interpreters

This section presents a general overview of our approach and presents the optimizations that can be derived from the abstraction provided by the metalanguage in the context of the approach to language specification presented in the previous section.

Our approach and its context are illustrated in Figure 5.1. The first column presents the state of practice of language interpreter implementation using the standard EMF Switch pattern for the language interpreter implementations and uses HotSpot VM for the program's execution. Then, the second column presents a solution that substitutes the Switch for the Interpreter implementation pattern and uses GraalVM for the program's execution. This column exploits state of the practice solutions exclusively but already provides performance speedup and helps to position our approach. Finally, the third column presents our contribution and introduces a new EMF to Java compiler, allowing the automated introduction of Truffle optimizations while staying compatible with existing EMF implementation patterns. This way, language engineers benefit from performance optimization allowed by Truffle without having to be exposed to its technical details.

In Section 5.4.1, we present some preliminary results that help understand the scope of our approach, corresponding to the transition from the first to the second column. Then, Section 5.4.2 presents the metamodels generation while complying with the constraints of Truffle, and Section 5.4.3 presents the automatic introduction of Truffle boundaries. Finally, Section 5.4.4 details the aspects of Truffle that are not included in our approach.

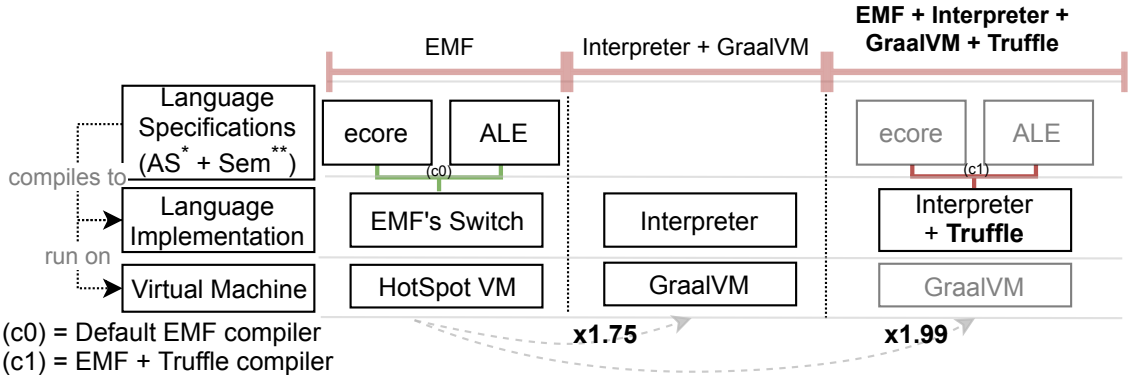


Figure 5.1 – General overview of the proposed approach. Decimal numbers prefixed with an x represent performance speedups. * Abstract Syntax. ** Semantics.

5.4.1 Preliminary Results

The choice of runtime and implementation pattern for model-based DSLs is composed of HotSpot VM and the Switch implementation pattern. As we target GraalVM and the Interpreter implementation pattern, we first investigate the impact of the transition to the latter. Even if this is not strictly part of our contribution, it is still useful to have an estimation of the expected speedup of such design decisions.

Table 5.1 summarizes the performance speedups observed from our benchmarks on several languages and programs. The speedups presented here are computed from the average speedups of the programs between executions using different choices of virtual machine and implementation patterns. We refer the reader to Section 8.2 for an in-depth explanation of our benchmark methodology to better understand how these numbers are computed. The results of our measurements show the impact of the transitions from HotSpot VM to GraalVM with a speedup of x1.56, from the Switch to Interpreter implementation pattern with a speedup of x1.23, and both combined with a speedup of x1.75. We can observe that each transition, independently, is very beneficial, and even more when combined.

5.4.2 Truffle-Compliant Object Model Implementation

Starting from a language implementation based on the Interpreter pattern and running on GraalVM, the next challenge is to translate Ecore metamodels to an object model, in the

Table 5.1 – Average speedups resulting from the transition from HotSpot VM to GraalVM and from the Switch implementation pattern to the Interpreter implementation pattern.

	HotSpot VM	GraalVM
Switch pattern	x1.00	x1.65
Interpreter pattern	x1.23	x1.75

form of a set of Java classes that can be efficiently optimized by Truffle. Truffle expects Java classes to form an immutable AST, whereas Ecore defines graph-based and mutable set of classes. Consequently, the translation from Ecore is not straightforward, and multiple steps are needed to produce an efficient Truffle compliant implementation.

The first step is the discrimination between the immutable metaclasses, that can be part of Truffle object models, and mutable metaclasses of the Execution metamodel, that cannot. Metaclasses that match this definition can be compiled into Truffle nodes, whereas the others are compiled to standard Java classes.

We proceed to this classification by analysis of the ALE specifications. Metaclasses instantiated using the **create()** operation (i.e., possibly created at runtime) are part of the Execution metamodel, whereas the other metaclasses are part of the Abstract Syntax. Only metaclasses of the later are compiled as Truffle nodes.

Truffle nodes are identified by their inheritance to the Truffle `Node` class. We duplicated the usual EMF class hierarchy in order to create a Truffle specific hierarchy by introducing the `Node` class at the top of the hierarchy. Classes of the Execution metamodel inherit from the standard EMF hierarchy, whereas classes of the Abstract Syntax inherit from the Truffle specific hierarchy.

The second step is the identification of the metaclasses' references that can be translated into Truffle parent-child relation, which defines the tree-shaped hierarchy in the object model. We identified the following constraints i) Metaclasses' references must be containment references; ii) the references must take place between metaclasses that can be translated into Truffle nodes; iii) the reference cannot be mutated during the language execution. References that conform to these three constraints are promoted as parent-child relations.

The identification of the containments is realized by a straightforward analysis of the metamodel. We presented above how Truffle nodes are identified. Finally, the mutability analysis is realized by analyzing the occurrences of field modification operations (e.g., call to setters or modification of collections). If a relation between two metaclasses conforms to the three constraints above, the compiler introduces a parent-child relation.

If the reference has an upper multiplicity of one, parent-child relations are realized by the annotation of the field with the `@Child` annotation. When its upper multiplicity is greater than one, the `@Children` annotation is added to the field, but this introduces an additional constraint on the generated code. Indeed, Truffle constrains the fields annotated with `@Children` to be a Java array, instead of the usual EMF `EList`. Additionally, in order to preserve the compatibility with EMF model loading (i.e., one of EMF's tool support), all the fields derived from references with an upper multiplicity greater than one must be of type `EList`. We satisfy those contradictory constraints by introducing a new field, named after the original reference and suffixed with `Arr`, and typed by an array of elements of the same type as the original reference.

Listing 5.1 shows an example of the resulting compilation for a `Block` class with a `statements` field containing `Statement` objects. This array is initialized the first time one of the methods declared in ALE is called (Line 14), by copying the element

```
1  @NodeInfo(description="Block")
2  class Block extends Node {
3    private List<Statement> statements;
4
5    @Children
6    private Statement[] statementsArr;
7
8    public List<Statement> getStatements() {
9      if(statements == null) statements = new ArrayList<>();
10     return statements;
11   }
12
13   public void execute() {
14     if(this.statementsArr == null) {
15       CompilerDirectives.transferToInterpreterAndInvalidate();
16       if(statements != null) statementsArr = statements.toArray(Statement[0]);
17       else statementsArr = new Statement[] {};
18     }
19     // ... statementsArr is used whenever statements is used
20     // in the semantics specification.
21   }
22 }
```

Listing 5.1 – Integration of the Children annotation on a Block statement.

from the list to the array (lines 16 and 17). *CompilerDirectives.transferToInterpreterAndInvalidate()* (Line 15) warns Truffle to return to the Java bytecode interpreter because a JIT-compiled machine code would be deprecated (i.e., deoptimized), in case of early Truffle Partial Evaluation.

5.4.3 Truffle Boundaries

We explained in Section 5.2.2 the challenge of Truffle boundaries identification. Placing relevant Truffle boundaries is crucial to obtain interesting performance speedups. We can take advantage of our approach based on metalanguages with domain-specific expressiveness and a generic compilation scheme, allowing the safe reification of boundaries in the compiler.

Consequently, we can identify exhaustively the places where boundaries are relevant without the need for advanced heuristics or static analysis, allowing the safe generation of interpreters without risks of Partial Evaluation failure during program interpretation. For instance, the Java sources produced by EMF introduce involved proxy mechanisms to guarantee models consistency (e.g., bidirectional reference, where referential integrity is required), which leads to code explosion when partially evaluated. Hence, they are placed behind Truffle boundaries.

Truffle boundaries are defined by annotating methods with the `@TruffleBoundary`

annotation. Boundaries are placed on every method that is not directly derived from ALE specifications. In other words, we isolate the code directly derived from ALE specification, from the code that implicitly supports it. For instance, the call to the classes of the library supporting EMF, or the code indirectly called from it (e.g., the operations of the reflective API of the classes) are placed behind boundaries.

In practice, the code directly derived from ALE specifications is compiled to simple operations that are not subject to code explosion (e.g., accessors, variable affectation, Java operators) and isolated from code subject to code explosion.

5.4.4 Discussion of Additional Truffle Optimizations

For the approach presented in this chapter, we set the constraint to exploit existing abstractions of language specifications and to preserve the compatibility with the tool support of languages. This section discusses possible performance improvement that could not be considered given such constraints. We first illustrate this idea by presenting the Polymorphic Inline Cache optimization (Section 5.4.4.a), before discussing other optimizations (Section 5.4.4.b).

5.4.4.a Polymorphic Inline Cache

Using Truffle allows the definition of Polymorphic Inline Cache (PIC) [73]. PIC is a language optimization historically implemented in the Smalltalk language [45] that aims at optimizing dynamically the performance of the call sites frequently dispatched to different methods. Würthinger et al. [178] present the use of PICs for the optimization of Truffle-based language interpreter implementations.

The introduction of the PIC optimization raises the challenge of the automatic identification of the call site that would benefit from such optimization. Indeed, badly placed, PICs can be detrimental to the performances of programs.

In practice, the identification of the methods that benefit from the introduction of PIC optimizations in language implementations is challenging and can even lead to slowdown or runtime errors if done wrong. We tried to identify the relevant uses of PICs in the

```
1 class LogService {
2   @TruffleBoundary
3   static void log(Object self, String level) {
4     if (level.equals("INFO"))
5       Logger.info(self);
6     ...
7   }
8 }
```

Listing 5.2 – Extract of an ALE service. Methods are annotated with `@TruffleBoundary` in order to anticipate Partial Evaluation issues at runtime.

implementation of languages by benchmarking a large sampling of usage. To do so, we introduced a *dispatch* keyword in ALE, that can be used as a method declaration prefix by language engineers to define the introduction of PICs explicitly.

We proceeded by automatic mutation of the MiniJava object-oriented language presented in Section 8.2.1 to introduce the dispatch keyword on random method declarations. The results of our experiment are available on the companion webpage[2]. The conclusions of our experiments clearly show that the use of PICs has a strong influence on the performance of the language but did not permit us to infer actionable rules to automate the placement of the PICs.

The companion webpage[3] presents additional technical details on the use of Truffle on the implementation of the Polymorphic Inline Cache.

5.4.4.b Other Optimizations

Truffle offers a profiling library, allowing the fine-tuning of the interpreter implementation by the introduction of runtime state monitoring at relevant places (e.g., by monitoring the condition of an `if` statement). Placing those profiling probes is highly context-sensitive and requires knowledge that goes beyond the abstractions available in operational semantics specifications. Consequently, using the profiling capabilities of Truffle is outside the scope of our approach.

Truffle also provides loop unrolling capabilities, assisted by specific annotations and classes. To properly exploit loop unrolling requires to be able to estimate the size of the content of a loop. Since programs of DSLs are very diverse in shapes and sizes, it leads to challenges similar to the one raised for the profiling library.

Truffle provides a `Frame` object, that assists in the definition of stack located variables, instead of the heap. `Frame` objects are non-trivial to manipulate as they are sensitive to escape analysis, and cannot, among others, be assigned to fields or be type-casted. The choice of the runtime data that is beneficial to move into frames is context-specific. Consequently, it falls into the same limitations as the optimizations presented above.

Finally, Truffle support method specialization. This technique is the foundation of various Truffle implementation patterns (e.g., type boxing). But it requires to define multiple methods (e.g., $\texttt{equal}_0\texttt{(int,int)}$, $\texttt{equal}_1\texttt{(String,String)}$) for a single operation (e.g., here, the equality of the values returned by two child nodes). The relevant method is called by inspection of the type of the value returned by the child nodes. This multiplication of methods breaks the public interface of the classes, hence making it possibly incompatible with the surrounding tool support.

[2]Companion webpage: https://manuelleduc.github.io/ecmfa-2020/#automated-feature-selection

[3]Companion webpage: https://manuelleduc.github.io/ecmfa-2020/#polymorphic-inline-cache-implementation

5.5 Conclusion on Language Performance Optimization

In this chapter, we propose an optimized alternative to the standard execution framework for EMF-based interpreted DSLs. The optimized interpreter is obtained by using more efficient implementation patterns and virtual machines and leveraging information obtained from higher levels of abstraction provided by a model-based approach to automatically incorporate Truffle optimizations. Hence, we propose an approach that both allows the introduction of *Language-Specific optimizations* while remaining *Application Domain Agnostic*. In addition, we preserve the tool support that accompanies DSLs. In other word, language engineers can benefit from improved language performance but are not required to have knowledge of the underlying intricacies of the language performance optimization (i.e., the integration of Truffle)

However, we also emphasize the complexity introduced by the use of Truffle. To address this challenge, we leverage on an approach to language implementations based on high-level metalanguages, dedicated to the specification of important aspects of a language (i.e., abstract syntax, and semantics). This approach is complemented with the identification of a useful set of four general-purpose Truffle based optimizations. We show how the expressiveness of our metalanguages is useful to automatically compile an optimize language implementation, guided by the metalanguages definitions. Our approach to language implementation is based on mainstream, high-level object-oriented languages dedicated to the definition of languages.

The implementation details and the concrete speedup results of our approach are presented in Chapter 8.

PART III

Implementation and Evaluation

CHAPTER 6

TECHNICAL BACKGROUND

In this chapter, we introduce the technical background required for the understanding of the implementation and evaluation of our approaches. First, we introduce the ALE *metalanguage (Section 6.1). We use* ALE *for the specification of the semantics of the languages developed in the context of our evaluations. Furthermore, we later extend* ALE *with two compilers, dedicated to the challenges of this thesis (Chapters 7 and 8). Then, we present our environment dedicated to the definition of repeatable performance benchmarks (Section 6.2).*

6.1 Introducing the ALE Metalanguage

ALE is an existing interpreted metalanguage that is part of the GEMOC Studio and integrates seamlessly with the EMF ecosystem. In particular, it relies on Ecore for defining the abstract syntax of language modules in the form of a metamodel and allows the definition of operations on top of metaclasses. The interoperability with EMF enables language engineers to benefit from other tools of the ecosystem, such as graphical editors implemented with Xtext or Sirius. In practice, ALE is an alternative to other approaches to the definition of semantics in the EMF ecosystem (e.g., Kermeta, Xtend), with a particular focus on modularity and composition.

ALE is close the other action languages of the Eclipse ecosystem such as the Epsilon Object Language (EOL) [91], but can be differentiated by its open class mechanism. This open class mechanism is inspired by Kermeta [79] and follows the same *open-class* principle [32]. Indeed, ALE allows to "re-open" metaclasses of a metamodel to weave operational semantics as a set of methods. Method bodies are written in a superset of AQL (Acceleo Query Language)[1], extended with control flow operations (e.g., if, while), and variable declaration and assignment. AQL is itself a dialect of OCL [126].

As an illustrative example, Listing 6.1 depicts the definition, using ALE, of the `fire()` method giving the semantics of a transition for the FSM language depicted in Figure 4.3. The `open class` keyword re-opens the `Transition` metaclass, to weave the `fire` method into it. The `Transition` metaclass is imported from the `fsm.ecore` file using the `import ecore` keywords (e.g., Line 3). In the same way, ALE files are imported using the `import ale` keywords.

ALE allows the import of Java-based operations with the `use` keyword. This import mechanism is based on the extension method mechanism that allows adding new methods

[1] Acceleo Query Language website: https://www.eclipse.org/acceleo/documentation/aql.html

```
1  behavior evalfsm;
2
3  import ecore "fsm.ecore";
4
5  use LogService;
6
7  ...
8  open class Transition {
9    def void fire(Context ctx) {
10     if(not self.guard.eval(ctx)) {
11       'Unsatisfied guard'.log();
12     } else {
13       self.action.run(ctx);
14       ctx.current := self.tgt;
15     }
16   }
17 }
18 ...
```

Listing 6.1 – Definition of the fire method in the Transition class of the FSM language in ALE.

to existing types without modifying them. This mechanism can be found in Xtend[2], for instance. In our example, the LogService class (Line 5) presented in Listing 6.2 is imported, and its `log` method is added to the scope. The `log` method is called Line 11 and prints a warning in the standard output if the evaluation of the guard evaluates to false.

The `self` keyword is equivalent to `this` in Java and refers to the currently re-opened metaclass. The remainder of the statement and expressions used is the body follows a standard object-oriented expressiveness.

```
1 class LogService {
2   static void log(Object log) {
3     System.out.println(log);
4   }
5 }
```

Listing 6.2 – LogService class in Java.

Complementarily, we present how ALE is used for the specification of modular language semantics. Listing 6.3 presents a printing semantics defined using ALE for the FSM language presented in Figure 4.2. It shows the ALE specification equivalent to the Java code of Listing 4.2.

Listing 6.4 presents an example of language extension. First `timed.ecore` extends

[2]Xtend extension method documentation: https://www.eclipse.org/xtend/documentation/202_xtend_classes_members.html#extension-methods

```
behavior printfsm;

import ecore "fsm.ecore";

open class Machine {
  def String print() {
    String ret := "";
    for (s in self.states)
      ret := ret + s.print();
    result := ret;
  }
}

open class State {
  def String print() {
    result := ...;
  }
}

open class FinalState { // extends State implicitly
  def String print() {
    result := "*" + super.print();
  }
}

open class Transition {
  def String print() {
    result := self.event + " ⇒ " + self.tgt.name;
  }
}
```

Listing 6.3 – FSM pretty printer implemented with ALE

`fsm.ecore` and introduces the `TimedTransition` metaclass. `TimedTransition` inherits from `Transition`. Then, the existing `printfsm` ALE specification is imported and `TimedTransition` is re-opened to weave a new print operation that specifies the semantics of the print operation in the context of a timed transition. Doing so, we modularly extend the syntax and the semantics of the original FSM language to define a Timed FSM language.

Finally, Listing 6.5 presents the definition of a Guarded FSM language by the modular composition of the FSM language with an expression language, used for the definition of the guards. Using ALE, language composition is conceptually close to language extension, but involves the import of multiple existing language specifications. In this context of our example, a new `guarded.ecore` metamodel is introduced, that reuses the metamodels of the FSM and expression languages. This example follows the same scenario as the independent extensibility example presented in Listing 4.7. A `Guarded` metaclass is introduced that inherits from the `Transition` metaclass from the FSM language, but also holds a reference — named `guard` — to the `Exp` metaclass from the metamodel

```
behavior printtimed;

import ecore "timed.ecore";
import ale printfsm;

open class TimedTransition { // extends Transition implicitly
  def String print() {
    result := self.time + "@" + super.print();
  }
}
```

Listing 6.4 – Printing timed transitions in ALE

of the expression language. Then, the new `guarded.ecore` metamodel and the ALE specifications of the execution semantics of the two reused languages are imported. Finally, the `Guarded` metaclass is re-opened to weave the `fire` method that defines the semantics of the step operation of the guarded transition.

```
behavior execguarded;

import ecore "guarded.ecore";
import ale execfsm;
import ale evalexp;

open class Guarded { // extends Transition implicitly
  def void step(String ch) {
    if (self.guard.eval()) {
      super.step(ch);
    }
  }
}
```

Listing 6.5 – Executing guarded transitions in ALE

The compilation of an ALE specification to reusable language modules is presented in Chapter 7. First, the compilation of ALE specifications to the reusable REVISITOR pattern is presented in Section 7.1. Then, the evaluation of our approach for the reuse of language specifications, by extension and composition are respectively presented in Section 7.2 and Section 7.3.

The compilation of ALE specifications to Java implementations optimized for runtime performances is presented in Chapter 8. The technical details and evaluation of this compiler are presented respectively in Section 8.1 and Section 8.2.

6.2 Benchmarking Setup

Validating our approaches requires to perform performance measurements. However, measuring performances of programs is a non trivial task that requires the definition of dedicated experimental environments.

In this section, we present the benchmarking environment used for the evaluation of our approaches. The methodology presented below aims at producing repeatable performance measurement of language interpreter executions [65]. We qualify the performance by using the steady-state performance. In other words, we evaluate the performance of the programs once it has reached a stable execution state. This is representative of the execution of DSL interpreters, which are long running programs (i.e., more than a few milliseconds), and are not expected to have fast startup times.

All the benchmarks presented below are executed on a dedicated desktop running on Debian 9, with 15Gb of RAM and an Intel(R) Xeon(R) W-2104 CPU (Quad Core - 3.20GHz).

Besides, we use JMH v1.21[3] to run our experiments. JMH is a Java framework that mitigates the nondeterministic behaviors inherent to the JVM internals (e.g., dead code elimination or constant folding). JMH is primarily developed from the definition of micro-benchmark (i.e., to measure programs that have an execution time of the order of magnitude of milliseconds), but it is also relevant for the benchmarking of programs that have larger execution time (i.e., execution time of the order of magnitude of seconds).

Additionally, we execute our benchmarks using Krun [10]. Krun is a framework that assists in the definition of repeatable benchmarks. For instance, Krun restarts the benchmarking computer between each new measurement, to avoid the influence of pre-cached data on the program's execution. Krun also checks and sets various hardware settings known to influence the repeatability of measurements. For instance, Krun fixes the CPU frequency and checks the CPU temperature at the beginning of each measurement. With this setup, we mitigate the nondeterministic behavior at the hardware, system, and virtual machine levels, improving the repeatability of our benchmarks [82].

[3]Java Microbenchmark Harness (JMH): https://openjdk.java.net/projects/code-tools/jmh/

CHAPTER 7

ALE COMPILER FOR LANGUAGE REUSE

In this chapter, we present the technical details and the implementation and evaluation of our approach to language reuse (Chapter 4) using the ALE *metalanguage. We first present the implementation details of the compiler we developed in order to validate our approach to language reuse (Section 7.1). Then, we present two evaluations of the reuse capabilities of our approach, first by extension (Section 7.2), then by composition (Section 7.3). Finally, we draw some conclusions on the modular reuse of languages (Section 7.4).*

7.1 The REVISITOR compiler

In this section, we present our compiler from ALE specifications to Java implementation, following the REVISITOR implementation pattern. Listing 7.1 presents this compiler, implemented using Xtend. The complete source code of the compiler is available online[1].

Xtend provides template expressions, surrounded by triple-quotes (`'''`). Two main control operations are available, FOR/ENDFOR loops and IF/ELSE/ENDIF conditions. FOR loops are parameterized by an iterator and three parameters BEFORE, SEPARATOR, and AFTER, respectively inserted at the start of the produced string, between each iteration and at the end of the produced string. The template placed between the FOR and ENDFOR operators is parameterized by the iterator.

IF blocks are idiomatic and return the value of one block according to the result of the evaluation of the IF expression.

The `compile` method takes an `EPackage` class in parameter and produces a REVISITOR Java interface. Lines 16 to 17 deal with the generation of the REVISITOR interface generic types, and their binding with inherited REVISITOR interfaces. Each directly referenced `EPackage` produces an inheritance relation. One generic type is produced by `EClass` directly or transitively referenced. Lines 18 to 20 generate one factory method per concrete `EClass` of the `EPackage`.

Finally, the rest of the generation produces one `$-method` by `EClass`. If the current `EClass` is annotated as `«required»`, the method is left abstract, and no implementation is provided. Otherwise, each `$-method` inspect the dynamic type of the `it` parameter, eventually cast the parameter and pass it to the corresponding factory method, found either directly in the current REVISITOR interface, or one of its parents.

[1] The ALE compiler on github: https://github.com/manuelleduc/ale-lang/

```
/*
Omited for undestandability and conciseness:
- Multiple inheritance;
- naming conflicts (e.g., 2 classes with the same name in separate packages);
- generic types declaration order.
*/

...
def compile(EPackage t) {
  val ⊤ = true
  val ⊥ = false

  '''
  package «t.name»;

  interface «t.rn»«t.gts(⊤)»
    «FOR p: t.rps BEFORE='extends' SEPARATOR=','»«p.rn»«p.gts(⊥)»«ENDFOR» {
    «FOR c: t.alc.filter[!abstract]»
    «c.gt(⊥)» «c.name»(«c.name» it);
    «ENDFOR»

    «FOR c: t.acs»
      «IF c.rq»
        «c.gt(⊥)» $(«c.name» it);
      «ELSE»
        default «c.get(⊥)»(«c.name» it) {
          «FOR sc: c.ascs»
          if (it.getClass() == «sc.name».class)
            return «sc.name»((«sc.name») it);
          «ENDFOR»
          return null;
        }
      «ENDIF»
    «ENDFOR»
  }
  '''
}

 // revisitor name.
def rn(EClass c) '''«c.name»Revisitor'''

// packages directly referenced from p.
def List<EPackage> rps(EPackage p) { ... }

// generic type declaration name generation.
def gts(EPackage p, boolean ext)
  '''
  «FOR c: p.acs BEFORE='<' SEPARATOR=',' AFTER='>'»«c.gt(ext)»«ENDFOR»
  '''

```

```
51 // all classes transitively referenced from p.
52 def List<EClass> acs(EPackage p) { ... }
53
54 // classes directly declared in p.
55 def List<EClass> alc(EPackage p) { ... }
56
57 // generic revisitor type name, eventually with type bound.
58 def gt(EClass c, boolean ext) {
59   '''«c.name»T «IF ext && c.hasParent»extends «c.parents.head.gt(⊥)»«ENDIF»'''
60 }
61
62 // return True if ecls has a super type.
63 def hasParent(EClass ecls) { ... }
64
65 // return True if the class is required.
66 def rq(EClass c) { ... }
67
68 // list of all concrete subclasses, including c itself.
69 def List<EClass> ascs(EClass c) { ... }
70 ...
```

Listing 7.1 – Compiler of Ecore packages to Java interfaces following the REVISITOR implementation pattern.

References to inherited interfaces are resolved by fully qualified name conventions. Generic parameters are sensible to declaration order. We ensure consistency between declared and bound generics by sorting generic parameter declarations according to their `EClass` fully qualified name's alphabetical order.

Generating implementations consists mainly in the generation of a concrete REVISITOR interface in which generic types are bound to concrete implementations of the factory methods. The body of the concrete factory methods contains a compilation of the ALE classes' methods in Java. Most of the compilation process is straightforward, translating ALE code to the equivalent code in Java.

An interesting point is the implicit introduction of the $-methods call at the relevant places in the REVISITOR implementations, to map the syntactic objects to their respective semantics objects. The introduction is systematic and, from our experience, is the only point that distinguishes the compilation of ALE method bodies to REVISITOR compared to other implementation patterns (e.g., interpreter, visitor). More precisely, $-method calls are introduced on `dot` notations whenever the left-hand side is typed with a metaclass that is part of the language, and the right hand is a method that is part of the left-hand side's ALE class signature. For instance, the guard call presented in Listing 6.1: `self.guard.eval(ctx)` is compiled to the following Java code: `this.revisitor.$(this.getGuard()).eval(ctx)`.

The ALE compiler reads an ALE file, generates the abstract REVISITOR interface based on the imported syntax — if it does not exist yet — and an implementation of the REVISITOR interface defining the actual execution semantics.

To illustrate the behavior of our compiler, we present some interesting parts of the com-

pilation process, drawn from the modular language specifications presented in Section 6.1.

Syntax extension is specified by importing additional Ecore metamodel(s), as shown in Listing 6.4. In this case, the REVISITOR interface generated for the current language module extends the REVISITOR interface from the extended language module. For instance, the `printime` ALE specification (Listing 6.4) imports the `print` ALE specification. In this case, `TimedAlg` inherits from `FsmAlg`, following the extension pattern presented in Listing 4.5. The generated REVISITOR implementation will extend any preexisting implementation for the same behavior. For instance, in this case, `PrintTimed` will extend both the `PrintFsm` REVISITOR implementation as well as `TimedAlg` REVISITOR interface.

In the same way, in the example of Listing 6.5 where multiple ALE specifications are composed, the `GuardedAlg` REVISITOR interface is generated. `GuardedAlg` inherits from `TimedAlg` and `ExpAlg` because it imports two ALE specifications. Besides, the generated REVISITOR implementation inherits from the newly generated `GuardedAlg` REVISITOR interface, and from the REVISITOR implementations generated from the imported ALE specifications.

In summary, reusing existing language specification does not involve the modification or regeneration of previously derived language artifacts.

7.2 Language Extension Evaluation

In this section, we evaluate our approach to language reuse in the context of language extension. To do so, we redefine modularly the semantics of the fUML language as specified in the model execution case of the *Transformation Tool Contest*, 2015 (TTC'15). Our implementation is then compared to three alternative implementations of the same semantics. All material presented in this section is available online.[2]

7.2.1 Scenario

The model execution case of the TTC'15 consists of a subset of the fUML language [111], an executable subset of UML. Defining the execution semantics of fUML proceeds in two steps. First, the static metamodel of the activity diagram is extended by another metamodel that specifies the runtime data needed for capturing the state of executing models. Then, the execution semantics of the fUML language is defined on the resulting metamodel. The organizers of the contest provide a reference implementation, inspired by the Interpreter pattern, where operations are directly embedded in a non-modular way within the Java code generated from the Ecore metamodel.

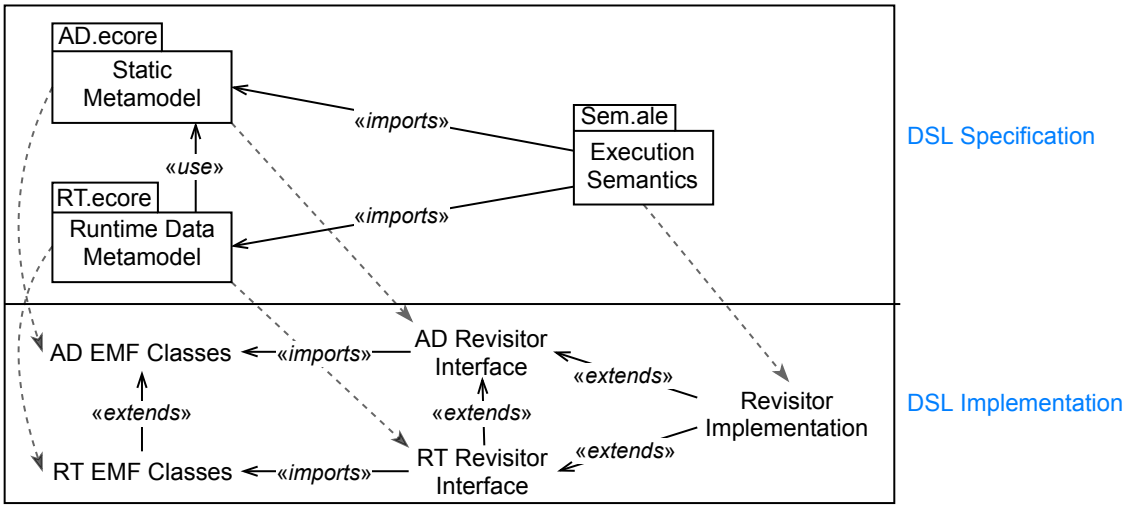


Figure 7.1 – A modular implementation of fUML using REVISITORS and ALE; dashed lines denote generation flows

7.2.2 fUML with ALE

The architecture of the fUML implementation using ALE is shown in Figure 7.1. The syntax is defined by the static metamodel (`AD`) and the runtime data metamodel (`RT`) as defined by the TTC'15 use case. Metaclasses contained in the `RT` metamodel either extend existing metaclasses of the `AD` metamodel to insert new syntactic features that hold the runtime state (e.g., a reference from `Activity` to the `Tokens` it holds), or insert new metaclasses that only play a role at runtime (e.g., `Token` and `Offer`). These metamodels are compiled to corresponding Java classes using the EMF compilation chain (indicated by the dashed arrow on the left of Figure 7.1).

The ALE compiler generates the corresponding REVISITOR interfaces from the same Ecore files. The REVISITOR interface of `RT` extends the REVISITOR interface of `AD` to take into account the new metaclasses, just like the `RT` metamodel extends the `AD` metamodel. Both refer to the classes generated by EMF.

The execution semantics of fUML is defined in an ALE file that imports the `AD` and `RT` metamodels and defines its operations using the `open class` mechanism. From this ALE specification, the compiler generates a REVISITOR implementation, along with the corresponding semantic interfaces. The REVISITOR implementation modularly extends both previously generated REVISITOR interfaces, and provides concrete implementations for the factory methods.

Altogether, the ALE/REVISITOR implementation of fUML supports independent extensibility and incremental compilation since every artifact generated in a given phase is reused *as is* in the following phases. Furthermore, our implementation reuses the metamodels proposed in the TTC'15 model execution case and does not make any assumption

[2] http://gemoc.org/ale/revisitors/revisitor-artifacts.pdf

on the way they are designed, thus fulfilling the opportunistic reuse requirement.

7.2.3 Performance Evaluation

To investigate the performance overhead of the REVISITOR pattern, we compare the performance of the ALE implementation to three other implementations: the reference implementation that follows the Interpreter pattern, a traditional Visitor-based implementation, and an implementation based on the Switch mechanism provided by EMF. Each implementation executes the three performance-oriented benchmarks proposed by the TTC'15 case. The first one (UC1) executes a model of 1000 activity nodes where every activity n is solely connected to the $n+1$-th activity. The second one (UC2) executes a model where one node forks into 1000 intermediate parallel activities that all reconnect to a single join node. The third one (UC3) is similar to UC2, but the 1000 intermediate activities are aggregated in groups of 10 and every consecutive nth group n increments a counter C_n from 0 to 10.

We use two distinct implementations generated from ALE for the benchmarks. The first one is a monolithic implementation of fUML where the semantics is defined on a monolithic metamodel where the runtime concepts are already merged, without using class extension, leading to a single REVISITOR interface and implementation. The second one is the modular implementation presented in Section 7.2.2. Having two versions based on the REVISITOR pattern with different modularity allows us to identify the impact of modularity on language performances. Note that the implementation of the body of the methods of the semantics is identical across all implementation variants: method bodies are copied from the reference implementation. The only difference is the way the code is structured and how dispatch on a model element is realized.

For each benchmark, we proceed to 10 measurements — each running on a new instance of a HotSpotVM version 1.8.0_222 JVM. A measurement consist of 50 warmup iterations, followed by 150 measured iterations.

Figure 7.2 gives an overview of the results of performance evaluation. For each implementation and each use case, it gives the standard deviation and mean of the execution time (in milliseconds).

The Visitor-based implementation show an execution time increased by 6.30% in comparison to the Interpreter-based implementation. This confirms the behavior observed earlier between those two implementation patterns [72]. The implementation based on EMF Switch mechanism is slightly slower than the Interpreter and Visitor variants. In this case, dispatch is implemented by checking EMF's generated `classifierId`, an unique integer identifying each class. As a result, this style of dispatch is outside the realm of the Java language, and hence cannot be optimized as aggressively by the JVM.

The monolithic REVISITOR implementation and modular REVISITOR implementation are both slower than the alternatives. This can be explained by additional object allocation in the factory methods, as well as two levels of indirection: first from within the $-method to the appropriate factory method, and then invoking the method that implements the required

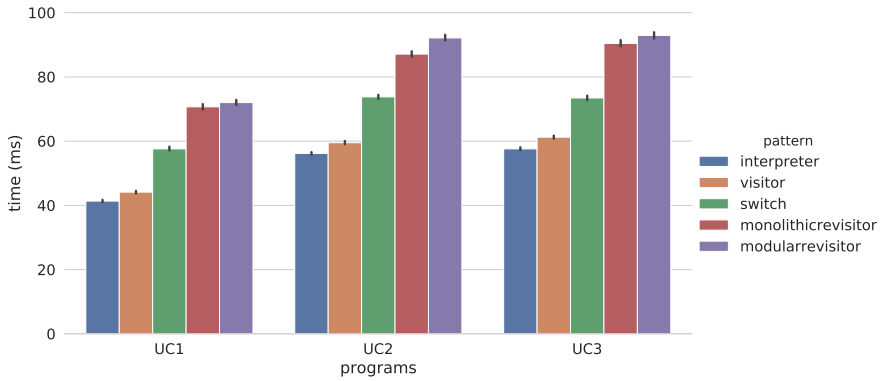


Figure 7.2 – Time measurements of the three use cases. The bar plots present the mean and standard deviation (c=99) for each program and pattern.

behavior (e.g., `run`, `fire`, etc.). Since this is a non-standard form of double dispatch, we assume the JVM is less able to optimize it.

The modular REVISITOR implementation is somewhat slower than the monolithic REVISITOR implementation, and shows a 3.57% increase in its execution time. This can be explained by the fact that the inheritance hierarchies in the metamodel are deeper in the modular case: addition of runtime features to the metamodel is defined by subclassing. In the activity diagram implementation the added inheritance leads to almost four times as many potential runtime checks in the $-methods.

To summarize, these results suggest that the added modularity features of the REVISITOR pattern do introduce additional performance overhead. Nevertheless, we claim that the overhead is an acceptable price to pay for additional benefits in terms of reuse, and that it does not prevent ALE or REVISITORS from being applied in an industrial setting.

7.3 Language Composition Evaluation

This section evaluates the composability of languages specification using ALE and is divided in three parts. First, Section 7.3.1 presents the Internet of Things (IoT) case study used to evaluate our approach. Then, Section 7.3.2 presents the implementation of the case study using ALE. Finally, Section 7.3.3 discusses the impact of our approach on metamodel complexity and runtime performance.

7.3.1 The IoT Case Study

To illustrate our approach, we re-implement a case study that was used to evaluate Melange in earlier work [41]. This case study consists in the definition of an executable modeling language for the Internet of Things (IoT) domain. It targets the definition of systems composed of multiple sensors and actuators deployed on resource-constrained micro-controller

devices (e.g., Arduino, Raspberry Pi, etc.). This language is built by reusing various existing modeling languages and composing them to form the targeted IoT modeling language. We keep the same list of requirements, reminded below:

- the language must provide an Interface Definition Language (IDL) to model the sensor interfaces in terms of provided services;
- the language must support the modeling of concurrent sensor activities;
- the primitive actions that can be invoked within the activities must be expressed with a popular language IoT developers are familiar with.

To fulfill those requirements, Melange's case study selected respectively: the OMG's Interface Definition Language (IDL),[3] the UML Activity Diagram language extracted from the *Transformation Tool Context*,[4] and the scripting language Lua.[5] We reuse the same language modules for our implementation.

Each language module is implemented in its own Eclipse plug-in. Dependencies between the modules are realized by the standard plug-in dependency mechanism offered by Eclipse.

7.3.2 Case Study Implementation

We implement the case study by composing the three modules detailed in Section 7.3.1, together with a fourth module named `IoT` which introduces the domain-specific constructs of an IoT system.

Figure 7.3 depicts how those four modules are composed together. The blank rectangles represent language modules and the red squares represent `«required»` concepts of the interfaces of the modules. Arrows between `«required»` concepts and modules represent the glue and binding defined to compose the modules. They are annotated by labels in the form *required concept* $\rightarrow$ *bound concept*.

The IoT module has two `«required»` concepts, `IoTActivity` and `IoTOperationDef`. They are respectively bound to `Activity` in the AD module and `OperationDef` in the IDL module. The AD module has two `«required»` concepts: `Expression`, and `BooleanVariable`. The former is bound twice: to `OperationDef` in the IDL module, and to `Statement` in the Lua module. The `BooleanVariable` concept is bound to `Statement_Assignment` in the Lua module. Finally, the IDL module has a `«required»` `IdlStmt` concept, bound to `Block` of the Lua module. This way, every `«required»` construct is bound to a concrete concept, and the IoT language is fully defined.

The binding of the concepts of the modules metamodels is defined by introducing a metaclass for each binding. For instance, the binding `IoTActivity` $\rightarrow$ `Activity` is realized by the introduction of a `IoTActivityBindActivity` metaclass, that inherits from `IoTActivity` and holds a reference to `Activity`, following the pattern presented in Section 4.3.3.b.

[3] https://www.omg.org/spec/IDL/
[4] http://www.transformation-tool-contest.eu/
[5] https://www.lua.org/

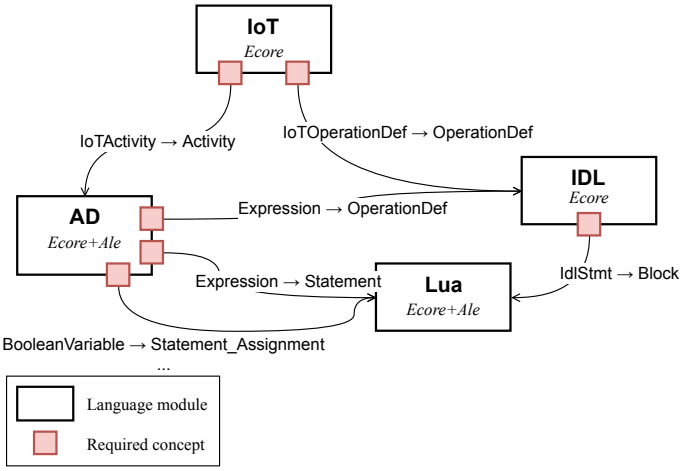


Figure 7.3 – Definition of the IoT language as a composition of four language modules

The glue is realized by re-opening the binding metaclasses with ALE and defining the semantics of the glue. For instance, Listing 7.2 presents an excerpt of the glue for the binding $Expression \rightarrow OperationDef$. The `ExpressionBindOperationDef` is defined in the metamodel of the IoT language. It inherits AD's `Expression` and has a field named `delegate` which references the `OperationDef` of IDL. In Listing 7.2, ALE is used to define the glue between `Expression` and `OperationDef` in the `execute` method of `ExpressionBindOperationDef`.

First, it initializes an `Environment` as expected by the IDL module and populates it with the local variables of the AD module. Then, it invokes the `execute` execution function of `OperationDef` through the $-method, passing the environment as an argument. Finally, once the operation has been executed, it reads the values that have been possibly updated and translate them back in the AD context.

Finally, from these definitions, we use the built-in EMF compiler and our own custom compiler for ALE to derive Java implementations of the modules themselves[6] and the code of their composition following the pattern presented in Section 4.3.3.

In the context of this case study, the glue between language modules is mostly expressed at this level of abstraction and consists of the translation of variables and stores from one module to the other.

7.3.3 Discussion

Starting from the case study presented above, we can question the consequences of the introduction of intermediate glues and bindings on the integration of tools on top of language built by composition.

Indeed, the use of the delegation pattern, described in Section 4.3.3.b, leads to the introduction of new *Bind* metaclasses in the metamodels of composed modules. These classes are needed to compose language modules but are merely technical artifacts that are

[6]unless they have already been compiled, and in this case no re-compilation is needed

```
open class ExpressionBindOperationDef {
  def void execute(Context c) {
    // Initialize the OperationDef environment
    val e = Environment.create()
    for(iv in c.inputValues) {
      e.putVariable(iv.variable.name, iv.value)
    }
    for(l in self.eContainer.oclAsType(OpaqueAction).activity.locals) {
      e.putVariable(l.name, l.currentValue.value)
    }

    // Invoke the execution semantics of OperationDef
    self.delegate.stmt.execute(e)

    // Update the local context back
    for(p in self.delegate.parameters) {
      if(#[PARAM_OUT, PARAM_INOUT].contains(p.direction)) {
        for( l in c.activity.locals) {
          if(l.name() == p.identifier) {
            l.currentValue.setValue(e.getVariable(p.identifier))
          }
        }
      }
    }
  }
}
```

Listing 7.2 – Glue for the $Expression \rightarrow OperationDef$ binding in ALE

not related to the domain constructs materialized by metamodels. Still, language engineers must deal with them when adding new tools on top of the composed language modules.

In order to evaluate the additional cost of the introduction of these extra classes on the development of tools supporting languages, we studied the implementation of an Xtext grammar for the resulting IoT language.

```
ExpBindOpDef returns activitydiagram::Exp:
  {iot_lua::ExpressionBindOperationDef} delegate=[idlmm::OperationDef];
```

Listing 7.3 – Production rule for the $Expression \rightarrow OperationDef$ binding.

We observe that managing the extra *Bind* metaclasses requires to introduce additional intermediate production rules in the grammar. For instance, Listing 7.3 presents a production rule for the `Expression` $\rightarrow$ `OperationDef` binding. These new production rules are only needed to simulate delegation between the production rules of the composed language modules. Such production rules are simple and do not require advanced grammar engineering knowledge. They account for 20 out of 555 lines in the grammar (<4%).

We claim that this cost is largely compensated by the benefits of our approach regarding modularity and reuse.

7.4 Conclusion on Modular Language Reuse

To close this chapter, we discuss the requirements of Section 4.2.4, on the modular reuse of languages.

Independent Extensibility (R1) Language modules can be extended, both syntactically and semantically, and without anticipation. This requirement is presented extensively in the language extension evaluation. However, this is also a fundamental prerequisite of the language composition evaluation too, allowing the definition of the glue language module, which realizes the composition of the independently defined language modules.

Incremental Compilation (R2) Each language is clearly isolated in its own Eclipse plug-in containing an Ecore model for its abstract syntax and an ALE file for its semantics. Each plug-in is type-checked and compiled separately. This means that the IoT language module of the language extension evaluation, could be implemented by importing the Eclipse plug-ins of the other modules from various places (including remotely, for instance) by using the standard Eclipse update site mechanism. This highlights the modularity of our approach. At the implementation level, modules interact with each other only through inheritance and reference to publicly exposed artifacts of the other modules. As long as the interfaces of the modules do not change, each module can evolve internally without having to recompile other language modules.

Opportunistic reuse (R3) This requirement is highlighted in the language extension and language composition evaluations. The language extension evaluation shows the capability of our approach to reuse pre-existing language modules and to compose them modularly. In addition, in the language composition evaluation, the Ecore metamodel, and conforming models were existing well before the beginning of our research, but we were still able to reuse them without recompilation.

Module Encapsulation (R4) The definition of the glue is fully realized through inheritance and invocations of the semantic interfaces of language modules. For instance, in order to compose the four language modules that are part of our language composition evaluation, we extend six classes, five methods are overridden for the definition of the glue, and two classes are needed to express how the internal evaluation contexts of different modules are translated. Language modules without requirements do not have external dependencies towards other language modules. Finally, the glue definitions only interact with a small and well-defined part of the reused language modules. Those observations highlight the isolation capabilities of our approach.

Explicit Required Interfaces (R5) Module requirements are easily identified by looking at the Ecore classes annotated with «required». While most of them are presented in Figure 7.3, the number of «required» classes per language modules is zero for Lua, one for the IDL, three for the AD, and two for the base IoT module itself. Each of those «required» classes is bound exactly once except for the Exp class of the AD module that is bound twice, one time to the IDL module, and another time to the Lua module. Each «required» construct declares a single execution function, except for the IoTOperationDef construct that has no associated semantics. This sums up the information needed for the composition and explicitly exposed in the language modules interfaces.

Module Substitutability (R6) Language modules can be substituted by either providing an alternative modules with an interface that is similar to the one of the initial module, or by modifying the glue. In the first case, the substitution is fully modular. For instance, Lua can be substituted by another language providing the set of concepts referenced by the glue: Statement, Statement_Assignement, and Block. In this case, no other modules has to be modified or recompiled. If the newly introduced module has a different interface, the glue must be adapted to fit the interface of the new module. In this case, only the glue must be recompiled to proceed to the composition.

Non-intrusivity (R7) The definition of the «required» interface is based only on the annotation of classes in the metamodel. This mechanism is built-in EMF directly and does not require any intrusive change. The definition of modular languages leverages inheritance and the delegation pattern [64], which are well-known object-oriented concepts. The REVISITOR implementation pattern is based on three object-oriented concepts: (i) parametric polymorphism (i.e., generics) with bounded type parameters (ii) multiple class or interface inheritance, and (iii) single dynamic dispatch. Such requirements are readily fulfilled by many mainstream object-oriented languages (e.g., Java, C#) and their underlying runtime platforms (e.g., JVM, CLR). In conclusion, every part of our approach is based on existing and well-known object-oriented concepts, which can be easily adapted to similar technological stacks.

This concludes the reuse aspect of this thesis, addressing **challenge #1**. In the next chapter, we present the implementations and evaluations related to the automatic optimization of DSLs interpreters, addressing **challenge #2**.

CHAPTER 8

ALE COMPILER FOR LANGUAGE PERFORMANCE OPTIMIZATION

In this chapter, we first present the implementation details of our compiler dedicated to the optimization of language performance (Section 8.1). Then, we present our evaluation of four languages and eight programs of the performance speedup obtained by the application of our approach (Section 8.2). Finally, we draw some conclusions on the evaluation of our approach to the automatic optimization of language performances (Section 8.3).

8.1 The Truffle Compiler

Figure 8.1 presents an overview of the existing Ecore compiler provided by EMF, and the Ecore + ALE compiler we build to validate our approach. On the left of Figure 8.1, we present the existing Ecore to Java compiler provided as part of the EMF framework, based on the Jet template engine [150]. This compiler allows the generation of Java object models that conform to Ecore semantics and comes with a mechanism to support the re-generation of Java source from an updated Ecore metamodel while preserving the code previously introduced manually in the generated code. This is the base mechanism to follow the Interpreter pattern using EMF.

On the right of the figure, we present our Ecore + ALE compiler that conceptually extends EMF's compiler by supporting the modular introduction of operations on top of Ecore metamodels using ALE. Our initial prototype was developed by extending the existing JET template, but our experience showed the limitation of such a template system during the integration of the compilation of ALE. Indeed, the compilation of the body of ALE methods and the introduction of more variation points in the compilation process (e.g., exploration of different Truffle implementation patterns during our experiments), lead to really large and challenging to maintain templates.

Consequently, we chose to develop our compiler using Xtend and Javapoet. Xtend is one of the technologies at the core of the Xtext framework [55] and proved itself a relevant solution for the implementation of language interpreters and compilers. Javapoet[1] is a Java library dedicated to the generation of Java source code and comes in the form of a fluent API.

At the bottom of Figure 8.1, we present the implementation relation between the Java object model generated by EMF's compiler and the Java object models with Truffle concepts generated by our compiler. Indeed, the object model produced by our compiler

[1] Javapoet: https://github.com/square/javapoet

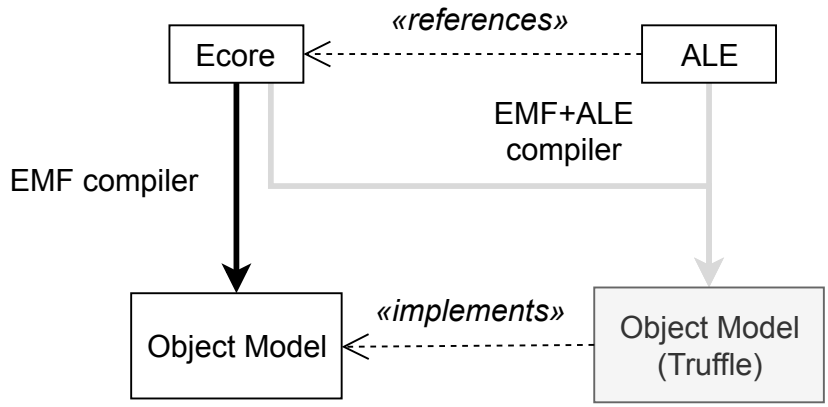


Figure 8.1 – EMF's compiler and our compiler dedicated to the optimization of DSLs performance.

implements the same interface as the object model produced by the Ecore compiler of EMF, making the former fully substitutable for the latter. Second, the introduced Truffle concepts are built to preserve the semantics. Consequently, the execution of two versions of the same specification compiled with and without our approach produces the same results. Meaning that our compiler is a safe replacement to EMF's compiler, allowing a fully automated gain of performance on top of interpreters running on GraalVM.

Our compiler is about 4000 lines of code. The Truffle specific parts are composed of 7 lines of code in the method body compiler, 180 lines of code in the classes structure generation, mainly related to the introduction of `@TruffleBoundary` annotations, and 2 lines of code in the EMF factory compiler, also for the introduction of `@TruffleBoundary` annotations. The integration of our approach in an EMF compiler required 189 lines of code in total. Porting back our approach to the official EMF implementation is straightforward and is the matter of translating the 189 lines into the JET template formalism.

8.2 Language Performance Evaluation

In this section, we present the evaluation of our approach. In Section 8.2.1, we present the languages and programs used for the evaluation. Finally, Section 8.2.2 presents and analyzes the experimentation results.

8.2.1 Benchmarked Languages and Programs

To evaluate our results, we implemented four languages: MiniJava [136], a teaching-oriented subset of Java, a functional language inspired by OCaml named Boa[2], a Finite State Machine language[3], and the educational procedural Logo language.

[2]Programming languages Zoo: http://plzoo.andrej.com/language/boa.html
[3]Finite State Machine language: https://github.com/gemoc/MODELS2017Tutorial/

MiniJava is used to implement: the Fibonacci algorithm (`m_fibonacci`), a bubble sort algorithm [37] (`m_sort`), a binary tree manipulation algorithm[4] (`binarytree`), and an implementation of the fannkuch algorithm[5] [4] (`fannkuchredux`). Boa is used to implement a Fibonacci algorithm (`b_fibonacci`) and an insert sort algorithm [90] (`b_sort`). The Finite State Machine language is used to define a set of four communicating state machines, sending messages to each other, presented in more detail on the companion webpage[6] (`buffers`). Finally, the Logo language is used to define a program that draws a Koch snowflake fractal[7] (`fractal`).

This selection represents a gallery of languages. The choice has been made for being a maximum representative of targeted languages and conforming programs or models. This includes one functional language, one object-oriented language, a language dedicated to domain experts, and an "end-user" language. For the languages with paradigms allowing various sort of implementation, we propose programs covering different styles from arithmetic intensive programs to structure intensive programs, and from recursive style to iterative style.

8.2.2 Results

Table 8.1 summarizes the measured performance of the programs on our benchmark. The results presented below are measured using HotSpot VM version 1.8.0_222, GraalVM version 19.1.1 and Truffle version 19.1.1.

The *Mean* is calculated by benchmarking each combination of a virtual machine, an implementation pattern, and a program by measuring three times 50 executions.

By manual inspection of the measured time, we observe that the ten first executions are enough to warm up the virtual machine and to reach a steady-state. We calculate the performance of the programs using the mean of the 120 remaining executions, once the ten warmup iterations of each run have been excluded. Additionally, we evaluated the confidence interval of the measurements for a confidence level of 99%. All the confidence intervals are below 0.03 seconds, which allows a safe and unambiguous analysis of our results.

The HotSpot VM + Switch, HotSpot VM + Interpreter, and GraalVM + Interpreter lines present the measurements of respectively: the Switch implementation on the HotSpot VM; the Interpreter implementation on the HotSpot VM; and the Interpreter implementation on the GraalVM. The ATG + Interpreter line presents the measurements of language compiled with our approach. Each *Mean* line presents the calculated mean time of the measurements in seconds.

[4]Binarytree algorithm: https://benchmarksgame-team.pages.debian.net/benchmarksgame/description/binarytrees.html#binarytrees

[5]Fannkuch algorithm: https://benchmarksgame-team.pages.debian.net/benchmarksgame/description/fannkuchredux.html#fannkuchredux

[6]Companion webpage: https://manuelleduc.github.io/ecmfa-2020/#system-of-finite-state-machines-example

[7]Koch snowflake: https://en.wikipedia.org/wiki/Koch_snowflake

Table 8.1 – Benchmarks measurement summary. Mean = mean execution time in second; SHS=Speedup relative to the HotSpot VM + Switch; SHI = Speedup relative to the HotSpot VM + Interpreter; SGI = Speedup relative to the GraalVM + Interpreter; ATG = Automatic Truffle Generation

	Minijava				Boa		FSM	Logo	Geo. Mean
	m_fibonacci	m_sort	binarytree	fannkuchredux	b_fibonacci	b_sort	buffers	fractal	
parameters:	(30)	(1000)	(11)	(8)	(30)	(500)	($5 \cdot 10^7$)	(17)	
HotSpot VM + Switch									
Mean (s)	11.80	16.39	7.06	6.67	2.04	1.94	7.84	12.44	
HotSpot VM + Interpreter									
Mean (s)	9.18	14.91	5.86	5.28	1.79	1.71	4.64	11.14	
SHS	x1.28	x1.10	x1.20	x1.26	x1.14	x1.13	x1.69	x1.12	x1.23
GraalVM + Interpreter									
Mean (s)	5.29	9.89	3.32	3.17	1.49	1.57	4.04	7.87	
SHI	x1.74	x1.51	x1.77	x1.66	x1.22	x1.09	x1.15	x1.42	x1.42
SHS	x2.23	x1.66	x2.13	x2.10	x1.40	x1.23	x1.94	x1.58	x1.75
ATG + Interpreter									
Mean (s)	4.21	9.23	2.98	2.72	1.25	1.43	3.54	7.24	
SGI	**x1.26**	**x1.07**	**x1.11**	**x1.17**	**x1.17**	**x1.10**	**x1.14**	**x1.09**	**x1.14**
SHI	x2.18	x1.61	x1.97	x1.94	x1.43	x1.19	x1.31	x1.547	x1.61
SHS	x2.81	x1.78	x2.37	x2.45	x1.63	x1.35	x2.21	x1.72	x1.99

Each SHS line presents the speedup of the current implementation compared to the HotSpot VM + Switch implementation. Each SHI line presents the speedup of the current implementation compared to the HotSpot VM + Interpreter implementation. Finally, the SGI line presents the speedup of the current implementation compared to the GraalVM + Interpreter implementation.

The speedups discussed below are produced using the geometric mean of the speedups obtained for each program on a given configuration and are presented in the rightmost column. We use the term *general* speedup when talking about the speedup calculated using the geometric mean.

First, as presented in Section 5.4.1, the straightforward transition from HotSpot VM + Switch to GraalVM + Interpreter already allows a general speedup of x1.75, ranging from x1.23 to x2.23.

Our approach, built on top of the GraalVM + Interpreter implementation, yields an additional general speedup of x1.14, ranging from x1.07 to x1.26. This adds up to a general speedup of x1.99, ranging from x1.35 to x2.81, when compared to the HotSpot VM + Switch implementation. While the effect of our contribution leads to smaller speedups than the straightforward switch from HotSpot VM + Switch to GraalVM + Interpreter, is it important to consider the opportunity offered by our approach to provide additional performance gains to language engineers at zero cost. Indeed, the only manual operation to perform is the recompilation of languages using a new compiler.

Complementarily, we also evaluate the manually introduced PIC presented in Section 5.4.4.a. Introducing the PIC yield a general speedup of x1.08, ranging from x0.92 to x1.19 relatively to the ATG + Interpreter implementation, making it detrimental in some cases and forces language engineers to manipulate performance-specific concepts during language specification.

In conclusion, our results show the benefit of our approach in all cases, especially when

improving the performance of language interpreters to their maximum is required while preserving the compatibility with existing EMF tool support.

8.3 Conclusion on the Automatic Optimization of Language Performances

In this chapter, we presented the evaluation of our approach to the automatic optimization of language performances. Through the implementation and evaluation of four representative languages and eight programs, we show a speedup of x1.14 on average, ranging from x1.07 to x1.26, while requiring no additional development effort from language engineers. When asking the language engineers to use an additional declarative keyword, for the Polymorphic Inline Cache optimization, we obtain a speedup of x1.23 on average, ranging from x1.05 to x1.36.

PART IV

Conclusion and Perspectives

CHAPTER 9

CONCLUSION AND PERSPECTIVES

In this chapter, we summarize the contributions of this thesis (Section 9.1). Then, we propose some research directions that emerge from our results (Section 9.2).

9.1 Conclusion

The development of DSLs in the context of MDE involves the use of metalanguages. These metalanguages provide useful abstractions that assist language engineers in two ways. On the one hand, they offer abstractions dedicated to the specification of language concepts (e.g., abstract syntax, concrete syntax, semantics). On the other hand, they ensure the applicability of software engineering practices (e.g., modularity, tests, performances). However, some useful properties available on language specification are lost when they are translated to language service implementations, especially regarding reusability and performance.

We state that these limitations can be alleviated by better exploiting the abstractions available at the language specification level. In other words, without impacting the current way of specifying languages. From this observation, we draw two challenges, improving the reusability of language service implementations (**challenge #1**), and improving the performances of language service implementations (**challenge #2**).

We address the challenge of language reuse (**challenge #1**) by proposing the REVISITOR implementation pattern. The REVISITOR pattern allows the reuse of language modules. This reuse can be realized safely and modularly, both syntactically and semantically, while relying solely on object-oriented programming concepts.

We evaluate the REVISITOR implementation pattern by defining heterogeneous DSL modules. First, we build a modular fUML language, by copying the existing monolithic specification of the existing fUML language from the *Transformation Tool Contest*, 2015 (TTC'15). Second, we build an IoT language by the composition of independently defined language modules. This IoT language is a copy of an existing modular language specification, initially specified using Kermeta [79] and composed with Melange [41].

Moreover, we evaluate the impact of the REVISITOR implementation pattern on the runtime performances of interpreter implementations by comparing the execution time of four implementations of the fUML language. Each implementation conforms to a different implementation pattern [64]: the Interpreter pattern, the Visitor pattern, the EMF Switch pattern [150], and the REVISITOR pattern [104]. We observe that the REVISITOR has a performance overhead comparable to that of the other implementation patterns while offering additional benefits in terms of reuse while remaining compliant with the existing

tool support of languages.

Then, we address the automatic introduction of language-specific optimizations in language interpreter implementations (**challenge #2**). We evaluate our approach by measuring the speedup induced by the optimizations on eight programs defined in four languages. We select the evaluated languages to be representative of the existing diversity found in languages. Our selection of languages includes an object-oriented GPL (MiniJava), a functional GPL (Boa), a modeling language (a Finite State Machine DSL), and an educative language (Logo). From the eight programs, we implement four programs in MiniJava, two in Boa, one in the Finite State Machine DSL, and one in Logo. We also select the evaluated programs to be as diverse as possible. This selection includes iterative and recursive programs as well as arithmetic-intensive and structure-intensive programs. We compare our automatically optimized language implementations to the standard language implementation patterns, running on the JVM. Our measurements show a speedup of x1.14, ranging from x1.07 to x1.26.

These optimizations are introduced without impacting the usual development method of language engineers and preserve the compatibility with the tool support of languages.

All our contributions are seamlessly integrated into the EMF ecosystem, a de facto standard both in academia and in the industry.

In conclusion, we exploit the abstractions available in language specifications to improve the reusability and the performances of language service implementations. Additionally, we did not impact the usual development methodology of language engineers and present its integration in a well known and exploited framework. Thus, we believe our contributions are widely applicable in an industrial context, both on legacy language specifications and for the development of new languages.

9.2 Perspectives

This work is the first step towards more in-depth exploitation of high-level abstractions manipulated by language engineers. We identified some promising research directions during the thesis. We present three research directions: Reusability and Performance tradeoff (Section 9.2.1), Automatic Domain Adaptation (Section 9.2.2), and Safer Language Reuse (Section 9.2.3).

9.2.1 Reusability and Performance Tradeoff

The two challenges of this thesis are aimed at improving the reusability and performance of DSLs. We address these challenges by proposing implementation patterns and translations from existing metalanguages to those implementation patterns. So far, each implementation pattern addresses a single challenge. One way to go further in this direction is to support implementation patterns that support both the improvement of the reusability and performance of DSLs. Merging these two challenges would require first to study performance-oriented implementation patterns that support the constraints of reusability.

For instance, reusability induces modularity, which implies information hiding, which can be detrimental to software performances. Consequently, it is interesting to provide language engineers with clear guidelines and tradeoffs of such implementation patterns.

9.2.2 Automatic Domain Adaptation

DSLs are bound to be used and executed in new and unanticipated contexts. Consequently, language engineers should be able to understand the impact of their design choices better and should be provided with the tools and methods to adapt existing languages to new development environments quickly.

Design Choices Impact Quantification Throughout this thesis, we ask ourselves the question of the relevant design choices, at every level, to improve the non-functional properties of languages. We observed that such design choices could have a complex impact on the resulting language services implementation. Furthermore, the order of magnitude of the impact of such choices, and in which circumstances, are difficult to anticipate.

For instance, we observed during our researches on language performance that the introduction of the some optimizations (e.g., the PIC) could be beneficial for some programs while being detrimental to some others. Also, our effort to quantify the speedup of language interpreter implementations repeatably highlight the complexity of the task, which requires to apply carefully crafted benchmarking environments and methodologies.

This makes it difficult and costly for language engineers to make informed design decisions and to anticipate the impact of their design choices on the non-functional properties of languages. Consequently, language engineers should be equipped with tools and methods to evaluate the impact of their design choices.

From these observations, we propose two possible axes of research. First, we propose to study the relevant properties to quantify and to explore the context in which design decisions impact such properties. Second, we propose to study how such properties can be communicated in an actionable way to language engineers. Furthermore, we propose to study how such additional information impacts the language engineering process and its results.

Development Environment Agility Traditionally, modeling is done in a desktop application running locally and with very few network interactions, if we exclude dependency management. With the introduction of programming scientific notebooks and IDE integrated into web browsers, activities related to modeling are quickly shifting towards distributed and web-based platforms. This shows a trend towards the integration of languages into new kinds of environments.

In this quickly evolving context, it is not realistic to ask language engineers to port legacy languages to new environments at such a fast pace.

From those observations, we propose to address the challenge of the adaptation of language specification to new development environments without being intrusive of legacy language specifications.

Besides, the shift of language to new environments gives access to programming activities to new kinds of users. Consequently, it is interesting to study the capability of new and possibility untrained users to perform programming tasks in these new development contexts.

9.2.3 Safer Language Reuse

In this thesis, we studied some challenges related to language reuse. Doing so, we identified multiple research directions we deem interesting, presented below.

Automatic composition In this thesis, the reuse of language modules is realized by the introduction of glue code, dedicated to the connection of the language modules. This addresses the problem using the tool familiar to the language engineers. Nonetheless, this task is error-prone and can quickly become time-consuming when composing large language modules. Consequently, it is interesting to explore the challenge of the automatic composition of language modules. Either to diminish the amount of work required by language engineers or even to allow domain experts to choose and compose language modules for their needs without the intervention of language engineers.

Language Module Granularity The long-studied field of object-oriented software development led to the definition of well-defined guidelines for choosing the right granularity of objects [110], i.e., how to distribute roles and constructs among objects. In particular, this leads to the definition of design patterns [64] that guide developers with well studied and reusable patterns of object-oriented software design.

The same question, applied to the development of languages, does not yield a clear answer, and no guidelines exist to define the granularity of language modules.

In principle, language modules may range from small generic language modules (e.g., Expressions) to large framework-like language modules (e.g., automotive business processes languages). From these observations, we believe it is interesting to explore the definition of a set of generalized and well-studied granularity guidelines that would be beneficial for the production of reusable language modules.

PART V

Indexes

LIST OF FIGURES

List of Tables

List of Listings

BIBLIOGRAPHY

[1] Dec. 2005, URL: https://www.omg.org/spec/UML/2.5.1 (cit. on p. 14).

[2] Ali-Reza Adl-Tabatabai et al., "Fast, Effective Code Generation in a Just-in-time Java Compiler", in: *SIGPLAN Not.* 33.5 (May 1998), pp. 280–290, ISSN: 0362-1340, DOI: 10.1145/277652.277740, URL: http://doi.acm.org/10.1145/277652.277740 (cit. on p. 74).

[3] Marcus Alanen, Ivan Porres, et al., *A relation between context-free grammars and meta object facility metamodels*, Citeseer, 2004 (cit. on p. 19).

[4] Kenneth R. Anderson and Duane Rettig, "Performing Lisp Analysis of the FANNKUCH Benchmark", in: *SIGPLAN Lisp Pointers* VII.4 (Oct. 1994), pp. 2–12, ISSN: 1045-3563, DOI: 10.1145/382109.382124, URL: http://doi.acm.org/10.1145/382109.382124 (cit. on p. 103).

[5] Colin Atkinson and Thomas Kühne, "Model-Driven Development: A Metamodeling Foundation", in: *IEEE Software* 20.5 (2003), pp. 36–41, DOI: 10.1109/MS.2003.1231149, URL: https://doi.org/10.1109/MS.2003.1231149 (cit. on p. 12).

[6] John Aycock, "A brief history of just-in-time", in: *ACM Comput. Surv.* 35.2 (2003), pp. 97–113, DOI: 10.1145/857076.857077, URL: https://doi.org/10.1145/857076.857077 (cit. on p. 33).

[7] John W. Backus et al., "Report on the algorithmic language ALGOL 60", in: *Commun. ACM* 3.5 (1960), pp. 299–314, DOI: 10.1145/367236.367262, URL: https://doi.org/10.1145/367236.367262 (cit. on p. 19).

[8] Ankica Barisic et al., "How to reach a usable DSL? Moving toward a Systematic Evaluation", in: *ECEASST* 50 (2011), DOI: 10.14279/tuj.eceasst.50.741, URL: https://doi.org/10.14279/tuj.eceasst.50.741 (cit. on p. 17).

[9] Edd Barrett et al., "Fine-grained Language Composition: A Case Study", in: *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, ed. by Shriram Krishnamurthi and Benjamin S. Lerner, vol. 56, LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016, 3:1–3:27, ISBN: 978-3-95977-014-9, DOI: 10.4230/LIPIcs.ECOOP.2016.3, URL: https://doi.org/10.4230/LIPIcs.ECOOP.2016.3 (cit. on p. 29).

[10] Edd Barrett et al., "Virtual machine warmup blows hot and cold", in: *PACMPL* 1.*OOPSLA* (2017), 52:1–52:27, DOI: 10.1145/3133876, URL: https://doi.org/10.1145/3133876 (cit. on p. 87).

[11] Bas Basten et al., “Modular language implementation in Rascal - experience report”, in: *Sci. Comput. Program.* 114 (2015), pp. 7–19, DOI: 10.1016/j.scico.2015.11.003, URL: https://doi.org/10.1016/j.scico.2015.11.003 (cit. on pp. 36, 54).

[12] Alexander Bergmayr and Manuel Wimmer, “Generating Metamodels from Grammars by Chaining Translational and By-Example Techniques”, in: *Proceedings of the First International Workshop on Model-driven Engineering By Example co-located with ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2013), Miami, Florida, USA, September 29, 2013*. Ed. by Marouane Kessentini, Philip Langer, and Houari A. Sahraoui, vol. 1104, CEUR Workshop Proceedings, CEUR-WS.org, 2013, pp. 22–31, URL: http://ceur-ws.org/Vol-1104/3.pdf (cit. on p. 19).

[13] Aggelos Biboudis, Pablo Inostroza, and Tijs van der Storm, “Recaf: Java dialects as libraries”, in: *ACM SIGPLAN Notices* 52.3 (Oct. 2016), pp. 2–13, DOI: 10.1145/3093335.2993239, URL: https://doi.org/10.1145%2F3093335.2993239 (cit. on p. 44).

[14] L. Thomas van Binsbergen, Neil Sculthorpe, and Peter D. Mosses, “Tool support for component-based semantics”, in: *Companion Proceedings of the 15th International Conference on Modularity, Málaga, Spain, March 14 - 18, 2016*, ed. by Lidia Fuentes, Don S. Batory, and Krzysztof Czarnecki, ACM, 2016, pp. 8–11, ISBN: 978-1-4503-4033-5, DOI: 10.1145/2892664.2893464, URL: https://doi.org/10.1145/2892664.2893464 (cit. on p. 32).

[15] Carl Friedrich Bolz and Laurence Tratt, “The impact of meta-tracing on VM design and implementation”, in: *Sci. Comput. Program.* 98 (2015), pp. 408–421, DOI: 10.1016/j.scico.2013.02.001, URL: https://doi.org/10.1016/j.scico.2013.02.001 (cit. on pp. 34, 38).

[16] Carl Friedrich Bolz et al., “Tracing the meta-level: PyPy’s tracing JIT compiler”, in: *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems, ICOOOLPS 2009, Genova, Italy, July 6, 2009*, ed. by Ian Rogers, ACM, 2009, pp. 18–25, ISBN: 978-1-60558-541-3, DOI: 10.1145/1565824.1565827, URL: https://doi.org/10.1145/1565824.1565827 (cit. on pp. 34, 38).

[17] Patrick Borras et al., “CENTAUR: The System”, in: *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Boston, Massachusetts, USA, November 28-30, 1988*, ed. by Peter B. Henderson, ACM, 1988, pp. 14–24, ISBN: 0-89791-290-X, DOI: 10.1145/64135.65005, URL: https://doi.org/10.1145/64135.65005 (cit. on p. 22).

[18] 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, Amsterdam, The Netherlands, October 31 - November 1, 2016*, ed. by Tijs van der Storm, Emilie Balland, and Dániel Varró, ACM, 2016, pp. 84–89, ISBN: 978-1-4503-4447-0, DOI: 10.1145/2997364, URL: http://dl.acm.org/citation.cfm?id=2997384 (cit. on p. 74).

[19] Mark van den Brand, Dragan Gasevic, and Jeff Gray, eds., *Software Language Engineering, Second International Conference, SLE 2009, Denver, CO, USA, October 5-6, 2009, Revised Selected Papers*, vol. 5969, Lecture Notes in Computer Science, Springer, 2010, ISBN: 978-3-642-12106-7, DOI: 10.1007/978-3-642-12107-4, URL: https://doi.org/10.1007/978-3-642-12107-4.

[20] Martin Bravenboer et al., “Stratego/XT 0.17. A language and toolset for program transformation”, in: *Sci. Comput. Program.* 72.*1-2* (2008), pp. 52–70, DOI: 10.1016/j.scico.2007.11.003, URL: https://doi.org/10.1016/j.scico.2007.11.003 (cit. on p. 36).

[21] Arvid Butting et al., “Modeling language variability with reusable language components”, in: *Proceeedings of the 22nd International Systems and Software Product Line Conference - Volume 1, SPLC 2018, Gothenburg, Sweden, September 10-14, 2018*, ed. by Thorsten Berger et al., ACM, 2018, pp. 65–75, DOI: 10.1145/3233027.3233037, URL: https://doi.org/10.1145/3233027.3233037 (cit. on p. 35).

[22] Arvid Butting et al., “Systematic composition of independent language features”, in: *Journal of Systems and Software* 152 (2019), pp. 50–69, DOI: 10.1016/j.jss.2019.02.026, URL: https://doi.org/10.1016/j.jss.2019.02.026 (cit. on pp. 30, 35, 46, 54).

[23] Peter Canning et al., “F-bounded Polymorphism for Object-oriented Programming”, in: *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA ’89, Imperial College, London, United Kingdom: ACM, 1989, pp. 273–280, ISBN: 0-89791-328-0, DOI: 10.1145/99370.99392, URL: http://doi.acm.org/10.1145/99370.99392 (cit. on p. 32).

[24] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan, “Finally Tagless, Partially Evaluated”, in: *Programming Languages and Systems*, Springer Berlin Heidelberg, pp. 222–238, DOI: 10.1007/978-3-540-76637-7_15, URL: https://doi.org/10.1007%2F978-3-540-76637-7_15 (cit. on pp. 32, 38).

[25] Walter Cazzola and Diego Mathias Olivares, “Gradually Learning Programming Supported by a Growable Programming Language”, in: *IEEE Trans. Emerging Topics Comput.* 4.*3* (2016), pp. 404–415, DOI: 10.1109/TETC.2015.2446192, URL: https://doi.org/10.1109/TETC.2015.2446192 (cit. on p. 36).

[26] Walter Cazzola and Edoardo Vacchi, "Neverlang 2 - Componentised Language Development for the JVM", in: *Software Composition - 12th International Conference, SC 2013, Budapest, Hungary, June 19, 2013. Proceedings*, ed. by Walter Binder, Eric Bodden, and Welf Löwe, vol. 8088, Lecture Notes in Computer Science, Springer, 2013, pp. 17–32, ISBN: 978-3-642-39613-7, DOI: 10.1007/978-3-642-39614-4_2, URL: https://doi.org/10.1007/978-3-642-39614-4_2 (cit. on p. 54).

[27] Walter Cazzola and Edoardo Vacchi, "On the incremental growth and shrinkage of LR goto-graphs", in: *Acta Inf.* 51.7 (2014), pp. 419–447, DOI: 10.1007/s00236-014-0201-2, URL: https://doi.org/10.1007/s00236-014-0201-2 (cit. on p. 36).

[28] Betty H. C. Cheng et al., "On the Globalization of Domain-Specific Languages", in: *Globalizing Domain-Specific Languages - International Dagstuhl Seminar Dagstuhl Castle, Germany, October 5-10, 2014 Revised Papers*, ed. by Betty H. C. Cheng et al., vol. 9400, Lecture Notes in Computer Science, Springer, 2014, pp. 1–6, ISBN: 978-3-319-26171-3, DOI: 10.1007/978-3-319-26172-0_1, URL: https://doi.org/10.1007/978-3-319-26172-0_1 (cit. on p. 17).

[29] Martin Churchill et al., "Reusable Components of Semantic Specifications", in: *Trans. Aspect-Oriented Software Development*, Lecture Notes in Computer Science 12 (2015), ed. by Shigeru Chiba et al., pp. 132–179, DOI: 10.1007/978-3-662-46734-3_4, URL: https://doi.org/10.1007/978-3-662-46734-3_4 (cit. on p. 31).

[30] Tony Clark, "Type Checking UML Static Diagrams", in: *«UML»'99: The Unified Modeling Language - Beyond the Standard, Second International Conference, Fort Collins, CO, USA, October 28-30, 1999, Proceedings*, ed. by Robert B. France and Bernhard Rumpe, vol. 1723, Lecture Notes in Computer Science, Springer, 1999, pp. 503–517, DOI: 10.1007/3-540-46852-8_36, URL: https://doi.org/10.1007/3-540-46852-8_36 (cit. on p. 48).

[31] Thomas Cleenewerck, "Component-Based DSL Development", in: *Generative Programming and Component Engineering, Second International Conference, GPCE 2003, Erfurt, Germany, September 22-25, 2003, Proceedings*, ed. by Frank Pfenning and Yannis Smaragdakis, vol. 2830, Lecture Notes in Computer Science, Springer, 2003, pp. 245–264, ISBN: 3-540-20102-5, DOI: 10.1007/978-3-540-39815-8_15, URL: https://doi.org/10.1007/978-3-540-39815-8_15 (cit. on p. 30).

[32] Curtis Clifton et al., "MultiJava: modular open classes and symmetric multiple dispatch for Java", in: *Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA 2000), Minneapolis, Minnesota, USA, October 15-19, 2000*. Ed. by Mary Beth Rosson and Doug Lea, ACM, 2000, pp. 130–145, ISBN: 1-58113-200-X, DOI:

10.1145/353171.353181, URL: http://doi.acm.org/10.1145/353171.353181 (cit. on pp. 32, 83).

[33] Benoît Combemale, Ralf Lämmel, and Eric Van Wyk, "SLEBOK: The Software Language Engineering Body of Knowledge (Dagstuhl Seminar 17342)", in: *Dagstuhl Reports* 7.8 (2018), ed. by Manuel Leduc, pp. 45–54, ISSN: 2192-5283, DOI: 10.4230/DagRep.7.8.45, URL: http://drops.dagstuhl.de/opus/volltexte/2018/8429 (cit. on p. 8).

[34] Benoît Combemale et al., *Engineering modeling languages: Turning domain knowledge into tools*, Chapman and Hall/CRC, 2016 (cit. on p. 11).

[35] Benoît Combemale et al., "Concern-oriented language development (COLD): Fostering reuse in language engineering", in: *Computer Languages, Systems & Structures* 54 (2018), pp. 139–155, DOI: 10.1016/j.cl.2018.05.004, URL: https://doi.org/10.1016/j.cl.2018.05.004 (cit. on pp. 8, 49).

[36] William R. Cook, "On understanding data abstraction, revisited", in: *OOPSLA*, ACM, 2009, pp. 557–572 (cit. on p. 19).

[37] Thomas H Cormen et al., *Introduction to algorithms second edition*, 2001 (cit. on p. 103).

[38] 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, SLE 2018, Boston, MA, USA, November 05-06, 2018*, ed. by David Pearce, Tanja Mayerhofer, and Friedrich Steimann, ACM, 2018, pp. 215–219, ISBN: 978-1-4503-6029-6, DOI: 10.1145/3276604.3276623, URL: https://doi.org/10.1145/3276604.3276623 (cit. on p. 23).

[39] Jesús Sánchez Cuadrado and Juan de Lara, "Open meta-modelling frameworks via meta-object protocols", in: *Journal of Systems and Software* 145 (2018), pp. 1–24, ISSN: 0164-1212, DOI: https://doi.org/10.1016/j.jss.2018.07.023, URL: http://www.sciencedirect.com/science/article/pii/S0164121218301432 (cit. on p. 31).

[40] Thomas Degueule, Benoît Combemale, and Jean-Marc Jézéquel, "On Language Interfaces", in: *Present and Ulterior Software Engineering*. 2017, pp. 65–75, DOI: 10.1007/978-3-319-67425-4_5, URL: https://doi.org/10.1007/978-3-319-67425-4_5 (cit. on pp. 17, 31, 47, 49).

[41] Thomas Degueule et al., "Melange: a meta-language for modular and reusable development of DSLs", in: *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, SLE 2015, Pittsburgh, PA, USA, October 25-27, 2015*, ed. by Richard F. Paige, Davide Di Ruscio, and Markus Völter, ACM, 2015, pp. 25–36, ISBN: 978-1-4503-3686-4, DOI: 10.1145/2814251.2814252, URL: http://doi.acm.org/10.1145/2814251.2814252 (cit. on pp. i, 4, 17, 23, 29, 31, 35, 38, 44, 47, 95, 109).

[42] Thomas Degueule et al., “Safe model polymorphism for flexible modeling”, in: *Computer Languages, Systems & Structures* 49 (2017), pp. 176–195, DOI: 10.1016/j.cl.2016.09.001, URL: https://doi.org/10.1016/j.cl.2016.09.001 (cit. on pp. 46, 52).

[43] Arie van Deursen and Paul Klint, “Little languages: little maintenance?”, in: *Journal of Software Maintenance* 10.2 (1998), pp. 75–92, DOI: 10.1002/(SICI)1096-908X(199803/04)10:2%3C75::AID-SMR168%3E3.0.CO;2-5, URL: https://doi.org/10.1002/(SICI)1096-908X(199803/04)10:2%3C75::AID-SMR168%3E3.0.CO;2-5 (cit. on p. 16).

[44] Arie van Deursen, Paul Klint, and Joost Visser, “Domain-specific languages: an annotated bibliography”, in: *SIGPLAN Notices* 35.6 (2000), pp. 26–36, DOI: 10.1145/352029.352035, URL: https://doi.org/10.1145/352029.352035 (cit. on p. 16).

[45] L. Peter Deutsch and Allan M. Schiffman, “Efficient Implementation of the Smalltalk-80 System”, in: *POPL*, ACM Press, 1984, pp. 297–302 (cit. on p. 78).

[46] Tom Dinkelaker, Christian Wende, and Henrik Lochmann, *Implementing and Composing MDSD-Typical DSLs*, Technische Universität Darmstadt, Oct. 2009, URL: http://tuprints.ulb.tu-darmstadt.de/1939/ (cit. on p. 31).

[47] Sven Efftinge et al., “Xbase”, in: *Proceedings of the 11th International Conference on Generative Programming and Component Engineering - GPCE ’12*, ACM Press, 2012, DOI: 10.1145/2371401.2371419, URL: https://doi.org/10.1145%2F2371401.2371419 (cit. on p. 48).

[48] Torbjörn Ekman and Görel Hedin, “The JastAdd system - modular extensible compiler construction”, in: *Sci. Comput. Program.* 69.1-3 (2007), pp. 14–26, DOI: 10.1016/j.scico.2007.02.003, URL: https://doi.org/10.1016/j.scico.2007.02.003 (cit. on p. 34).

[49] Matthew Emerson and Janos Sztipanovits, “Techniques for metamodel composition”, in: *OOPSLA–6th Workshop on Domain Specific Modeling*, 2006, pp. 123–139 (cit. on pp. 28, 30).

[50] Sebastian Erdweg, Paolo G. Giarrusso, and Tillmann Rendel, “Language composition untangled”, in: *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications - LDTA ’12* (2012), pp. 1–8, DOI: 10.1145/2427048.2427055, URL: https://www.mathematik.uni-marburg.de/~rendel/erdweg12language.pdf (cit. on pp. 43, 46).

[51] 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, DOI: 10.1016/j.cl.2015.08.007, URL: https://doi.org/10.1016/j.cl.2015.08.007 (cit. on p. 23).

[52] Sebastian Erdweg et al., “SugarJ: library-based syntactic language extensibility”, in: *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, ed. by Cristina Videira Lopes and Kathleen Fisher, ACM, 2011, pp. 391–406, ISBN: 978-1-4503-0940-0, DOI: 10.1145/2048066.2048099, URL: https://doi.org/10.1145/2048066.2048099 (cit. on p. 46).

[53] M. Anton Ertl and David Gregg, “The Behavior of Efficient Virtual Machine Interpreters on Modern Architectures”, in: *Euro-Par 2001 Parallel Processing*, Springer Berlin Heidelberg, 2001, pp. 403–413, DOI: 10.1007/3-540-44681-8_59, URL: https://doi.org/10.1007%2F3-540-44681-8_59 (cit. on p. 74).

[54] M. Anton Ertl and David Gregg, “The Structure and Performance of Efficient Interpreters”, in: *J. Instruction-Level Parallelism* 5 (2003), URL: http://www.jilp.org/vol5/v5paper12.pdf (cit. on p. 33).

[55] Moritz Eysholdt and Heiko Behrens, “Xtext: implement your language faster than the quick and dirty way”, in: *Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, SPLASH/OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, ed. by William R. Cook, Siobhán Clarke, and Martin C. Rinard, ACM, 2010, pp. 307–309, ISBN: 978-1-4503-0240-1, DOI: 10.1145/1869542.1869625, URL: https://doi.org/10.1145/1869542.1869625 (cit. on pp. 18, 38, 101).

[56] Jean-Marie Favre, Jacky Estublier, and Mireille Blay-Fornarino, *L’ingénierie dirigée par les modèles: au-delà du MDA*, Hermes Science, 2006 (cit. on p. 11).

[57] Jean-Marie Favre et al., “Empirical Language Analysis in Software Linguistics”, in: *Software Language Engineering - Third International Conference, SLE 2010, Eindhoven, The Netherlands, October 12-13, 2010, Revised Selected Papers*, ed. by Brian A. Malloy, Steffen Staab, and Mark van den Brand, vol. 6563, Lecture Notes in Computer Science, Springer, 2010, pp. 316–326, ISBN: 978-3-642-19439-9, DOI: 10.1007/978-3-642-19440-5_21, URL: https://doi.org/10.1007/978-3-642-19440-5_21 (cit. on pp. i, 1, 27).

[58] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt, *Semantics Engineering with PLT Redex*, MIT Press, 2009, ISBN: 978-0-262-06275-6, URL: http://mitpress.mit.edu/catalog/item/default.asp?ttype=2%5C&tid=11885 (cit. on p. 29).

[59] Sébastien Ferré and Olivier Ridoux, “A Framework for Developing Embeddable Customized Logics”, in: *Logic Based Program Synthesis and Transformation*, Springer Berlin Heidelberg, 2002, pp. 191–215, DOI: 10.1007/3-540-45607-4_11, URL: https://doi.org/10.1007%2F3-540-45607-4_11 (cit. on p. 29).

[60] Bryan Ford, “Parsing expression grammars”, in: *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '04*, ACM Press, 2004, DOI: 10.1145/964001.964011, URL: https://doi.org/10.1145%2F964001.964011 (cit. on p. 28).

[61] Martin Fowler, *Language workbenches: The killer-app for domain specific languages?*, 2005, URL: https://martinfowler.com/articles/languageWorkbench.html (cit. on p. 22).

[62] Robert B. France and Bernhard Rumpe, “Model-driven Development of Complex Software: A Research Roadmap”, in: *International Conference on Software Engineering, ISCE 2007, Workshop on the Future of Software Engineering, FOSE 2007, May 23-25, 2007, Minneapolis, MN, USA*, ed. by Lionel C. Briand and Alexander L. Wolf, IEEE Computer Society, 2007, pp. 37–54, ISBN: 0-7695-2829-5, DOI: 10.1109/FOSE.2007.14, URL: https://doi.org/10.1109/FOSE.2007.14 (cit. on p. 11).

[63] Yoshihiko Futamura, “Partial Evaluation of Computation Process, Revisited”, in: *Higher Order Symbol. Comput.* 12.4 (Dec. 1999), pp. 377–380, ISSN: 1388-3690, DOI: 10.1023/A:1010043619517, URL: https://doi.org/10.1023/A:1010043619517 (cit. on p. 73).

[64] Erich Gamma, *Design patterns: elements of reusable object-oriented software*, Pearson Education India, 1995 (cit. on pp. 32, 38, 74, 100, 109, 112).

[65] Andy Georges, Dries Buytaert, and Lieven Eeckhout, “Statistically rigorous java performance evaluation”, in: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, 2007, pp. 57–76, DOI: 10.1145/1297027.1297033, URL: https://doi.org/10.1145/1297027.1297033 (cit. on p. 87).

[66] Maria Gouseti, Chiel Peters, and Tijs van der Storm, “Extensible language implementation with object algebras (short paper)”, in: *Generative Programming: Concepts and Experiences, GPCE'14, Vasteras, Sweden, September 15-16, 2014*, ed. by Ulrik Pagh Schultz and Matthew Flatt, ACM, 2014, pp. 25–28, ISBN: 978-1-4503-3161-6, DOI: 10.1145/2658761.2658765, URL: https://doi.org/10.1145/2658761.2658765 (cit. on p. 29).

[67] Clement Guy et al., “On Model Subtyping”, in: *Modelling Foundations and Applications - 8th European Conference, ECMFA 2012, Kgs. Lyngby, Denmark, July 2-5, 2012. Proceedings*, ed. by Antonio Vallecillo et al., vol. 7349, Lecture Notes in Computer Science, Springer, 2012, pp. 400–415, ISBN: 978-3-642-31490-2, DOI: 10.1007/978-3-642-31491-9_30, URL: https://doi.org/10.1007/978-3-642-31491-9_30 (cit. on p. 31).

[68] Michael Haupt et al., "The SOM family: virtual machines for teaching and research", in: *Proceedings of the 15th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, ITiCSE 2010, Bilkent, Ankara, Turkey, June 26-30, 2010*, ed. by Reyyan Ayfer, John Impagliazzo, and Cary Laxer, ACM, 2010, pp. 18–22, ISBN: 978-1-60558-729-5, DOI: 10.1145/1822090.1822098, URL: https://doi.org/10.1145/1822090.1822098 (cit. on p. 34).

[69] Jan Heering et al., "The syntax definition formalism SDF - reference manual", in: *SIGPLAN Notices* 24.*11* (1989), pp. 43–75, DOI: 10.1145/71605.71607, URL: https://doi.org/10.1145/71605.71607 (cit. on p. 18).

[70] Rajamohana Hegde and Naresh R. Shanbhag, "Energy-efficient signal processing via algorithmic noise-tolerance", in: *Proceedings of the 1999 International Symposium on Low Power Electronics and Design, 1999, San Diego, California, USA, August 16-17, 1999*, ed. by Farid N. Najm, Jason Cong, and David T. Blaauw, ACM, 1999, pp. 30–35, ISBN: 1-58113-133-X, DOI: 10.1145/313817.313834, URL: https://doi.org/10.1145/313817.313834 (cit. on p. 34).

[71] Robert Heim et al., "Compositional Language Engineering Using Generated, Extensible, Static Type-Safe Visitors", in: *Proceedings of the 12th European Conference on Modelling Foundations and Applications (ECMFA'16)*, Springer International Publishing, 2016, pp. 67–82, DOI: 10.1007/978-3-319-42061-5_5, URL: https://doi.org/10.1007%2F978-3-319-42061-5_5 (cit. on p. 35).

[72] Mark Hills et al., "A Case of Visitor versus Interpreter Pattern", in: *Proceedings of the 49th International Conference on Objects, Models, Components, Patterns (TOOLS'11)*, 2011, pp. 228–243, DOI: 10.1007/978-3-642-21952-8_17, URL: https://doi.org/10.1007%2F978-3-642-21952-8_17 (cit. on pp. 22, 46, 94).

[73] Urs Hölzle, Craig Chambers, and David M. Ungar, "Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches", in: *ECOOP*, vol. 512, Lecture Notes in Computer Science, Springer, 1991, pp. 21–38 (cit. on p. 78).

[74] John Hunt, "Cake Pattern", in: *Scala Design Patterns: Patterns for Practical Reuse and Design*, Cham: Springer International Publishing, 2013, pp. 115–119, ISBN: 978-3-319-02192-8, DOI: 10.1007/978-3-319-02192-8_13, URL: https://doi.org/10.1007/978-3-319-02192-8_13 (cit. on pp. 32, 38).

[75] John Edward Hutchinson, Jon Whittle, and Mark Rouncefield, "Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure", in: *Sci. Comput. Program.* 89 (2014), pp. 144–161, DOI: 10.1016/j.scico.2013.03.017, URL: https://doi.org/10.1016/j.scico.2013.03.017 (cit. on pp. i, 1).

[76] Pablo Inostroza and Tijs van der Storm, "Modular interpreters with implicit context propagation", in: *Computer Languages, Systems & Structures* 48 (2017), pp. 39–67, DOI: 10.1016/j.cl.2016.08.001, URL: https://doi.org/10.1016/j.cl.2016.08.001 (cit. on p. 32).

[77] MPS JetBrains, "Meta programming system", in: *URL http://www. jetbrains. com/mps* (2014) (cit. on pp. 31, 38).

[78] Jean-Marc Jézéquel, Benoît Combemale, and Didier Vojtisek, *Ingénierie Dirigée par les Modèles: des concepts à la pratique...* Ellipses, 2012 (cit. on p. 12).

[79] Jean-Marc Jézéquel et al., "Mashup of metalanguages and its implementation in the Kermeta language workbench", in: *Software & Systems Modeling* 14.2 (May 2015), pp. 905–920, ISSN: 1619-1374, DOI: 10.1007/s10270-013-0354-4, URL: https://doi.org/10.1007/s10270-013-0354-4 (cit. on pp. 32, 35, 38, 83, 109).

[80] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare, "Playing by the rules: rewriting as a practical optimisation technique in GHC", in: *Haskell workshop*, vol. 1, 2001, pp. 203–233 (cit. on p. 34).

[81] Ulrik Jørring and William L. Scherlis, "Compilers and Staging Transformations", in: *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '86, St. Petersburg Beach, Florida: ACM, 1986, pp. 86–96, DOI: 10.1145/512644.512652, URL: http://doi.acm.org/10.1145/512644.512652 (cit. on pp. i, 2).

[82] Tomas Kalibera, Lubomir Bulej, and Petr Tuma, "Benchmark precision and random initial state", in: *Proceedings of the 2005 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS 2005)*, 2005, pp. 484–490 (cit. on p. 87).

[83] Ted Kaminski and Eric Van Wyk, "Modular well-definedness analysis for attribute grammars", in: *Proceedings of the 5th International Conference on Software Language Engineering (SLE)*, vol. 7745, Lecture Notes in Computer Science, Springer Verlag, Sept. 2012, pp. 352–371, DOI: 10.1007/978-3-642-36089-3_20 (cit. on pp. 44, 46).

[84] Ted Kaminski et al., "Reliable and automatic composition of language extensions to C: the ableC extensible language framework", in: *PACMPL* 1.*OOPSLA* (2017), 98:1–98:29, DOI: 10.1145/3138224, URL: https://doi.org/10.1145/3138224 (cit. on pp. 30, 32, 38, 44, 46, 54).

[85] Lennart C. L. Kats and Eelco Visser, "The spoofax language workbench: rules for declarative specification of languages and IDEs", in: *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, ed. by William R. Cook, Siobhán Clarke, and Martin C. Rinard,

ACM, 2010, pp. 444–463, ISBN: 978-1-4503-0203-6, DOI: 10.1145/1869459.1869497, URL: https://doi.org/10.1145/1869459.1869497 (cit. on pp. 18, 23, 36).

[86] Gregor Kiczales, Jim Des Rivieres, and Daniel Gureasko Bobrow, *The art of the metaobject protocol*, MIT press, 1991 (cit. on p. 31).

[87] Gregor Kiczales et al., "Metaobject protocols: Why we want them and what else they can do", in: *Object-Oriented Programming: The CLOS Perspective* (1993), pp. 101–118 (cit. on p. 31).

[88] Paul Klint, Ralf Lämmel, and Chris Verhoef, "Toward an engineering discipline for grammarware", in: *ACM Trans. Softw. Eng. Methodol.* 14.*3* (2005), pp. 331–380, DOI: 10.1145/1072997.1073000, URL: https://doi.org/10.1145/1072997.1073000 (cit. on p. 20).

[89] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju, "RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation", in: *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2009, Edmonton, Alberta, Canada, September 20-21, 2009*, IEEE Computer Society, 2009, pp. 168–177, ISBN: 978-0-7695-3793-1, DOI: 10.1109/SCAM.2009.28, URL: https://doi.org/10.1109/SCAM.2009.28 (cit. on pp. 18, 23, 36, 38, 44).

[90] Donald Ervin Knuth, *The art of computer programming: sorting and searching*, vol. 3, Pearson Education, 1997 (cit. on p. 103).

[91] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack, "The epsilon object language (EOL)", in: *European Conference on Model Driven Architecture-Foundations and Applications*, Springer, 2006, pp. 128–142 (cit. on p. 83).

[92] Tomaz 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, DOI: 10.1007/s10664-017-9593-2, URL: https://doi.org/10.1007/s10664-017-9593-2 (cit. on p. 17).

[93] Holger Krahn, Bernhard Rumpe, and Steven Völkel, "MontiCore: a framework for compositional development of domain specific languages", in: *International Journal on Software Tools for Technology Transfer* 12.*5* (Sept. 2010), pp. 353–372, ISSN: 1433-2787, DOI: 10.1007/s10009-010-0142-1, URL: https://doi.org/10.1007/s10009-010-0142-1 (cit. on pp. 23, 35, 38).

[94] Thomas Kühn, Walter Cazzola, and Diego Mathias Olivares, "Choosy and picky: configuration of language product lines", in: *Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015*, ed. by Douglas C. Schmidt, ACM, 2015, pp. 71–80, ISBN: 978-1-4503-3613-0, DOI: 10.1145/2791060.2791092, URL: https://doi.org/10.1145/2791060.2791092 (cit. on pp. 29, 30).

[95] Andreas Kunert, "Semi-automatic Generation of Metamodels and Models From Grammars and Programs", in: *Electr. Notes Theor. Comput. Sci.* 211 (2008), pp. 111–119, DOI: 10.1016/j.entcs.2008.04.034, URL: https://doi.org/10.1016/j.entcs.2008.04.034 (cit. on p. 19).

[96] Ivan Kurtev, Jean Bézivin, and Mehmet Aksit, "Technological spaces: An initial appraisal", in: *CoopIS, DOA* 2002 (2002) (cit. on p. 12).

[97] R. Lammel and C. Verhoef, "Cracking the 500-language problem", in: *IEEE Software* 18.6 (Nov. 2001), pp. 78–88, ISSN: 0740-7459, DOI: 10.1109/52.965809 (cit. on p. 15).

[98] Ralf Lämmel, *Software Languages*, Springer International Publishing, 2018, DOI: 10.1007/978-3-319-90800-7, URL: https://doi.org/10.1007%2F978-3-319-90800-7 (cit. on p. 15).

[99] Juan de Lara and Esther Guerra, "From types to type requirements: genericity for model-driven engineering", in: *Software & Systems Modeling* 12.3 (July 2013), pp. 453–474, ISSN: 1619-1374, DOI: 10.1007/s10270-011-0221-0, URL: https://doi.org/10.1007/s10270-011-0221-0 (cit. on p. 47).

[100] Juan de Lara, Esther Guerra, and Jesús Sánchez Cuadrado, "A-posteriori typing for Model-Driven Engineering", in: *18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MoDELS 2015, Ottawa, ON, Canada, September 30 - October 2, 2015*, ed. by Timothy Lethbridge, Jordi Cabot, and Alexander Egyed, IEEE Computer Society, 2015, pp. 156–165, ISBN: 978-1-4673-6908-4, DOI: 10.1109/MODELS.2015.7338246, URL: https://doi.org/10.1109/MODELS.2015.7338246 (cit. on pp. 31, 38).

[101] Juan de Lara and Hans Vangheluwe, "AToM^3: A Tool for Multi-formalism and Meta-modelling", in: *Fundamental Approaches to Software Engineering, 5th International Conference, FASE 2002, held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, ed. by Ralf-Detlef Kutsche and Herbert Weber, vol. 2306, Lecture Notes in Computer Science, Springer, 2002, pp. 174–188, ISBN: 3-540-43353-8, DOI: 10.1007/3-540-45923-5_12, URL: https://doi.org/10.1007/3-540-45923-5_12 (cit. on p. 12).

[102] A. Ledeczi et al., "On metamodel composition", in: *Proceedings of the 2001 IEEE International Conference on Control Applications (CCA'01) (Cat. No.01CH37204)*, Sept. 2001, pp. 756–760, DOI: 10.1109/CCA.2001.973959 (cit. on p. 30).

[103] Manuel Leduc, Thomas Degueule, and Benoît Combemale, "Modular language composition for the masses", in: *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2018, Boston, MA, USA, November 05-06, 2018*, ed. by David Pearce, Tanja Mayerhofer, and Friedrich Steimann, ACM, 2018, pp. 47–59, ISBN: 978-1-4503-6029-6, DOI: 10.1145/

3276604.3276622, URL: https://doi.org/10.1145/3276604.3276622 (cit. on pp. iv, 4, 7, 43, 46).

[104] Manuel Leduc et al., "Revisiting Visitors for Modular Extension of Executable DSMLs", in: *20th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2017, Austin, TX, USA, September 17-22, 2017*, IEEE Computer Society, 2017, pp. 112–122, ISBN: 978-1-5386-3492-9, DOI: 10.1109/MODELS.2017.23, URL: https://doi.org/10.1109/MODELS.2017.23 (cit. on pp. iv, 4, 7, 29, 43, 44, 63, 109).

[105] Manuel Leduc et al., "The Software Language Extension Problem", in: *Software and Systems Modeling* (2019), pp. 1–4, URL: https://hal.inria.fr/hal-02399166 (cit. on p. 7).

[106] Sheng Liang and Paul Hudak, "Modular Denotational Semantics for Compiler Construction", in: *Programming Languages and Systems - ESOP'96, 6th European Symposium on Programming, Linköping, Sweden, April 22-24, 1996, Proceedings*, ed. by Hanne Riis Nielson, vol. 1058, Lecture Notes in Computer Science, Springer, 1996, pp. 219–234, ISBN: 3-540-61055-3, DOI: 10.1007/3-540-61055-3_39, URL: https://doi.org/10.1007/3-540-61055-3_39 (cit. on p. 29).

[107] Sheng Liang, Paul Hudak, and Mark P. Jones, "Monad Transformers and Modular Interpreters", in: *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, ed. by Ron K. Cytron and Peter Lee, ACM Press, 1995, pp. 333–343, ISBN: 0-89791-692-1, DOI: 10.1145/199448.199528, URL: https://doi.org/10.1145/199448.199528 (cit. on pp. 32, 38).

[108] Lionello A. Lombardi, "Incremental Computation: The Preliminary Design of a Programming System Which Allows for Incremental Data Assimilation in Open-Ended Man-Computer Information Systems", in: Advances in Computers 8 (1967), ed. by Franz L. Alt and Morris Rubinoff, pp. 247–333, ISSN: 0065-2458, DOI: https://doi.org/10.1016/S0065-2458(08)60698-1, URL: http://www.sciencedirect.com/science/article/pii/S0065245808606981 (cit. on p. 73).

[109] Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck, "Cross-language compiler benchmarking: are we fast yet?", in: *Proceedings of the 12th Symposium on Dynamic Languages, DLS 2016, Amsterdam, The Netherlands, November 1, 2016*, ed. by Roberto Ierusalimschy, ACM, 2016, pp. 120–131, ISBN: 978-1-4503-4445-6, DOI: 10.1145/2989225.2989232, URL: https://doi.org/10.1145/2989225.2989232 (cit. on p. 73).

[110] James Martin, *Principles of object-oriented analysis and design*, Prentice-Hall, Inc., 1993 (cit. on p. 112).

[111] Tanja Mayerhofer and Manuel Wimmer, “The TTC 2015 Model Execution Case”, in: *Proceedings of the 8th Transformation Tool Contest (TTC'15)*, vol. 1524, CEUR Workshop Proceedings, 2015, pp. 2–18 (cit. on pp. iv, 92).

[112] David Méndez-Acuña et al., “Leveraging Software Product Lines Engineering in the development of external DSLs: A systematic literature review”, in: *Comput. Lang. Syst. Struct.* 46.*C* (Nov. 2016), pp. 206–235, ISSN: 1477-8424, DOI: 10.1016/j.cl.2016.09.004, URL: https://doi.org/10.1016/j.cl.2016.09.004 (cit. on p. 29).

[113] David Méndez-Acuña et al., “Reverse-engineering reusable language modules from legacy domain-specific languages”, in: *International Conference on Software Reuse*, Springer, 2016, pp. 368–383 (cit. on p. 35).

[114] Marjan Mernik, “An object-oriented approach to language compositions for software language engineering”, in: *Journal of Systems and Software* 86.*9* (2013), pp. 2451–2464, DOI: 10.1016/j.jss.2013.04.087, URL: https://doi.org/10.1016/j.jss.2013.04.087 (cit. on pp. 2, 35, 46).

[115] Marjan Mernik, Jan Heering, and Anthony M. Sloane, “When and how to develop domain-specific languages”, in: *ACM Computing Surveys* 37.*4* (Dec. 2005), pp. 316–344, ISSN: 03600300, DOI: 10.1145/1118890.1118892, URL: http://portal.acm.org/citation.cfm?doid=1118890.1118892 (cit. on p. 22).

[116] Marjan Mernik et al., “LISA: An Interactive Environment for Programming Language Development”, in: *Compiler Construction*, ed. by R. Nigel Horspool, Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 1–4, ISBN: 978-3-540-45937-8, DOI: 10.1007/3-540-45937-5_1, URL: https://doi.org/10.1007%2F3-540-45937-5_1 (cit. on pp. 23, 35, 38).

[117] Parastoo Mohagheghi et al., “An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases”, in: *Empirical Software Engineering* 18.*1* (2013), pp. 89–116, DOI: 10.1007/s10664-012-9196-x, URL: https://doi.org/10.1007/s10664-012-9196-x (cit. on pp. i, 1).

[118] Peter D. Mosses, “Modular structural operational semantics”, in: *J. Log. Algebr. Program.* 60-61 (2004), pp. 195–228, DOI: 10.1016/j.jlap.2004.03.008, URL: https://doi.org/10.1016/j.jlap.2004.03.008 (cit. on pp. 29, 31).

[119] Peter D. Mosses and Mark J. New, “Implicit Propagation in Structural Operational Semantics”, in: *Electron. Notes Theor. Comput. Sci.* 229.*4* (Aug. 2009), pp. 49–66, ISSN: 1571-0661, DOI: 10.1016/j.entcs.2009.07.073, URL: http://dx.doi.org/10.1016/j.entcs.2009.07.073 (cit. on p. 31).

[120] Peter D Mosses and Ferdinand Vesely, “Funkons: Component-based semantics in K”, in: *International Workshop on Rewriting Logic and its Applications*, Springer, 2014, pp. 213–229, DOI: 10.1007/978-3-319-12904-4_12, URL: https://doi.org/10.1007%2F978-3-319-12904-4_12 (cit. on p. 32).

[121] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers, "Polyglot: An Extensible Compiler Framework for Java", in: *Compiler Construction, 12th International Conference, CC 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, ed. by Görel Hedin, vol. 2622, Lecture Notes in Computer Science, Springer, 2003, pp. 138–152, ISBN: 3-540-00904-3, DOI: 10.1007/3-540-36579-6_11, URL: https://doi.org/10.1007/3-540-36579-6_11 (cit. on p. 34).

[122] Martin Odersky et al., *An overview of the Scala programming language*, tech. rep., 2004 (cit. on p. 19).

[123] Bruno C. d. S. Oliveira, "Modular Visitor Components", in: *ECOOP 2009 – Object-Oriented Programming*, ed. by Sophia Drossopoulou, Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 269–293, ISBN: 978-3-642-03013-0, DOI: 10.1007/978-3-642-03013-0_13, URL: https://doi.org/10.1007%2F978-3-642-03013-0_13 (cit. on pp. 32, 38, 44).

[124] Bruno C. d. S. Oliveira and William R. Cook, "Extensibility for the Masses - Practical Extensibility with Object Algebras", in: *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*, ed. by James Noble, vol. 7313, Lecture Notes in Computer Science, Springer, 2012, pp. 2–27, ISBN: 978-3-642-31056-0, DOI: 10.1007/978-3-642-31057-7_2, URL: https://doi.org/10.1007/978-3-642-31057-7_2 (cit. on pp. 32, 38, 43, 55, 68, 74).

[125] Cyrus Omar et al., "Safely Composable Type-Specific Languages", in: (), URL: https://www.cs.cmu.edu/%7B~%7Daldrich/papers/ecoop14-tsls.pdf (cit. on p. 54).

[126] OMG, *2.0 Object Constraint Language (OCL) Final Adopted specification, Object Management Group*, 2003, URL: http://www.omg.org/cgi-bin/doc?ptc/2003-10-14 (cit. on pp. 14, 83).

[127] OMG, *Meta Object Facility (MOF) 2.0 Core Specification*, http://www.omg.org/spec/MOF/2.0/, 2006 (cit. on p. 12).

[128] OracleLabs, *GraalVM*, 2019, URL: https://www.graalvm.org (visited on 03/30/2019) (cit. on p. 71).

[129] Terence John Parr and Russell W. Quong, "ANTLR: A Predicated- *LL(k)* Parser Generator", in: *Softw., Pract. Exper.* 25.7 (1995), pp. 789–810, DOI: 10.1002/spe.4380250705, URL: https://doi.org/10.1002/spe.4380250705 (cit. on pp. 18, 19).

[130] Terence Parr and Kathleen Fisher, "LL(*): the foundation of the ANTLR parser generator", in: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, ed. by Mary W. Hall and David A. Padua, ACM, 2011, pp. 425–436, ISBN: 978-1-4503-0663-8, DOI: 10.1145/1993498.1993548, URL: https://doi.org/10.1145/1993498.1993548 (cit. on pp. 30, 38).

[131] David Pearce, Tanja Mayerhofer, and Friedrich Steimann, eds., *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2018, Boston, MA, USA, November 05-06, 2018*, ACM, 2018, ISBN: 978-1-4503-6029-6, DOI: 10.1145/3276604, URL: https://doi.org/10.1145/3276604.

[132] Ana Pescador et al., "Pattern-based development of Domain-Specific Modelling Languages", in: *18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MoDELS 2015, Ottawa, ON, Canada, September 30 - October 2, 2015*, ed. by Timothy Lethbridge, Jordi Cabot, and Alexander Egyed, IEEE Computer Society, 2015, pp. 166–175, ISBN: 978-1-4673-6908-4, DOI: 10.1109/MODELS.2015.7338247, URL: https://doi.org/10.1109/MODELS.2015.7338247 (cit. on p. 28).

[133] Sophie Pinchinat, Mathieu Acher, and Didier Vojtisek, "ATSyRa: An Integrated Environment for Synthesizing Attack Trees - (Tool Paper)", in: *Graphical Models for Security - Second International Workshop, GraMSec 2015, Verona, Italy, July 13, 2015, Revised Selected Papers*, ed. by Sjouke Mauw, Barbara Kordy, and Sushil Jajodia, vol. 9390, Lecture Notes in Computer Science, Springer, 2015, pp. 97–101, ISBN: 978-3-319-29967-9, DOI: 10.1007/978-3-319-29968-6_7, URL: https://doi.org/10.1007/978-3-319-29968-6_7 (cit. on p. 16).

[134] Klaus Pohl, Günter Böckle, and Frank J van Der Linden, *Software product line engineering: foundations, principles and techniques*, Springer Science & Business Media, 2005, DOI: 10.1007/3-540-28901-1, URL: https://doi.org/10.1007%2F3-540-28901-1 (cit. on p. 29).

[135] Mitchel Resnick et al., "Scratch: programming for all", in: *Commun. ACM* 52.*11* (2009), pp. 60–67, DOI: 10.1145/1592761.1592779, URL: https://doi.org/10.1145/1592761.1592779 (cit. on p. 20).

[136] Eric Roberts, "An overview of MiniJava", in: *Proceedings of the 32rd SIGCSE Technical Symposium on Computer Science Education, 2001, Charlotte, North Carolina, USA, 2001*, ed. by Henry MacKay Walker et al., ACM, 2001, pp. 1–5, ISBN: 1-58113-329-4, DOI: 10.1145/364447.364525, URL: https://doi.org/10.1145/364447.364525 (cit. on pp. i, 5, 102).

[137] Marcelino Rodriguez-Cancio, Benoît Combemale, and Benoit Baudry, "Approximate loop unrolling", in: *Proceedings of the 16th ACM International Conference on Computing Frontiers, CF 2019, Alghero, Italy, April 30 - May 2, 2019*. Ed. by Francesca Palumbo et al., ACM, 2019, pp. 94–105, ISBN: 978-1-4503-6685-4, DOI: 10.1145/3310273.3323841, URL: https://doi.org/10.1145/3310273.3323841 (cit. on p. 34).

[138] Tiark Rompf and Nada Amin, "Functional pearl: a SQL to C compiler in 500 lines of code", in: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, ed. by Kathleen Fisher and John H. Reppy, ACM, 2015, pp. 2–9, ISBN: 978-1-4503-3669-7, DOI: 10.1145/2784731.2784760, URL: https://doi.org/10.1145/2784731.2784760 (cit. on p. 34).

[139] Tiark Rompf and Martin Odersky, "Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs", in: *Generative Programming And Component Engineering, Proceedings of the Ninth International Conference on Generative Programming and Component Engineering, GPCE 2010, Eindhoven, The Netherlands, October 10-13, 2010*, ed. by Eelco Visser and Jaakko Järvi, ACM, 2010, pp. 127–136, ISBN: 978-1-4503-0154-1, DOI: 10.1145/1868294.1868314, URL: https://doi.org/10.1145/1868294.1868314 (cit. on pp. 32, 33, 38).

[140] Tiark Rompf et al., "Go Meta! A Case for Generative Programming and DSLs in Performance Critical Systems", in: *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA*, ed. by Thomas Ball et al., vol. 32, LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015, pp. 238–261, ISBN: 978-3-939897-80-4, DOI: 10.4230/LIPIcs.SNAPL.2015.238, URL: https://doi.org/10.4230/LIPIcs.SNAPL.2015.238 (cit. on p. 34).

[141] Grigore Rosu and Traian-Florin Serbanuta, "An overview of the K semantic framework", in: *J. Log. Algebr. Program.* 79.6 (2010), pp. 397–434, DOI: 10.1016/j.jlap.2010.03.012, URL: https://doi.org/10.1016/j.jlap.2010.03.012 (cit. on p. 29).

[142] Bruno C. d. S. Oliveira et al., "Feature-Oriented Programming with Object Algebras", in: *ECOOP 2013 - Object-Oriented Programming - 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings*, ed. by Giuseppe Castagna, vol. 7920, Lecture Notes in Computer Science, Springer, 2013, pp. 27–51, ISBN: 978-3-642-39037-1, DOI: 10.1007/978-3-642-39038-8_2, URL: https://doi.org/10.1007/978-3-642-39038-8_2 (cit. on p. 32).

[143] Douglas C Schmidt, "Model-driven engineering", in: *COMPUTER-IEEE COMPUTER SOCIETY-* 39.2 (2006), p. 25 (cit. on p. 1).

[144] Matthias Schöttle et al., "Feature modelling and traceability for concern-driven software development with TouchCORE", in: *Companion Proceedings of the 14th International Conference on Modularity, MODULARITY 2015, Fort Collins, CO, USA, March 16 - 19, 2015*, ed. by Robert B. France, Sudipto Ghosh, and Gary T. Leavens, ACM, 2015, pp. 11–14, ISBN: 978-1-4503-3283-5, DOI: 10.1145/2735386.2735922, URL: https://doi.org/10.1145/2735386.2735922 (cit. on p. 20).

[145] August Schwerdfeger and Eric Van Wyk, "Verifiable Parse Table Composition for Deterministic Parsing", in: *Software Language Engineering, Second International Conference, SLE 2009, Denver, CO, USA, October 5-6, 2009, Revised Selected Papers*, ed. by Mark van den Brand, Dragan Gasevic, and Jeff Gray, vol. 5969, Lecture Notes in Computer Science, Springer, 2009, pp. 184–203, ISBN: 978-3-642-12106-7, DOI: 10.1007/978-3-642-12107-4_15, URL: https://doi.org/10.1007/978-3-642-12107-4_15 (cit. on p. 30).

[146] Amir Shaikhha, Vojin Jovanovic, and Christoph E. Koch, "Compiler Generation for Performance-oriented Embedded DSLs (Short Paper)", in: *Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2019, Athens, Greece: ACM, 2019, pp. 94–101, ISBN: 978-1-4503-6980-0, DOI: 10.1145/3357765.3359520, URL: http://doi.acm.org/10.1145/3357765.3359520 (cit. on pp. 34, 38).

[147] Diomidis Spinellis, "Notable design patterns for domain-specific languages", in: *Journal of Systems and Software* 56.*1* (2001), pp. 91–99, DOI: 10.1016/S0164-1212(00)00089-3, URL: https://doi.org/10.1016/S0164-1212(00)00089-3 (cit. on p. 22).

[148] Lukas Stadler et al., "Optimizing R language execution via aggressive speculation", in: *Proceedings of the 12th Symposium on Dynamic Languages, DLS 2016, Amsterdam, The Netherlands, November 1, 2016*, 2016, pp. 84–95, DOI: 10.1145/2989225.2989236, URL: https://doi.org/10.1145/2989225.2989236 (cit. on p. 73).

[149] Jim Steel and Jean-Marc Jézéquel, "On model typing", in: *Software and System Modeling* 6.*4* (2007), pp. 401–413, DOI: 10.1007/s10270-006-0036-6, URL: https://doi.org/10.1007/s10270-006-0036-6 (cit. on p. 31).

[150] Dave Steinberg et al., *EMF: Eclipse Modeling Framework*, Pearson Education, 2008 (cit. on pp. iv, 4, 12, 51, 63, 101, 109).

[151] Tijs van der Storm, William R. Cook, and Alex Loh, "The Design and Implementation of Object Grammars", in: *Sci. Comput. Program.* 96.*P4* (Dec. 2014), pp. 460–487, ISSN: 0167-6423, DOI: 10.1016/j.scico.2014.02.023, URL: http://dx.doi.org/10.1016/j.scico.2014.02.023 (cit. on pp. 30, 38).

[152] Ana-Maria Sutîi, "MetaMod: A Modeling Formalism with Modularity at Its Core", in: *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, ed. by Myra B. Cohen, Lars Grunske, and Michael Whalen, IEEE Computer Society, 2015, pp. 890–893, ISBN: 978-1-5090-0025-8, DOI: 10.1109/ASE.2015.29, URL: https://doi.org/10.1109/ASE.2015.29 (cit. on pp. 31, 38).

[153] Mikael Svahnberg, Jilles van Gurp, and Jan Bosch, "A taxonomy of variability realization techniques", in: *Softw., Pract. Exper.* 35.*8* (2005), pp. 705–754, DOI: 10.1002/spe.652, URL: https://doi.org/10.1002/spe.652 (cit. on p. 28).

[154] Wouter Swierstra, "Data types à la carte", in: *J. Funct. Program.* 18.*4* (2008), pp. 423–436, DOI: 10.1017/S0956796808006758, URL: https://doi.org/10.1017/S0956796808006758 (cit. on pp. 32, 38).

[155] Mads Torgersen, "The Expression Problem Revisited", in: *ECOOP 2004 – Object-Oriented Programming*, ed. by Martin Odersky, Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 123–146, ISBN: 978-3-540-24851-4, DOI: 10.1007/978-3-540-24851-4_6, URL: https://doi.org/10.1007%2F978-3-540-24851-4_6 (cit. on pp. 26, 33, 38, 46).

[156] Maxime Tricoire et al., "KevoreeJS: Enabling Dynamic Software Reconfigurations in the Browser", in: *19th International ACM SIGSOFT Symposium on Component-Based Software Engineering, CBSE 2016, Venice, Italy, April 5-8, 2016*, IEEE Computer Society, 2016, pp. 49–58, ISBN: 978-1-5090-2569-5, DOI: 10.1109/CBSE.2016.20, URL: https://doi.org/10.1109/CBSE.2016.20 (cit. on p. 8).

[157] Edoardo Vacchi and Walter Cazzola, "Neverlang: A framework for feature-oriented language development", in: *Computer Languages, Systems & Structures* 43 (Oct. 2015), pp. 1–40, DOI: 10.1016/j.cl.2015.02.001, URL: https://doi.org/10.1016%2Fj.cl.2015.02.001 (cit. on pp. 23, 46).

[158] Edoardo Vacchi et al., "Neverlang 2: a framework for modular language implementation", in: *13th International Conference on Modularity, MODULARITY '14, Lugano, Switzerland, April 22-26, 2014*, ed. by Walter Binder et al., ACM, 2014, pp. 29–32, ISBN: 978-1-4503-2772-5, DOI: 10.1145/2584469.2584478, URL: https://doi.org/10.1145/2584469.2584478 (cit. on pp. 36, 38).

[159] Antonio Vallecillo, "On the Combination of Domain Specific Modeling Languages", in: *Modelling Foundations and Applications, 6th European Conference, ECMFA 2010, Paris, France, June 15-18, 2010. Proceedings*, ed. by Thomas Kühne et al., vol. 6138, Lecture Notes in Computer Science, Springer, 2010, pp. 305–320, ISBN: 978-3-642-13594-1, DOI: 10.1007/978-3-642-13595-8_24, URL: https://doi.org/10.1007/978-3-642-13595-8_24 (cit. on p. 30).

[160] Raja Vallée-Rai et al., “Soot - a Java Bytecode Optimization Framework”, in: *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON ’99, Mississauga, Ontario, Canada: IBM Press, 1999, pp. 13–, URL: http://dl.acm.org/citation.cfm?id=781995.782008 (cit. on p. 74).

[161] Vlad A. Vergu, Pierre Neron, and Eelco Visser, “DynSem: A DSL for Dynamic Semantics Specification”, in: *26th International Conference on Rewriting Techniques and Applications, RTA 2015, June 29 to July 1, 2015, Warsaw, Poland*, ed. by Maribel Fernández, vol. 36, LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015, pp. 365–378, ISBN: 978-3-939897-85-9, DOI: 10.4230/LIPIcs.RTA.2015.365, URL: https://doi.org/10.4230/LIPIcs.RTA.2015.365 (cit. on pp. 29, 36).

[162] Vlad A. Vergu, Andrew Tolmach, and Eelco Visser, “Scopes and Frames Improve Meta-Interpreter Specialization”, in: *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom*. Ed. by Alastair F. Donaldson, vol. 134, LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019, 4:1–4:30, ISBN: 978-3-95977-111-5, DOI: 10.4230/LIPIcs.ECOOP.2019.4, URL: https://doi.org/10.4230/LIPIcs.ECOOP.2019.4 (cit. on pp. 34, 38).

[163] Eelco Visser, *Scannerless generalized-LR parsing*, Universiteit van Amsterdam. Programming Research Group, 1997 (cit. on p. 28).

[164] Markus Voelter, “Language and IDE Modularization and Composition with MPS”, in: (2013), ed. by Ralf Lämmel, João Saraiva, and Joost Visser, pp. 383–430, DOI: 10.1007/978-3-642-35992-7_11, URL: https://doi.org/10.1007/978-3-642-35992-7_11 (cit. on pp. 29, 46, 54).

[165] Markus Voelter, “The Design, Evolution, and Use of KernelF - An Extensible and Embeddable Functional Language”, in: *Theory and Practice of Model Transformation - 11th International Conference, ICMT 2018, Held as Part of STAF 2018, Toulouse, France, June 25-26, 2018, Proceedings*, ed. by Arend Rensink and Jesús Sánchez Cuadrado, vol. 10888, Lecture Notes in Computer Science, Springer, 2018, pp. 3–55, ISBN: 978-3-319-93316-0, DOI: 10.1007/978-3-319-93317-7_1, URL: https://doi.org/10.1007/978-3-319-93317-7_1 (cit. on p. 36).

[166] Markus Voelter et al., “Lessons learned from developing mbeddr: a case study in language engineering with MPS”, in: *Software and Systems Modeling* 18.*1* (2019), pp. 585–630, DOI: 10.1007/s10270-016-0575-4, URL: https://doi.org/10.1007/s10270-016-0575-4 (cit. on p. 20).

[167] Markus Voelter et al., “mbeddr: an extensible C-based programming language and IDE for embedded systems”, in: *Conference on Systems, Programming, and Applications: Software for Humanity, SPLASH ’12, Tucson, AZ, USA, October 21-25, 2012*, ed. by Gary T. Leavens, ACM, 2012, pp. 121–140, ISBN: 978-1-4503-

1563-0, DOI: 10.1145/2384716.2384767, URL: https://doi.org/10.1145/2384716.2384767 (cit. on p. 36).

[168] Guido Wachsmuth, Gabriël D. P. Konat, and Eelco Visser, "Language Design with the Spoofax Language Workbench", in: *IEEE Software* 31.5 (2014), pp. 35–43, DOI: 10.1109/MS.2014.100, URL: https://doi.org/10.1109/MS.2014.100 (cit. on pp. 38, 46).

[169] Philip Wadler, *The Expression Problem*, http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt, Nov. 1998, URL: http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt (cit. on pp. 26, 43, 45).

[170] Yanlin Wang and Bruno C. d. S. Oliveira, "The Expression Problem, Trivially!", in: *Proceedings of the 15th International Conference on Modularity*, MODULARITY 2016, Málaga, Spain: ACM, 2016, pp. 37–41, ISBN: 978-1-4503-3995-7, DOI: 10.1145/2889443.2889448, URL: http://doi.acm.org/10.1145/2889443.2889448 (cit. on pp. 33, 38).

[171] Christian Wende, Nils Thieme, and Steffen Zschaler, "A Role-Based Approach towards Modular Language Engineering", in: *Software Language Engineering, Second International Conference, SLE 2009, Denver, CO, USA, October 5-6, 2009, Revised Selected Papers*, ed. by Mark van den Brand, Dragan Gasevic, and Jeff Gray, vol. 5969, Lecture Notes in Computer Science, Springer, 2009, pp. 254–273, ISBN: 978-3-642-12106-7, DOI: 10.1007/978-3-642-12107-4_19, URL: https://doi.org/10.1007/978-3-642-12107-4_19 (cit. on p. 31).

[172] Christian Wimmer and Thomas Würthinger, "Truffle: a self-optimizing runtime system", in: *Conference on Systems, Programming, and Applications: Software for Humanity, SPLASH '12, Tucson, AZ, USA, October 21-25, 2012*, ed. by Gary T. Leavens, ACM, 2012, pp. 13–14, ISBN: 978-1-4503-1563-0, DOI: 10.1145/2384716.2384723, URL: https://doi.org/10.1145/2384716.2384723 (cit. on pp. 71, 73).

[173] Niklaus Wirth, *Compiler construction*, International computer science series, Addison-Wesley, 1996, ISBN: 978-0-201-40353-4 (cit. on p. 36).

[174] Niklaus Wirth, "Extended backus-naur form (ebnf)", in: *Iso/Iec* 14977 (1996), p. 2996 (cit. on p. 20).

[175] Niklaus Wirth, "What Can We Do about the Unnecessary Diversity of Notation for Syntactic Definitions?", in: *Commun. ACM* 20.11 (1977), pp. 822–823, DOI: 10.1145/359863.359883, URL: https://doi.org/10.1145/359863.359883 (cit. on p. 18).

[176] Thomas Würthinger et al., "One VM to rule them all", in: *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*, ed. by Antony L. Hosking, Patrick Th. Eugster, and Robert Hirschfeld, ACM, 2013, pp. 187–204, ISBN: 978-1-4503-2472-4, DOI: 10.1145/2509578.2509581, URL: https://doi.org/10.1145/2509578.2509581 (cit. on pp. 34, 38, 72).

[177] Thomas Würthinger et al., "Practical partial evaluation for high-performance dynamic language runtimes", in: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, ed. by Albert Cohen and Martin T. Vechev, ACM, 2017, pp. 662–676, ISBN: 978-1-4503-4988-8, DOI: 10.1145/3062341.3062381, URL: https://doi.org/10.1145/3062341.3062381 (cit. on pp. 71, 73).

[178] Thomas Würthinger et al., "Self-optimizing AST Interpreters", in: *Proceedings of the 8th Symposium on Dynamic Languages*, DLS '12, Tucson, Arizona, USA: ACM, 2012, pp. 73–82, ISBN: 978-1-4503-1564-7, DOI: 10.1145/2384577.2384587, URL: http://doi.acm.org/10.1145/2384577.2384587 (cit. on pp. 34, 38, 74, 78).

[179] Eric Van Wyk et al., "Silver: An extensible attribute grammar system", in: *Science of Computer Programming* 75.*1* (2010), Special Issue on ETAPS 2006 and 2007 Workshops on Language Descriptions, Tools, and Applications (LDTA '06 and '07), pp. 39–54, ISSN: 0167-6423, DOI: https://doi.org/10.1016/j.scico.2009.07.004, URL: http://www.sciencedirect.com/science/article/pii/S0167642309001099 (cit. on pp. 28, 44).

[180] M. Zenger and M. Odersky, "Independently extensible solutions to the expression problem", in: *12th International Workshop on Foundations of Object-Oriented Languages (FOOL'05)*, ACM, 2005 (cit. on pp. 26, 43, 46).

[181] Weixin Zhang and Bruno C. d. S. Oliveira, "EVF: An Extensible and Expressive Visitor Framework for Programming Language Reuse", in: *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*, ed. by Peter Müller, vol. 74, LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017, 29:1–29:32, ISBN: 978-3-95977-035-4, DOI: 10.4230/LIPIcs.ECOOP.2017.29, URL: https://doi.org/10.4230/LIPIcs.ECOOP.2017.29 (cit. on pp. 32, 38, 46).

[182] Srdjan Zivkovic and Dimitris Karagiannis, "Towards Metamodelling-In-The-Large: Interface-Based Composition for Modular Metamodel Development", in: *Enterprise, Business-Process and Information Systems Modeling - 16th International Conference, BPMDS 2015, 20th International Conference, EMMSAD 2015, Held at CAiSE 2015, Stockholm, Sweden, June 8-9, 2015, Proceedings*, ed. by Khaled Gaaloul et al., vol. 214, Lecture Notes in Business Information Processing, Springer, 2015, pp. 413–428, ISBN: 978-3-319-19236-9, DOI: 10.1007/978-3-

319-19237-6_26, URL: https://doi.org/10.1007/978-3-319-19237-6_26 (cit. on p. 31).

[183] Steffen Zschaler et al., "VML* – a Family of Languages for Variability Management in Software Product Lines", in: *Proceedings of the Second International Conference on Software Language Engineering*, SLE'09, Denver, CO: Springer-Verlag, 2010, pp. 82–102, ISBN: 3-642-12106-3, 978-3-642-12106-7, DOI: 10.1007/978-3-642-12107-4_7, URL: https://doi.org/10.1007/978-3-642-12107-4_7 (cit. on p. 30).

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

Titre: Modularité et performances des implémentations de langages dédiés externes

Mot clés : réutilisation, performances, langages dédiés, ingénierie des langages

Resumé : L'Ingénierie Dirigée par les Modèles (IDM) a pour but d'assister les experts métiers dans le développement de systèmes complexes, en séparant les préoccupations par l'utilisation de modèles. Les modèles sont des représentations d'aspects spécifiques des systèmes et sont définis à l'aide d'abstractions dédiées. Ces abstractions sont définies avec des langages dédiés. Ces langages dédiés sont créés à l'aide d'Environnements de Développement Intégré (EDI) spécialisés, appelés language workbenches.[1] Les language workbenches aident à l'ingénierie de langages dédiés en proposant des abstractions utiles. Premièrement, ces abstractions sont dédiées à la définition des différents aspects des langages. Ensuite, elles répondent aux préoccupations inhérentes au développement de logiciels tels que la modularité ou la testabilité.

Cependant, les bénéfices de telles abstractions peuvent être perdues quand traduites vers des implémentations de services de langages (p. ex. éditeurs, interpréteurs, débogueurs...). Cela entraîne de nombreux inconvénients, en particulier concernant la réutilisabilité et les performances. Les experts métiers sont confrontés à ces limitations et sont forcés de raisonner à travers les détails d'implémentations complexes des implémentations des services de langages.

Ces problèmes peuvent être atténués en exploitant les informations fournies par les abstractions de langages dédiés. Pour ce faire, nous proposons deux patrons d'implémentation qui supportent la réutilisation et les performances. Ainsi qu'une traduction systématique depuis les spécifications de langages vers ces patrons. Le premier patron d'implémentation s'attaque à la réutilisation de langages et s'appelle REVISITOR. Le REVISITOR permet la réutilisation sûre et modulaire de langages, à la fois syntaxiquement et sémantiquement, tout en s'appuyant uniquement sur des concepts courants de programmation orientée objet. Le second patron d'implémentation adresse la performance des langages. Celles-ci sont améliorées par l'introduction d'optimisations spécifiques au langage. Ces optimisations sont introduites automatiquement, sans être intrusif de la manière usuelle de définir les langages.

Nous intégrons de manière transparent notre approche dans l'écosystème de l'EDI Eclipse. Pour ce faire, nous utilisons deux métalangages : Ecore et ALE. Ecore supporte la spécification de syntaxes abstraites sous la forme de méta-modèles. Ecore est fourni dans le cadriciel EMF[2], standard du monde industriel. ALE supporte

[1] Language workbench: Banc de création de langages.

[2] Eclipse Modeling Framework (EMF) : un cadriciel dédié à la modélisation dans Eclipse.

la spécification modulaire de sémantiques de langages par-dessus les méta-modèles Ecore. Nous fournissons deux compilateurs pour ces langages. Ils autorisent respectivement la compilation des spécifications de langages vers le patron REVISITOR ou l'introduction automatique d'optimisation de performance dans les implémentations des interpréteurs. Nous évaluons les bénéfices de ces approches par l'implémentation d'une sélection variée et hétérogène de langages dédiés.

Nos contributions rendent possible l'implémentation de langages dédiés réutilisable et efficaces (bien qu'actuellement séparées, nous travaillons à leur intégration) pour les experts métiers. En pratique, notre approche est à la fois non-intrusive de l'ingénierie des langages et basée sur une génération systématique des implémentations. Par conséquent, notre approche est directement applicable dans un contexte industriel et peut être intégrée avec des artefacts de langages dédiés existants.

Title: On modularity and performance of External Domain-Specific Language implementations

Keywords: reuse, performances, domain-specific languages, software language engineering

Abstract: Model-Driven Engineering (MDE) aims at supporting Domain Experts when developing complex systems, by separating concerns through the use of models. Models are representations of specific aspects of a system and are defined using relevant abstractions. Such abstractions are defined using Domain-Specific Languages (DSLs). DSLs are created with specialized Integrated Development Environment (IDE), called language workbenches. Language workbenches assist the engineering of languages by offering useful language abstractions. First, these abstractions have the benefit of providing the relevant level of abstraction for the specification of languages. Second, they address the concerns inherent to software development, such as modularity or testability.

However, the benefits of these abstractions can be lost when translated to language service implementations (e.g., editors, interpreters, debuggers). This has many drawbacks, especially in terms of reusability and performance. Domain Experts are subject to these limitations and are forced to reason in terms of the low-level intricacies of language services implementation.

These problems can be alleviated by exploiting the information provided by the abstractions available in the DSL specifications. To do so, we propose two new implementation patterns supporting reuse and performances, and a systematic translation of language specifications to these patterns. The first implementation pattern tackles language reuse and is called REVISITOR. The REVISITOR pattern allows the safe and modular reuse of languages, both syntactically and semantically, while relying solely on

mainstream object-oriented programming concepts. The second implementation pattern addresses language performances. Language performances are improved by introducing language-specific optimizations. These optimizations are automatically introduced without being intrusive of the usual language development methods.

We seamlessly implement our approaches on top of the Eclipse IDE ecosystem by using two metalanguages: Ecore and ALE. Ecore supports the specification of abstract syntaxes in the form of metamodels. Ecore is provided by the de facto industrial standard Eclipse Modeling Framework (EMF). ALE supports the modular specification of language semantics on top of Ecore metamodels. We provide two compilers for these metalanguages. They support respectively the compilation of language specifications to the REVISITOR pattern and the automatic introduction of performance-specific optimizations in DSL interpreter implementations. We evaluate the benefits of our approaches by implementing a varied selection of heterogeneous DSLs.

Our contributions make possible the implementation of reusable or efficient DSLs for Domain Experts — while currently separated, future work aims to integrate them. In practice, our approach is both non-intrusive of the usual methodology of language engineering and based on automated code generation. Consequently, our approach is directly applicable to industrial contexts and can be integrated with legacy DSLs artifacts.