Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

Automatic generation of Truffle-based interpreters for Domain-Specific Languages

Manuel Leduc[a] Gwendal Jouneaux[a] Thomas Degueule[b]
Gurvan Le Guernic[c] Olivier Barais[a] Benoit Combemale[ad]

a. Univ. Rennes, Inria, CNRS, IRISA, France
b. Univ. Bordeaux, Bordeaux INP, CNRS, LaBRI, UMR5800, F-33400 Talence, France
c. DGA MI, Bruz, France
d. Univ. Toulouse, IRIT, France

Abstract Numerous language workbenches have been proposed over the past decade to ease the definition of Domain-Specific Languages (DSLs). Language workbenches enable DSL designers to specify DSLs using high-level metalanguages, and to automatically generate their implementation (e.g., parsers, interpreters) and tool support (e.g., editors, debuggers). However, little attention has been given to the performance of the resulting interpreters. In many domains where performance is key (e.g., scientific and high-performance computing), this forces DSL designers to handcraft ad-hoc optimizations in interpreter implementations, or lose compatibility with tool support.

In this paper, we propose to systematically exploit domain-specific information of DSL specifications to derive optimized Truffle-based language interpreters executed over the GraalVM. Those optimizations are provided at no extra cost for the DSL designer. They are of course not as efficient as handcrafted optimizations, but do not require extra time or knowledge from the DSL designer (which industrial DSL designers often lack). We implement our approach on top of the Eclipse Modeling Framework (EMF) by complementing its existing compilation chain with Truffle-specific information, which drives GraalVM to benefit from optimized just-in-time compilation. A key benefit of our approach is that it leverages existing DSL specifications and does not require additional information from DSL designers who remain oblivious of Truffle's low-level intricacies and JIT optimizations in general while staying compatible with tool support.

We evaluate our approach using a representative set of four DSLs and eight conforming programs. Compared to the standard interpreters generated by EMF running on GraalVM, we observe an average speed-up of x1.14, ranging from x1.07 to x1.26. Although the benefits vary slightly from one DSL or program to another, we conclude that our approach yields substantial performance gains while remaining non-intrusive of EMF abstractions.

Keywords domain-specific languages; meta-languages; interpreters; just-in-time optimization

1 Introduction

Numerous language workbenches [EVDSV$^+$15] have been proposed over the past decade to ease the development of Domain-Specific Languages (DSLs). The main objective is to support domain experts by reifying concepts dedicated to a given application domain to ease the development of future complex systems in this particular domain. Language workbenches provide dedicated metalanguages to specify the various concerns of a DSL (e.g., abstract syntax, concrete syntax, and semantics), together with generative or generic approaches that automate the production of language tooling, including editors, compilers, interpreters, and analysis tools. Among others, we can cite industrial language workbenches such as MetaEdit+ [TR03], MPS [Voe11], EMF [SBMP08], or academic projects such as Rascal [BvdBH$^+$15], Spoofax [KV10], Neverlang [VC15] or the GEMOC Studio [BDV$^+$16].

By their very nature, generative and generic 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 do not take into account specific optimizations tailored to the specificities of a particular application domain or execution environment. For instance, from any given metamodel, EMF always derives the same kind of interpreters based on the same Visitor-like pattern, which is sub-optimal from an execution time point of view.

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 [WW12] relies on the Partial Evaluation capabilities provided by GraalVM [Ora19] to realize such optimizations. Truffle offers facilities to complement an initial DSL interpreter implementation with patterns and annotations to benefit from specific run-time optimizations. Performance gains reported in the literature are significant [WWH$^+$17]. 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 this paper, we aim at leveraging high-level abstractions, provided by meta-languages and language workbenches, to abstract away from low-level intricacies of interpreter optimization. In addition, we do so without breaking compatibility with other tools provided by DSLs workbenches (e.g., editors, debuggers). We introduce a systematic approach to automatically generate optimized Truffle-based language interpreters from model-based DSLs specifications, 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.

This non-intrusive approach to the optimization of model-based interpreter performances is evaluated on a representative set of four languages and eight conforming programs (from programming languages to "end-user" languages through modeling 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 paper enables language designers to automatically obtain efficient language interpreters while remaining oblivious of the technical details of the interpreter optimizations.

The remainder of this paper is organized as follows. Section 2 introduces useful preliminary notions. Section 3 presents the language design and implementation patterns studied in the remainder of this paper. Section 4 gives an overview of our approach and presents the Truffle optimizations applicable in our context. Section 5 presents our implementation choices. Section 6 evaluates the resulting tools and metalanguages on several languages and programs. Finally, Section 7 relates our approach to existing work, and Section 8 concludes and discusses future work.

2 Background

In this section, we provide background information on: language specification using the Eclipse Modeling Framework (EMF) and the Action Language for Ecore (ALE); and language execution and implementation using GraalVM and Truffle.

2.1 Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) [SBMP08] is aligned with the Meta-Object Facility (MOF) [OMG06], a standard for the definition of models and metamodels. EMF integrates Ecore, an industrial-grade metamodeling language and de-facto standard, well known in the modeling community. Due to its popularity, many tools are provided to manipulate Ecore metamodels (e.g., tree-based, textual,[1] or graphical[2] editors). This compatibility with a wide range of tools is a key aspect of EMF, and we aim at preserving it while improving language performance. Ecore metamodels are compiled to Java code that implements the Ecore concepts presented below. Our contribution extends the compilation chain of EMF to provide additional performance gains for EMF-based language interpreters without breaking their compatibility with the existing EMF ecosystem.

MiniJava is a teaching-oriented subset of Java [Rob01]. We use MiniJava as an illustration in this section, and later in our evaluation. Figure 1 presents an excerpt of MiniJava's metamodel. Following a popular language abstract syntax pattern, the concept of `Or` inherits from `Expression` which itself inherit from `Statement`. Inheritance relations are visually identifiable by unfilled arrows on the parent side of the relation. The concept of `Or` contains two sub-expressions of type `Expression`,

[1]Xcore: `https://wiki.eclipse.org/Xcore`
[2]EcoreTools: `https://www.eclipse.org/ecoretools/`

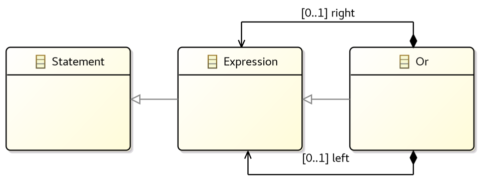

Figure 1 – An excerpt of the MiniJava metamodel defined using Ecore

```
use miniJava.LogService;

open class Statement {
  def dispatch void evaluateStatement(State state) {}
}

open class Expression {
  def void evaluateStatement(State state) {
    self.evaluateExpression(state);
  }
  def Value evaluateExpression(State state) {}
}

open class Or {
  def Value evaluateExpression(State state) {
    BooleanValue left := self.left.evaluateExpression(state);
    BooleanValue right := self.right.evaluateExpression(state);
    BooleanValue res := miniJava::BooleanValue.create();
    res.value := left.isValue() or right.isValue();
    res.value.log("INFO");
    result := res;
  }
}
```

Listing 1 – An excerpt of the operational semantics of MiniJava in ALE.

respectively named `left` and `right`. Sub-expressions are defined using containment relations—conceptually equivalent to UML's composition relations [UML05]—and are visually identifiable by black diamonds at the start of the relation.

Ecore provides the expressiveness needed to express abstract syntax in an object-oriented way. However, Ecore does not provide useful abstractions for the definition of language execution semantics. ALE, presented in the next section, enables0 the modular definition of operational semantics on top of Ecore-based abstract syntaxes.

2.2 Action Language for Ecore

Action Language for Ecore (ALE) is a metalanguage dedicated to the definition of language execution semantics. Inspired by Kermeta [JCB+15], ALE is statically typed and supports type inference of its expressions. ALE is based on the "open-class" principle [CLCM00], allowing the modular definition of semantics on top of existing metamodels (i.e. abstract syntaxes) [LDC+17]. Listing 1 presents an excerpt of the implementation of an execution semantics, in the form of an interpreter on top of the metamodel of MiniJava presented in Figure 1. The `Or` class re-opens the concept of `Or` defined in Figure 1 to insert an `evaluateExpression` method. The result of the method is obtained by evaluating the `left` and `right` sub-expressions (lines 16 and 17 of Listing 1) and applying the logical OR operation on the results (Line 19). Finally, the result of the evaluation of the logical OR is wrapped in an instance of `BooleanValue` and returned (Line 21). The `:=` operator is used for variable assignment. Inspired by the Eiffel language [Mey92], ALE does not have a **return** keyword, but the content of the **result** variable is returned. Class instantiation is done using the special **create()** method (Line 18). Extensions are possible thanks to the mechanism of services inspired by the extension method mechanism [EEK+12]. Lines 1 and 20 are respectively the import of an ALE logging service and the call to the `log` method provided by the service. An ALE service is a Java class with static methods, and aims at providing methods to solve problems outside the scope of the domain of the language (e.g., logging, user interfaces, database access).

EMF and ALE are seamlessly integrated into the Eclipse IDE and support language engineers in the specification of DSLs, following a standard object-oriented approach.

2.3 GraalVM

GraalVM is a JVM resulting from an internal project of Oracle [WWW+13], and has shown promising performance improvement results compared to the standard HotSpot VM. 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 Just-In-Time (JIT) compiler, called Graal, which produces native code from Java bytecode. Graal is also used as an ahead-of-time compiler by the Substrate VM,[3] allowing the compilation of Java bytecode to native machine code outside of a virtual machine.

2.4 Truffle

Truffle [WW12] is a framework designed to ease the development of efficient interpreters on the JVM. Several works demonstrate the speedup offered by Truffle-based *handcrafted* interpreters [SWHJ16, MDM16]. This makes it appealing to explore its use in *automatically optimized* DSL interpreters. Truffle provides a set of classes, annotations, and built-in operations in order to produce efficient Java implementations of interpreters following the Interpreter pattern. Using Truffle requires to additionally decorate classes with annotations that assist in the definition of efficient language implementations.

At run time, Truffle relies on the use of Partial Evaluation [Lom67, Fut99], on the combination of the program to be executed and the language interpreter, in order 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.

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

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 run-time data later contradict those assumptions (e.g., the variable eventually becomes null), Truffle can *deoptimize* the code and switch back to the original interpreter.

Without external constraints, Truffle explores the execution graph eagerly during Partial Evaluation. Consequently, this process might lead to code explosion [WWH+17], 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 behavior. Würthinger et al. [WWH+17] 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 abstraction proposed by Java.

3 DSL Design and Implementation

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, presented in Section 2.1, represents language concepts as a set of metaclasses and their relations. The semantics of a DSL assigns meaning to its constructs. In order to support the operational execution of the conforming models, the semantics is typically defined by an interpreter. It implements its operational semantics in the form of a transition function over execution states. In the modeling world, it is defined using an action language that extends the language concepts with operations. In this paper, we use ALE, presented in Section 2.2, as action language.

The operational semantics defines the evolution of the state of the execution of a program. This evolution implies the modification of the metamodel by the operational semantics at run time. The set of concepts modified during the execution is called the *execution metamodel* [BDV+16].

A DSL's semantics can be implemented either as an interpreter (a piece of software that directly interprets DSL code) or as a compiler (a piece of software that transforms DSL code into another language code that is directly executable or for which interpreters or compilers already exist) following well-defined implementation patterns [OC12, VRCG+99, EG01, WWS+12, ACL+98]. In the case of interpreters, the most common implementation patterns are the Interpreter pattern and the Visitor pattern [GHJV95]. While functionally equivalent, both patterns offer different advantages, notably regarding their modularity, but also regarding their performance. By default, EMF generates interpreter code following a variation of the Visitor pattern named the Switch pattern. In this paper, EMF generated interpreters based on the Switch pattern are therefore used as a reference for performance evaluation of our proposed approach, which itself builds upon interpreters based on the Interpreter pattern.

Below, we describe and study an approach to improve the execution performance of DSLs specified using the model-oriented approach presented above.

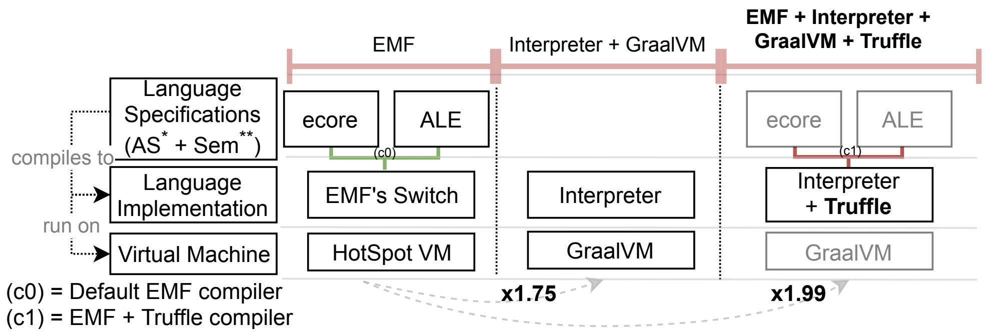

Figure 2 – General overview of the proposed approach. Decimal numbers prefixed with an x represent performance speedups. * Abstract Syntax. ** Semantics.

4 Approach Overview

This section presents a general overview of our approach and presents the optimizations that can be derived from the abstractions provided by EMF-based language specifications defined as presented in the previous section.

Our approach and its context are illustrated in Figure 2. The first column presents the state of practice of language interpreter implementation using the standard EMF Switch pattern for the language interpreter generation and uses the default Java VM (HotSpot VM) for the interpreter's execution. Then, the second column presents a solution that replaces the Switch pattern by the Interpreter pattern while generating interpreter implementations, and uses GraalVM for the execution of the generated interpreter (this is our first set of modifications to the EMF framework compilation chain). 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 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 4.2 discusses how to automatically generate Truffle-compliant object structures from EMF metamodels, and Section 4.3 presents the automatic introduction of Truffle boundaries. Finally, Section 4.4 details the aspects of Truffle that are not included in our approach.

4.1 Preliminary Results

HotSpot VM and the Switch implementation pattern are the default settings of EMF. As we target GraalVM and the Interpreter implementation pattern, we first investigate the impact of the transition to those two options. While this is not strictly a new contribution, it helps us to estimate the expected speedup of such design decisions.

Table 1 summarizes the impact of the transitions from HotSpot VM to GraalVM alone (providing a speedup of x1.56), from the Switch to Interpreter implementation pattern alone (providing a speedup of x1.23), and from both combined which provides

Table 1 – Average speedups resulting from the transition from HotSpot VM to GraalVM and from the Switch implementation pattern to the Interpreter implementation pattern (see Section 6).

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

an overall speedup of x1.75. We can observe that each transition, independently, is very beneficial, and even more when combined. This is likely due to: the improved efficiency of GraalVM's JIT compiler optimizations compared to those of the HotSpot's JIT compiler (at the loss of expressivity, such as introspection); the reduced level of indirections in the Interpreter pattern compared to the Switch pattern (at the price of some modularity). It is also worth noting that using GraalVM is not as efficient at improving execution time of interpreters based on the Interpreter pattern (x1.42 speedup) as it is for interpreters based on the Switch pattern (x1.65 speedup). We refer the reader to Section 6 for an in-depth explanation of our benchmark methodology to better understand how these numbers are computed.

4.2 Truffle-Compliant Object Model Implementation

Starting from an interpreter implementation based on the Interpreter pattern and running on GraalVM, the next challenge to unleash Truffle's power is to translate Ecore metamodels to object models, in the form of a set of Java classes that can be efficiently optimized by Truffle. Truffle expects Java classes to form an immutable tree-shaped set of classes, whereas EMF by default defines graph-based and mutable set of classes. Consequently, the translation from Ecore is not straightforward, and multiple steps are needed to produce efficient Truffle compliant code.

The first step is to discriminate between the immutable metaclasses that define the abstract syntax and should be part of Truffle object models, and mutable metaclasses that define the execution metamodel (see Section 3) that cannot. Immutable metaclasses instantiated in a tree-shaped containment relation can be compiled into Truffle nodes, whereas the others are compiled to standard Java classes. We discriminate those metaclasses using a conservative static analysis of the ALE specifications. Metaclasses instantiated using the **create()** operation (i.e., possibly created at run time) 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, therefore inheriting form the `Node` class (for Truffle compatibility) and from the `EObject` class (for EMF compatibility).

The second step is to identify metaclass references that can be translated into Truffle parent-child relations, which define the tree-shaped hierarchy in the object model. We identified the following constraints i) those references must be containment references; ii) those references must take place between metaclasses that can be translated into Truffle nodes; iii) those references cannot be mutated at run time. References that conform to these three constraints are promoted as parent-child relations.

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

Listing 2 – Integration of the `@Children` annotation on a `Block` statement.

The identification of containment references is realized by a straightforward analysis of the metamodel which explicitly contains this information. 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 annotating the corresponding field with the `@Child` annotation. If 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 array field, named after the original immutable field and suffixed with `Arr`, which "clones" the original `EList` field.

Listing 2 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 12), by copying the element from the list to the array (lines 14 and 15). *CompilerDirectives.transferToInterpreterAnd-Invalidate()* (Line 13) 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.

4.3 Truffle Boundaries

We explained in Section 2.4 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

```java
class LogService {
  @TruffleBoundary
  static void log(Object self, String level) {
    if (level.equals("INFO")) { Logger.info(self); }
    // ...
  }
}
```

Listing 3 – Extract of an ALE service. Methods are annotated with `@TruffleBoundary` in order to anticipate Partial Evaluation issues at run time.

boundaries in the compiler.

Consequently, we can identify exhaustively the places where boundaries are relevant without the need for advanced heuristics or static analysis, thus allowing the safe generation of interpreters without risks of Partial Evaluation failures during program interpretation. For instance, the Java code produced by EMF involves 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 generated 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, calls to classes of the library supporting EMF, or code indirectly called from it (e.g., operations of the reflective API of the classes) are placed behind boundaries.

In practice, 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.

4.4 Discussion of Additional Truffle Optimizations

Our approach to automatic DSL interpreter optimization, presented in this paper, aims at being non-intrusive of language specifications and preserving compatibility with EMF tool support. This section discusses possible performance improvements that we considered, but for which we were unable to come up with solutions respecting those constraints and therefore were not included for the evaluation of Section 6. We first discuss the Polymorphic Inline Cache optimization (Section 4.4.1), before discussing other optimizations (Section 4.4.2).

4.4.1 Polymorphic Inline Cache

Using Truffle allows the definition of Polymorphic Inline Cache (PIC) [HCU91]. PIC is a language optimization historically implemented in the Smalltalk language [DS84] that aims at optimizing dynamically the performance of the call sites frequently dispatched to different methods. Würthinger et al. [WWS+12] present the use of PICs for the optimization of Truffle-based language interpreter implementations.

The introduction of the PIC optimization raises the challenge of automatically identifying which call sites would benefit from such optimization. Indeed, badly placed, PICs can be detrimental to the performance 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 run-time errors if done wrong. We tried to identify the relevant uses of PICs in the 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 6.1 to introduce the dispatch keyword on random method declarations. The results of our experiment are available on the paper's companion webpage.[4] 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. As a result, we decide not to include PIC optimizations in our approach as they clash with our requirements.

4.4.2 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 variables located in the stack rather than in 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 allows for methods 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.

5 Implementation

On the left side of Figure 3, we present the existing Ecore to Java compiler provided as part of the EMF framework, based on the Jet template engine [SBMP08]. 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 code manually introduced previously in the generated code. This is the base mechanism to follow the Interpreter pattern using EMF.

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

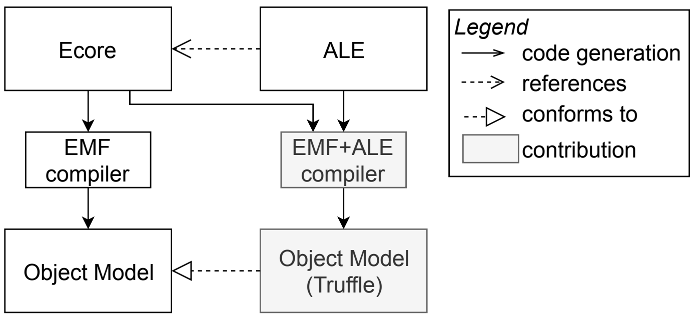

Figure 3 – Presentation of EMF's compiler and our compiler.

On the right side 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 large and hardly maintainable 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 [EB10] and proved itself a relevant solution for the implementation of language interpreters and compilers. Javapoet[5] 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 3, we present the conformance relation between the Java object model generated by EMF's compiler and the Java object models with Truffle concepts generated by our compiler. This conformance relation is twofold. First, the two object models are statically indistinguishable, present the same Java signature, making them fully substitutable. 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. It follows that our compiler is a safe replacement of the EMF 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 class structure compiler, 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 180 lines into the JET template formalism.

[5]Javapoet: `https://github.com/square/javapoet`

6 Evaluation

In this section, we present the evaluation of our approach. In Section 6.1, we present the languages and programs used for the evaluation. Then, in Section 6.2, we present the experimental setup. Finally, Section 6.3 presents and analyzes the results of our experiments.

6.1 Benchmarked Languages and Programs

To evaluate our results, we implemented four languages: MiniJava [Rob01], a teaching-oriented subset of Java, a functional language inspired by OCaml named Boa,[6] a Finite State Machine language,[7] and the educational procedural language Logo.

MiniJava is used to implement: the Fibonacci algorithm (`m_fibonacci`), a bubble sort algorithm [CLRS01] (`m_sort`), a binary tree manipulation algorithm[8] (`binarytree`), and an implementation of the fannkuch algorithm[9] [AR94] (`fannkuchredux`). Boa is used to implement a Fibonacci algorithm (`b_fibonacci`) and an insert sort algorithm [Knu97] (`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[10] (`buffers`). Finally, the Logo language is used to define a program that draws a Koch snowflake fractal[11] (`fractal`).

This selection represents a panel of languages. We made the choice to maximize the representativeness of languages and conforming 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.

6.2 Time Measurement

The methodology presented below aims at producing repeatable performance measurement of language interpreter performance [GBE07]. We qualify the performance by using the steady-state performance, i.e., the performance of a program once it has reached a stable execution state. Each program is executed fifty times in a row in a JVM. We repeat each measurement three times.

All the benchmarks presented below are executed on Debian 9, with 15Go of RAM and an Intel(R) Xeon(R) W-2104 CPU (Quad Core - 3.20GHz). We use HotSpotVM version 1.8.0_222, GraalVM version 19.1.1 and Truffle version 19.1.1.

In addition, we use JMH v1.21[12] to run our experiments. JMH is a Java framework that mitigates the nondeterministic behavior inherent to the JVM internals. Additionally, we execute our benchmarks using Krun [BBK+17]. Krun is a framework

[6]Programming languages Zoo: `http://plzoo.andrej.com/language/boa.html`
[7]Finite State Machine language: `https://github.com/gemoc/MODELS2017Tutorial/`
[8]Binarytree algorithm: `https://benchmarksgame-team.pages.debian.net/benchmarksgame/description/binarytrees.html#binarytrees`
[9]Fannkuch algorithm: `https://benchmarksgame-team.pages.debian.net/benchmarksgame/description/fannkuchredux.html#fannkuchredux`
[10]Companion webpage: `https://manuelleduc.github.io/ecmfa-2020/#system-of-finite-state-machines-example`
[11]Koch snowflake: `https://en.wikipedia.org/wiki/Koch_snowflake`
[12]Java Microbenchmark Harness (JMH): `https://openjdk.java.net/projects/code-tools/jmh/`

Table 2 – 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 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

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 [KBT05].

We benchmark each combination of a virtual machine, an implementation pattern, and a program by running three times 50 executions, obtaining 150 measurements for each combination.

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.

6.3 Results

Table 2 summarizes the measured performance of the programs on our benchmark. 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 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. Each SHS line presents the speedup of the current implementation compared to the HotSpot VM + Switch version. Each SHI line presents the speedup of the current implementation compared to the HotSpot VM + Interpreter version. Finally, the SGI line presents the speedup of the current implementation compared to the GraalVM + Interpreter version.

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 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 version, 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 version. 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.

In conclusion, our results show the benefit of our approach in all cases, especially when improving the performance of language interpreters to their maximum while preserving the compatibility with existing EMF tool support.

7 Related Work

Many language workbenches aim at providing tools and methods for defining the various concerns of languages. However, performance in language runtimes has mostly been tackled in an ad-hoc way, through handcrafted optimizations motivated by the number of language users and the targeted application domains. For instance, in the realm of general-purpose languages, the Java virtual machines have evolved through different generations of optimization until the current JIT compiler, mostly handcrafted [CFM^{+}97]. More recently, the V8 JavaScript engine[13] has been finely tuned by language experts to boost its performance and to make any application executed within the browser extremely efficient [MDM16].

Little attention has been given to the performance of language interpreters for DSLs engineered by language designers through systematic approaches offered by language workbenches. Internal DSLs can exploit the underlying host language runtime, by optimizing in the desugaring phase (e.g., LMS [RO12] or SugarJ [ERKO11]), for acceptable performance. In the context of external DSLs, which cannot rely on the infrastructure of a host language, compilers can be tailored using optimizing compiler techniques [NCM03, EH07, JTH01]. These techniques, however, require advanced language implementation skills, which hampers their use in language workbenches, where language designers are encouraged to manipulate language specifications instead.

Interpreted external DSLs are usually the ones suffering the most from poor performance. The common approach consists in defining an interpreter over the abstract syntax. Using an object-oriented metalanguage, the design pattern Interpreter [GHJV95] offers an elegant and widely adopted architecture to structure its definition through the traversal of the abstract syntax tree and context passing over the traversal. Advanced object-oriented paradigms such as open-classes can be employed to keep the separation of concerns at design time, and inline the interpreter within the structure of the abstract syntax at compile time in order to minimize the runtime overhead. The Switch mechanism offered by the Eclipse Modeling Framework for

[13] V8 Javascript engine: `https://v8.dev/`

traversing Ecore-based metamodels [SBMP08] demonstrates the benefits with regards to the common object-oriented design pattern or specific framework.

Vergu et al. [VTV19] propose their work on the performance of the metainterpreter of DynSem, a metalanguage integrated into the Spoofax language workbench. While our approach consists in automatically deriving Truffle-optimized language-specific interpreters from language specifications, they propose to optimize the metainterpreter of DynSem using Truffle, which can then be reused for any language specification written with DynSem. A key difference is the specifications that are exploited to derive the optimizations: metamodels and ALE specifications in our case, scope graphs and frames in theirs.

A different approach following a similar objective of improving performance consists in applying approximate computing techniques. Several authors have explored such an approach in the context of specific application domains such as signal processing [HS99]. While this approach provides good results in such applications domains, it is bound to possible areas of approximation. Instead, our approach keeps the nominal execution flow described by the execution semantics, and as such, can be applied to any DSL interpreter.

8 Conclusion and Future Work

In this paper, we propose an optimized alternative to the standard execution framework for metamodel-based interpreted DSLs, while preserving the compatibility with existing tool support. Following our approach, interpreters are optimized by introducing Truffle specification concepts on the object model implementation and introducing Truffle boundaries, allowing Truffle to optimize the interpreter at run time while preventing undesirable code explosions. We automatically incorporate Truffle in language interpreter implementations by leveraging the language-specific information provided by metalanguages. This makes some performance optimizations allowed by Truffle accessible to language engineers at zero cost.

We evaluate our approach on four heterogeneous languages and eight programs, covering a broad spectrum of language paradigms. We show a performance speedup of x1.14 on average, ranging from x1.07 to x1.26, while being non-intrusive of the usual development process, and while preserving the compatibility of the language interpreter implementations with their tool support.

Future work Our first exploratory results with the PIC optimizations highlight the interest of the introduction of additional optimizations. It is interesting to explore further the criteria that influence the relevance of Truffle optimization and to pursue the identification of systematically applicable criteria for their application. Besides, in this work, we leverage on the general-purpose optimizations provided by Truffle. A promising step towards more efficient language interpreters is the automatic generation of specialized Graal-based JIT optimizations, derived from the language specifications.

References

[ACL+98] Ali-Reza Adl-Tabatabai, Michal Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh, and James M. Stichnoth. Fast, effective code generation in a just-in-time java compiler. pages 280–290, 1998. doi:10.1145/277650.277740.

[AR94] Kenneth R. Anderson and Duane Rettig. Performing lisp analysis of the fannkuch benchmark. *SIGPLAN Lisp Pointers*, VII(4):2–12, October 1994. `doi:10.1145/382109.382124`.

[BBK+17] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. Virtual machine warmup blows hot and cold. *PACMPL*, 1(OOPSLA):52:1–52:27, 2017. `doi:10.1145/3133876`.

[BDV+16] Erwan Bousse, Thomas Degueule, Didier Vojtisek, Tanja Mayerhofer, Julien DeAntoni, and Benoît Combemale. Execution framework of the GEMOC studio (tool demo). In *SLE 2016*, pages 84–89, 2016. URL: `http://dl.acm.org/citation.cfm?id=2997384`.

[BvdBH+15] Bas Basten, Jeroen van den Bos, Mark Hills, Paul Klint, Arnold Lankamp, Bert Lisser, Atze van der Ploeg, Tijs van der Storm, and Jurgen J. Vinju. Modular language implementation in rascal - experience report. *Sci. Comput. Program.*, 114:7–19, 2015. `doi:10.1016/j.scico.2015.11.003`.

[CFM+97] Timothy Cramer, Richard Friedman, Terrence Miller, David Seberger, Robert Wilson, and Mario Wolczko. Compiling java just in time. *IEEE Micro*, 17(3):36–43, 1997. `doi:10.1109/40.591653`.

[CLCM00] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd D. Millstein. Multijava: modular open classes and symmetric multiple dispatch for java. In *OOPSLA 2000*, pages 130–145, 2000. `doi:10.1145/353171.353181`.

[CLRS01] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. Introduction to algorithms second edition, 2001.

[DS84] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the smalltalk-80 system. In *POPL*, pages 297–302. ACM Press, 1984.

[EB10] 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*, pages 307–309, 2010. `doi:10.1145/1869542.1869625`.

[EEK+12] Sven Efftinge, Moritz Eysholdt, Jan Köhnlein, Sebastian Zarnekow, Robert von Massow, Wilhelm Hasselbring, and Michael Hanus. Xbase: implementing domain-specific languages for java. In *GPCE 2012*, pages 112–121, 2012. `doi:10.1145/2371401.2371419`.

[EG01] M. Anton Ertl and David Gregg. The behavior of efficient virtual machine interpreters on modern architectures. In Rizos Sakellariou, John Gurd, Len Freeman, and John Keane, editors, *Euro-Par 2001 Parallel Processing*, pages 403–413, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[EH07] Torbjörn Ekman and Görel Hedin. The jastadd system - modular extensible compiler construction. *Sci. Comput. Program.*, 69(1-3):14–26, 2007. `doi:10.1016/j.scico.2007.02.003`.

[ERKO11] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. Sugarj: library-based syntactic language extensibility. In *OOPSLA 2011*, pages 391–406, 2011. doi:10.1145/2048066.2048099.

[EVDSV+15] Sebastian Erdweg, Tijs Van Der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, et al. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures*, 44:24–47, 2015.

[Fut99] Yoshihiko Futamura. Partial evaluation of computation process, revisited. *Higher-Order and Symbolic Computation*, 12(4):377–380, 1999.

[GBE07] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. In *OOPSLA 2007*, pages 57–76, 2007. doi:10.1145/1297027.1297033.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, 1 edition, 1995.

[HCU91] Urs Hölzle, Craig Chambers, and David M. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *ECOOP 1991*, pages 21–38, 1991. doi:10.1007/BFb0057013.

[HS99] 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*, pages 30–35, 1999. doi:10.1145/313817.313834.

[JCB+15] Jean-Marc Jézéquel, Benoît Combemale, Olivier Barais, Martin Monperrus, and François Fouquet. Mashup of metalanguages and its implementation in the kermeta language workbench. *Software and System Modeling*, 14(2):905–920, 2015. doi:10.1007/s10270-013-0354-4.

[JTH01] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: rewriting as a practical optimisation technique in ghc. In *Haskell workshop*, volume 1, pages 203–233, 2001.

[KBT05] 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)*, pages 484–490, 2005.

[Knu97] Donald Ervin Knuth. *The art of computer programming: sorting and searching*, volume 3. Pearson Education, 1997.

[KV10] Lennart C. L. Kats and Eelco Visser. The spoofax language workbench: rules for declarative specification of languages and ides. In *OOPSLA 2010*, pages 444–463, 2010. doi:10.1145/1869459.1869497.

[LDC+17] Manuel Leduc, Thomas Degueule, Benoît Combemale, Tijs van der Storm, and Olivier Barais. Revisiting visitors for modular extension of executable dsmls. In *MODELS 2017*, pages 112–122, 2017. doi:10.1109/MODELS.2017.23.

[Lom67] Lionello Lombardi. Incremental computation: The preliminary design of a programming system which allows for incremental data assimilation in open-ended man-computer information systems. *Advances in Computers*, 8:247–333, 1967.

[MDM16] Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. Cross-language compiler benchmarking: are we fast yet? In *DLS 2016*, pages 120–131, 2016. `doi:10.1145/2989225.2989232`.

[Mey92] Bertrand Meyer. *Eiffel the language Prentice Hall object-oriented series*. Prentice hall Upper Saddle River, NJ, USA, 1992.

[NCM03] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for java. In *Compiler Construction, CC 2003*, pages 138–152, 2003. `doi:10.1007/3-540-36579-6_11`.

[OC12] Bruno C. d. S. Oliveira and William R. Cook. Extensibility for the masses - practical extensibility with object algebras. In *ECOOP 2012*, pages 2–27, 2012. `doi:10.1007/978-3-642-31057-7_2`.

[OMG06] OMG. Meta Object Facility (MOF) 2.0 Core Specification. `http://www.omg.org/spec/MOF/2.0/`, 2006.

[Ora19] OracleLabs. Graalvm, 2019. URL: `https://www.graalvm.org`.

[RO12] Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. *Commun. ACM*, 55(6):121–130, 2012. `doi:10.1145/2184319.2184345`.

[Rob01] Eric Roberts. An overview of minijava. In *SIGCSE 2001*, pages 1–5, 2001. `doi:10.1145/364447.364525`.

[SBMP08] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.

[SWHJ16] Lukas Stadler, Adam Welc, Christian Humer, and Mick Jordan. Optimizing R language execution via aggressive speculation. In *DLS 2016*, pages 84–95, 2016. `doi:10.1145/2989225.2989236`.

[TR03] Juha-Pekka Tolvanen and Matti Rossi. Metaedit+: defining and using domain-specific modeling languages and code generators. In *OOPSLA 2003*, pages 92–93, 2003. `doi:10.1145/949344.949365`.

[UML05] Dec 2005. URL: `https://www.omg.org/spec/UML/2.5.1`.

[VC15] Edoardo Vacchi and Walter Cazzola. Neverlang: A framework for feature-oriented language development. *Computer Languages, Systems & Structures*, 43:1–40, 2015. `doi:10.1016/j.cl.2015.02.001`.

[Voe11] Markus Voelter. Language and IDE modularization and composition with MPS. In *GTTSE 2011*, pages 383–430, 2011. `doi:10.1007/978-3-642-35992-7_11`.

[VRCG$^+$99] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–. IBM Press, 1999. URL: `http://dl.acm.org/citation.cfm?id=781995.782008`.

[VTV19] Vlad A. Vergu, Andrew Tolmach, and Eelco Visser. Scopes and frames improve meta-interpreter specialization. In *ECOOP 2019*, pages 4:1–4:30, 2019. `doi:10.4230/LIPIcs.ECOOP.2019.4`.

[WW12] 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*, pages 13–14, 2012. `doi:10.1145/2384716.2384723`.

[WWH$^+$17] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. Practical partial evaluation for high-performance dynamic language runtimes. In *PLDI 2017*, pages 662–676, 2017. `doi:10.1145/3062341.3062381`.

[WWS$^+$12] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-optimizing ast interpreters. In *Proceedings of the 8th Symposium on Dynamic Languages*, DLS '12, pages 73–82, New York, NY, USA, 2012. ACM. URL: `http://doi.acm.org/10.1145/2384577.2384587`, `doi:10.1145/2384577.2384587`.

[WWW$^+$13] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 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*, pages 187–204, 2013. `doi:10.1145/2509578.2509581`.

About the authors

Manuel Leduc is a Ph.D. from the University of Rennes 1. His research interests are model-based language reuse and language performance. Contact him at `manuel.leduc@gmail.com`, or visit `https://mleduc.xyz`.

Gwendal Jouneaux is a master student at the University of Rennes 1. His research interests include Software Language Engineering and adaptive systems. Contact him at `gwendal.jouneaux@etudiant.univ-rennes1.fr`, or visit `https://gwandalff.github.io`.

Thomas Degueule is a research scientist for CNRS working in the Software Engineering group of Laboratoire Bordelais de Recherche en Informatique (LaBRI). His research interests include Software Language Engineering, Model-Driven Engineering, and Empirical Software Engineering. Contact him at `thomas.degueule@labri.fr`, or visit `https://tdegueul.github.io/`.

Gurvan Le Guernic is an Engineer at DGA with a Ph.D. in Computer Science. His research interests include Model-based System Engineering and Languages for Secure System Design. Contact him at `gurvan.le-guernic@def.gouv.fr`.

Olivier Barais Olivier Barais is a Full Professor at the University of Rennes 1, leading the Inria research team DiverSE. He received a Ph.D. degree in computer science from the University of Lille 1, France in 2005. His research interests include Software Architecture, Model-Driven Engineering and Language Engineering. Olivier Barais has co-authored articles in conferences and journals such as SoSyM, IEEE Computer, ICSE, ASE, MoDELS, SPLC and CBSE. Contact him at olivier.barais@irisa.fr, or visit https://olivier.barais.fr/.

Benoit Combemale is a Full Professor of Software Engineering at University of Toulouse and a Research Scientist at Inria. His research interests in Software Engineering include Software Language Engineering, Model-Driven Engineering, and Software Validation & Verification. Contact him at benoit.combemale@inria.fr, or visit https://www.irit.fr/~Benoit.Combemale/.

Acknowledgments This work is partially supported by the CWI/Inria associate team ALE (http://gemoc.org/ale/), the EU's Horizon 2020 Project No. 732223 CROSSMINER, and by the Direction Générale de l'Armement (Pôle de Recherche CYBER).