Revisiting Visitors for Modular Extension of Executable DSMLs

Manuel Leduc
University of Rennes 1
France
manuel.leduc@irisa.fr

Thomas Degueule
CWI
The Netherlands
degueule@cwi.nl

Benoit Combemale
University of Rennes 1
France
benoit.combemale@irisa.fr

Tijs van der Storm
CWI & U of Groningen
The Netherlands
storm@cwi.nl

Olivier Barais
University of Rennes 1
France
olivier.barais@irisa.fr

***Abstract*—Executable Domain-Specific Modeling Languages (xDSMLs) are typically defined by metamodels that specify their abstract syntax, and model interpreters or compilers that define their execution semantics. To face the proliferation of xDSMLs in many domains, it is important to provide language engineering facilities for opportunistic reuse, extension, and customization of existing xDSMLs to ease the definition of new ones. Current approaches to language reuse either require to anticipate reuse, make use of advanced features that are not widely available in programming languages, or are not directly applicable to metamodel-based xDSMLs. In this paper, we propose a new language implementation pattern, named REVISITOR, that enables independent extensibility of the syntax and semantics of metamodel-based xDSMLs with incremental compilation and without anticipation. We seamlessly implement our approach alongside the compilation chain of the Eclipse Modeling Framework, thereby demonstrating that it is directly and broadly applicable in various modeling environments. We show how it can be employed to incrementally extend both the syntax and semantics of the fUML language without requiring anticipation or re-compilation of existing code, and with acceptable performance penalty compared to classical handmade visitors.**

I. INTRODUCTION

The integration of domain-specific concepts and development best practices into modeling languages significantly improves software and systems engineers' productivity and system quality (e.g., [1], [2], [3]). Yet, the development of modeling languages has only been recently recognized as a significant and challenging software engineering task itself, which requires specialized knowledge.

Recent efforts in the modeling community provided various facilities to support the definition of new modeling languages. For example, the Eclipse Modeling Framework [4] (EMF) relies on the object-oriented paradigm to support the definition of both the abstract syntax and semantics of a modeling language. In this context, an Executable Domain-Specific Modeling Language (xDSML) typically consists of a metamodel that defines its abstract syntax, and an interpreter or compiler that defines its execution semantics. The latter can be expressed on top of the abstract syntax using different paradigms (Objects, Aspects, Rules, etc.), all requiring a traversal of the abstract syntax either explicitly (e.g., using the Visitor pattern [5]) or implicitly (e.g., in the underlying execution engine of a declarative specification). The execution semantics is thus eventually implemented through a visitor-like pattern that traverses the abstract syntax [6], and interprets or compiles model elements.

Following the increasing use of xDSMLs in more and more application domains, development and evolution of xDSMLs become recurrent tasks for software and systems engineers. While every xDSML is specifically tailored to a particular purpose (either a specific application domain or a specific system concern), many xDSMLs share recurrent paradigms [7]. For instance, there exist many syntactic and semantic variation points for state machines, which have led to different modeling languages [8]. Similarly, many xDSMLs require an action language to express localized behaviors (e.g., Xbase [9]).

To face the proliferation of xDSMLs and foster reuse between them, a disciplined approach to ease reuse of existing xDSMLs in the development of new ones is of utmost importance. More specifically, this requires supporting extension of xDSML along the syntax and semantics axes in a non-linear way, and the recombination of such extensions (i.e., *independent extensibility*). This should happen without having to anticipate these extensions (i.e., to support *opportunistic reuse*), and without having to recompile existing language modules (i.e., *incremental compilation*). We detail these requirements and provide an overview of our approach in Section II.

Existing approaches fail to comply with the requirements mentioned above. Traditional techniques such as object-oriented interpreters, or the Visitor pattern, offer limited extensibility because of the Expression Problem [10]: interpreters support modular syntax extension, and visitors support adding new operations, but neither support both. Existing solutions to the Expression Problem either use advanced programming features (e.g., path-dependent and value types [11]) unavailable in mainstream languages used in most modeling frameworks (e.g., Java for EMF), or give up on explicit abstract syntax structure (e.g., [12]). Other approaches, such as Melange, overcome these limitations [13], but do not support incremental compilation.

In this paper, we propose a new language implementation pattern, called REVISITOR (Section III), that enables independent extensibility of the syntax and semantics of metamodel-based xDSMLs with incremental compilation and without anticipation. This pattern is inspired by the Object Algebra design pattern [12] as an alternative to the Visitor pattern. The underlying intuition is to combine the behavioral extensibility

of Object Algebras, while keeping the structural extensibility of the object model for the abstract syntax, and at the same time retaining the explicit representation of the abstract syntax for programmatic manipulation. Furthermore, Abstract Syntax Tree (AST) classes do not have to anticipate the application of REVISITOR, as is, for instance, the case for ordinary visitors which require the presence of `accept` methods.

The core idea is to define accept-like methods outside of the AST code that dispatch to visit-like methods using runtime type-checks. Both kinds of methods are defined in a generic REVISITOR interface (like an Object Algebra interface), so that it can be instantiated for different operations (e.g., interpretation, compilation, printing). The dispatch methods *generically* dispatch to the visit-methods which need to be implemented in concrete classes implementing the REVISITOR interface. The dispatch methods only have to be written or generated once, and can be reused for any concrete semantics of a language. In Section IV we show how REVISITORS support (independent) extension of both syntax and semantics of a language.

Although the dispatch methods are themselves type unsafe, instantiations of the REVISITOR pattern can be easily generated from high-level specification languages such as MOF. This also shows that our approach is directly and broadly applicable in various modeling environments. The additional specification level ensures safe reuse and manipulation of xDSMLs (through advanced type group checking [14]), and guarantees that the dispatch methods are correctly implemented. The REVISITOR pattern can be used in any object-oriented programming language that supports (i) parametric polymorphism (generics) with bounded type parameters (ii) multiple class or interface inheritance, and (iii) single dynamic dispatch. For instance, it is straightforward to write REVISITORS using traits in Scala or multiple class inheritance and templates in C++. For the purpose of this paper however, we choose to use Java 8 in all listings to illustrate the pattern.

We have implemented our approach in a prototype language and compiler called ALE (Section V), which seamlessly complements the EMF compilation chain to provide an alternative to the usual Switch class provided for implementing visitors over metamodels. We show how it can be employed to incrementally extend both the syntax and semantics of the fUML language without requiring anticipation or recompilation of existing code, and with acceptable runtime penalty compared to classical handmade visitors or the Switch class mechanism provided by EMF (Section VI).

II. BACKGROUND AND MOTIVATING EXAMPLE

In this section, we recall background notions and highlight the limitations of current approaches to semantics definition and language extension. In the course of the discussion, we present the Expression Problem, what it implies for xDSML engineering, and outline the boundaries of our contributions.

A. Language Engineering and the Expression Problem

The definition of a new xDSML 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 metamodeling formalisms, such as the Meta-Object Facility (MOF), represent language concepts as a set of meta-classes and their relations [15]. Concrete implementations of MOF, such as in the Eclipse Modeling Framework (EMF), generate Java classes corresponding to these meta-classes, and models conforming to a metamodel are graphs of objects that are instance of these classes. The execution semantics of an xDSML assigns meaning to its constructs. It is typically defined by an interpreter that implements its operational semantics in the form of a transition function over model state.

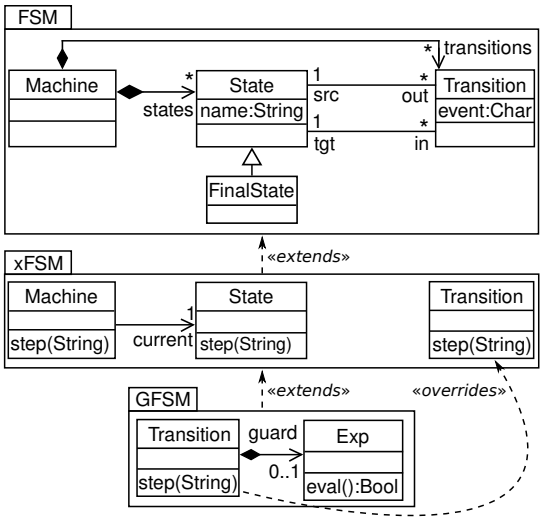

Fig. 1: Extensions of a finite-state machine language

Just as any software, xDSML implementations are bound to evolve to meet new requirements of language users or the specificities of a new domain of application. This situation is highlighted by the idiomatic finite-state machine (FSM) language example depicted in Fig. 1. 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.

When facing such kind of language engineering scenarios, language designers 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 *mutable* AST, whose structure is prescribed by a metamodel, and maintain the *static type safety* usually found in popular modeling workbenches such as EMF.

Given these constraints, we impose additional requirements for modular extensions which are inspired by the classical Expression Problem [10]:

a) 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.

b) Incremental Compilation (R2): 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.

c) 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 designers.

B. Overview: Revisiting Visitors

Fig. 2 shows a high-level visualization of our approach. At the specification level, an xDSML is specified through a classical metamodeling process: an abstract syntax defined by a metamodel (an Ecore model [4] in our case), complemented with an operational semantics which define 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.

The explicit specification of an xDSML 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 for supporting advanced reuse, extension and customization. Second, the xDSML 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 xDSML thanks to a dedicated type groups checker [14].

xDSML specifications are compiled to Java, following the REVISITOR language implementation pattern. 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 Fig. 2). This standard compilation chain is then seamlessly complemented by compiling ALE semantics specifications to REVISITOR artifacts (black arrow in Fig. 2).

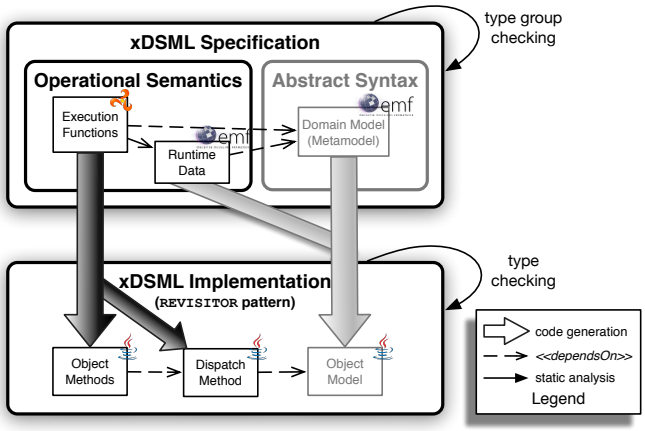

Fig. 2: Approach overview

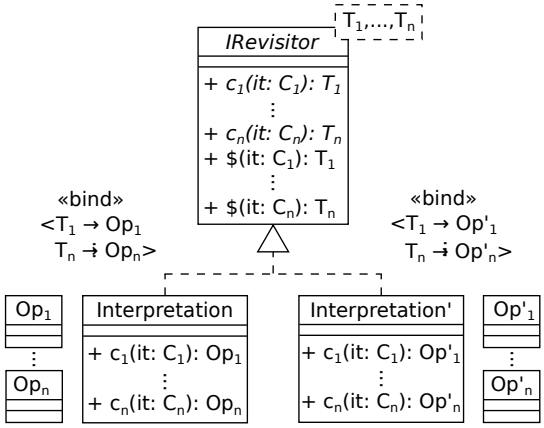

Fig. 3: The REVISITOR Pattern maps syntactic objects of types $C_1,...,C_n$ to semantic objects of types $Op_1,...,Op_n$. Different implementations of the REVISITOR interface of a language lead to different interpretations. Note that the $-methods are implemented in *IRevisitor* and reused for all interpretations.

The compilation of an xDSML specification is incremental: classes representing base syntax or behavior do not have to be recompiled. This property is enabled by the REVISITOR pattern, which we discuss in detail in the next section.

III. THE REVISITOR PATTERN

The REVISITOR pattern is a language implementation pattern that reconciles the modular extensibility offered by Object Algebras [12] with the requirements of having an explicit, metamodel-based abstract syntax to describe graph-structured, mutable models. Fig. 3 depicts the essence of the REVISITOR pattern and is discussed in greater details below. 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 the purpose of this section, metamodels are assumed to be defined by plain Java classes; Section V discusses how the pattern can be applied in the context of EMF.

```
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)
12       return finalState((FinalState) it);
13     return state(it);
14   }
15 }
```

Listing 1: REVISITOR interface for the FSM language depicted in Fig. 1

```
1  interface Pr { String print(); }
2
3  interface PrintFsm extends FsmAlg<Pr, Pr, Pr, Pr> {
4    default Pr machine(Machine it) {
5      return () → it.states.stream()
6        .map(s → $(s).print())
7        .collect(Collectors.joining("\n"));
8    }
9    default Pr state(State it) {
10     ... // omitted for brevity
11   }
12   default Pr finalState(FinalState it) {
13     return () → "*" + state(it).print();
14   }
15   default Pr transition(Transition it) {
16     return () → it.event + "=>" + it.tgt.name;
17   }
18 }
```

Listing 2: A REVISITOR implementation for FSM implementing a pretty-printer

A. REVISITOR *Interfaces*

REVISITOR interfaces (*IRevisitor* in Fig. 3) 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 Fig. 1 is shown in Listing 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 Fig. 3). 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]

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 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 Fig. 3). 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 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 1) is required. The collection of states in a machine abstracts over the difference between ordinary states and final states. As a result, 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

[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 V.

```
1  interface St { void step(String ch); }
2
3  interface ExecFsm extends FsmAlg<St, St, St, St> {
4    default St machine(Machine it) {
5      return ch → { … };
6    }
7    default St state(State it ) { … }
8    // ... etc.
9  }
```

Listing 3: Executing state machines

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

Listing 4: Extending FsmAlg to support timed transitions

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 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.

IV. MODULAR EXTENSIBILITY 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).

A. Semantic Extension

Semantic extension consists of providing a different implementation of the REVISITOR interface of a language. In the example of state machines, for instance, a different semantics could be executing state machines. Listing 3 shows the skeleton code of an execution semantics for 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 needed to be changed or duplicated.

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

Listing 5: Printing timed transitions

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 state machine semantics, the REVISITOR interface `FsmAlg` is extended as `TimedFsmAlg` as shown in Listing 4. The new type parameter `TT` (extending the transition parameter `T`) will represent the semantics of 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 state machines containing timed transitions, reusing the existing `PrintFsm` code. This is shown in Listing 5: 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).

C. Independent Extensiblity

The extensibility scenarios presented up till 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.

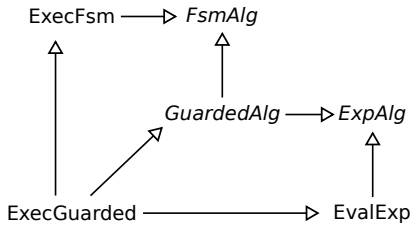

Fig. 4: Independent extension of state machines and expressions, via guarded transitions, reusing existing execution semantics

Listing 6 shows skeleton code illustrating independent extensibility in the context of the state machine example. A visual illustration of the extension relations is shown in Fig. 4. In this case, a new variant of the state machine 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 `GuardAlg` 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 state machines (`ExecFsm`) and the evaluation of expression (`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.

V. MODULAR EXTENSIBILITY WITHIN EMF

One can observe that the REVISITOR pattern does not scale well when implemented manually: the number of concepts in a language grows quickly, resulting in many type parameters (with type bounds). Implementing the $-methods correctly by hand is tedious and error-prone, especially in the presence of complex inheritance hierarchies. The $-methods are type unsafe, and the dispatch conditionals using runtime type-checks (e.g., **`instanceof`** in Java) must be written in the correct order.

However, the REVISITOR pattern can be used as a target for code generation. In this section, we introduce a high-level specification language supporting modular extension of both syntax and semantics where the REVISITOR pattern ensures separate compilation of language modules. We assume the abstract syntax is defined using Ecore metamodels. Methods defining model behavior are specified using a simple *Action Language for Ecore* (ALE) which brings modular extensibility for xDSMLs to the masses.

```
1  // state machine execution
2  interface ExecFsm extends FsmAlg<St,St,St,St> { … }
3
4  // expression language
5  interface ExpAlg<E, …> { … }
6
7  // semantic type for expression evaluation
8  interface Ev { Object eval(); }
9
10 // expression evaluator
11 interface EvalExp extends ExpAlg<Ev, …> { … }
12
13 // "metamodel" extension
14 class Guarded extends Transition {
15   Exp guard;
16 }
17
18 // FSM + expression + guarded transitions as glue
19 interface GuardedAlg<E,…,M,S,F extends S,T,G extends T>
20   extends FsmAlg<M, S, F, T>, ExpAlg<E, …> {
21   G guarded(Guarded it);
22
23   @Override
24   default T $(Transition it) { … }
25   default G $(Guarded it) { … }
26 }
27
28 // guarded FSM execution reusing ExecFsm and EvalExp
29 interface ExecGuarded extends
30   GuardedTrans<Ev, …, St, St, St, St, St>,
31   ExecFsm, EvalExp {
32
33   default St guarded(Guarded it) {
34     return ch → {
35       if ($(it.guard).eval().equals(true))
36         trans(it).step(ch);
37     };
38   }
39 }
```

Listing 6: Independent extensibility: combining state machines and expressions through guarded transitions

A. Extending DSMLs using ALE

To express the semantics of xDSMLs, we propose the *Action Language for Ecore* (ALE), a new meta-language inspired by previous experiments with Kermeta [16]. ALE takes the form of an action language that extends the existing query language AQL (Acceleo Query Language) embedded within the graphical modeling tool Sirius.[2] AQL is a simplified and optimized variant of OCL, without implicit variable references, auto-collect and auto-flatten. It is statically typed and supports type inference of AQL expressions. The main difference between ALE and classical AQL is the support for side effects (to execute models operationally) and the **`open class`** mechanism to retroactively add behavior and state to metamodel classes.

ALE also supports defining Ecore syntax extensions using the **`class`** construct. For the purpose of presentation, however, we assume all syntax is defined as ordinary Ecore metamodels, which are then imported by ALE modules.

Listing 7 shows the state machine printing semantics defined using ALE. The listing shows the ALE equivalent of the Java code shown earlier in Listing 2. The `print` methods are defined as part of opened metamodel classes. In the current implementation prototype of ALE the user still has to explicitly

[2]Cf. http://cedric.brun.io/eclipse/introducing-aql/

```
behavior printfsm;

import ecore "fsm.ecore";

open class Machine {
  def String print() {
    String ret = "";
    for (State s in self.getStates())
      ret = ret + $[s].print();
    return ret;
  }
}

open class State {
  def String print() { … }
}

open class FinalState extends State {
  def String print() { return "*" + $[super].print(); }
}

open class Transition {
  def String print() {
    return self.getEvent() + "=>" + self.getTgt().getName();
  }
}
```

Listing 7: Printing state machines in ALE

```
behavior printtimed;

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

open class TimedTransition extends Transition {
  def String print() {
    return self.getTime() + "@" + $[super].print();
  }
}
```

Listing 8: Printing timed transitions in ALE

"invoke" the $-method, through the use of the `$[...]` construct (e.g., on lines 9 and 19). In future versions, however, we plan to automatically insert these invocations through improved type inference on ALE modules.

The ALE code generator reads an ALE module, 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. The generated REVISITOR factory methods will instantiate separately generated classes implementing the semantic type corresponding to the behavior (e.g., like `Pr` and `St`); these classes contain the actual methods defined in the ALE module.

Defining the printing semantics of state machines, as shown in Listing 7, follows the exact same pattern as in Listing 3. In this case, a different **behavior** heading would indicate that the methods belong to a different kind of semantics. Currently, it is not possible to combine different behaviors (e.g., print and execution) in a modular fashion, since the generated semantic classes cannot be retroactively extended with additional methods.

Syntax extension is specified by importing additional Ecore metamodel(s), as shown in Listing 8. In this case, the generated

```
behavior execguarded;

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

open class Guarded extends Transition {
  def void step(String ch) {
    if ($[self.getGuard()].eval().equals(true)) {
      $[super].step(ch);
    }
  }
}
```

Listing 9: Executing guarded transitions in ALE

```
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);
  AT $(A it) {
    if (it instanceof C) return c_as_a(it);
    return a(it);
  }
  BT $(B it) {
    if (it instanceof C) return c_as_b(it);
    return b(it);
  }
  C $(C it) {
    return c(it);
  }
}
```

Listing 10: Multiple inheritance in REVISITOR interfaces

REVISITOR interface will extend the REVISITOR interfaces corresponding to the imported metamodels. In this example, the interface will extend `FsmAlg` and `TimedAlg`. The generated REVISITOR implementation will extend any preexisting implementation for the same behavior. For instance, in this case, `PrintTimed` will extend `PrintFsm` as well as `TimedAlg`.

The example of independent extensibility, as shown in Listing 6, is reproduced in ALE as shown in Listing 9. The ALE module only contains the definition of the execution semantics of guarded transitions. All other artifacts are reused.

B. Representing Multiple Inheritance

The REVISITOR pattern as presented in Section III 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 inheritance used in the metamodel cannot be directly represented. The workaround employed by the ALE compiler is to not introduce factory methods 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 3 concepts, A, B, and C, where C extends both A and B. The

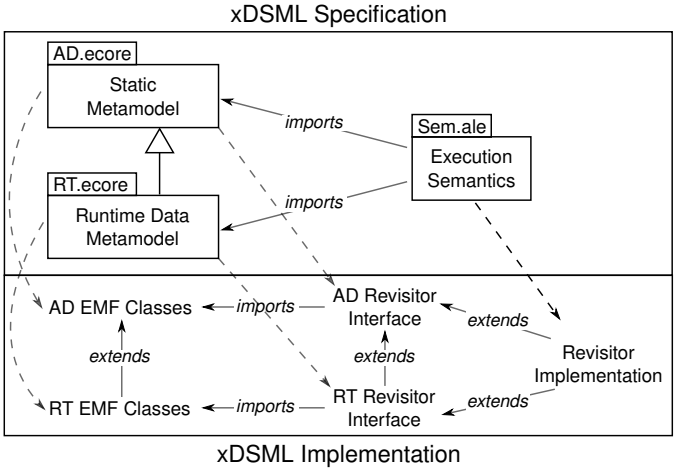

Fig. 5: A modular implementation of fUML using REVISITORS and ALE; dashed lines denote generation flows

generated REVISITOR interface would then be as shown in Listing 10. 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.

VI. EVALUATION

In this section, we illustrate how the REVISITOR pattern and its realization within ALE support modular definition and extension of xDSMLs by modularly defining 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 at the companion webpage.[3]

A. Scenario

The model execution case of the TTC'15 consists of a subset of the fUML language [17], 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 concepts needed for capturing the state of executing models. Then, the computation steps performing transitions of the executing models from one state to the other are defined on the resulting metamodel. The organizers of the contest provide a reference implementation, inspired by the Interpreter pattern, where operations are directly embedded within the Java code generated from the Ecore metamodel in a non-modular way.

B. fUML in ALE

The architecture of the fUML implementation using ALE is shown in Fig. 5. The syntax is defined by the static metamodel (`AD`) and the runtime data metamodel (`RT`) as defined by the TTC'15 use case. Meta-classes contained in the `RT` metamodel either extend existing meta-classes 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 meta-classes that only play a role at run-time (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 Fig. 5).

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 meta-classes, 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 the computation steps 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 following phases. Furthermore, our implementation reuses the metamodels proposed in the TTC'15 model execution case and does not make any assumption on the way they are designed, thus fulfilling the opportunistic reuse requirement.

C. 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 built into EMF. Each implementation executes the three performance-oriented benchmarks proposed by the TTC'15 case. The first one (UC1) executes a model of 1000 activities 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 VI-B. Note that the implementation of the computation steps is identical across all implementation variants: method bodies are copied from the reference implementation. The only difference is the way the

[3] http://www.remodd.org/v1/content/ale-compiler-and-benchmarks

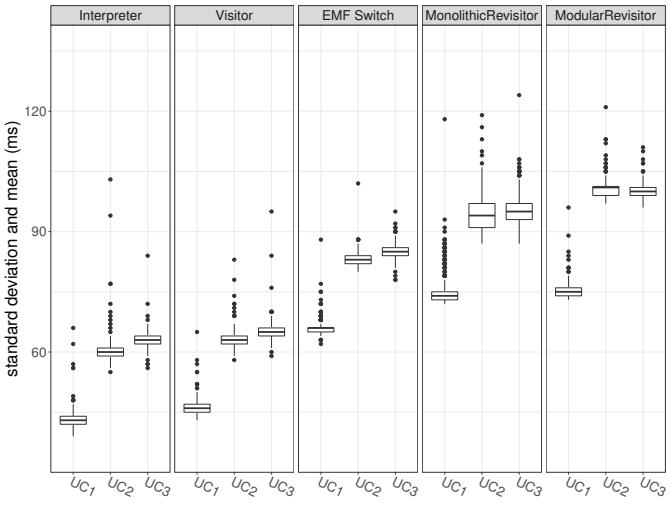

Fig. 6: Summary of the benchmarks

code is structured and how dispatch on a model element is realized.

For each benchmark, the model is executed 500 times after 50 warmup iterations. All benchmarks are run on the same computer: 4 cores i7-5600U @ 2.6Ghz – 16Gb RAM – Java 8 with no tuning options. A fresh new JVM is launched for each use case and implementation.

D. Results

Fig. 6 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).

Both the Interpreter-based implementation and the Visitor-based implementation show similar performance. This confirms the behavior observed earlier between those two implementation patterns [6]. 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 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. 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 extensibility, and that it does not prevent ALE or REVISITORS to be applied in industrial setting.

VII. DISCUSSION & RELATED WORK

The REVISITOR pattern provides modular extensibility of both syntax and semantics for metamodel-based xDSMLs. In this section, we compare our pattern to alternative implementation strategies and discuss limitations. Table I summarizes different approaches to implementing language semantics along the dimensions relevant to our requirements.

A. Extensibility in Programming

Interpreter and Visitor are the traditional design patterns to traverse and interpret AST structures [5]. Both lack extensibility in either semantics or syntax, respectively. Object Algebras support modular, type safe extension in both directions, but eliminate an explicit AST [12]. Kermeta [16] represents a model-based approach to modular behavior weaving, but fails to deliver separate compilation. Finally, the recently proposed *trivial* solution to the expression problem delivers on all accounts, except that ASTs have to be immutable [18].

The REVISITOR pattern proposed here supports all requirements, except for type safety. It is shown as "partially supported", because the developer of the semantics enjoys full type safety. The unsafety becomes apparent when client code passes model objects to a $-method which are of a more specific type than supported by the chain of type checks. As a result, the REVISITOR pattern is less applicable as a programming pattern per se, but can rather be seen as a code generation pattern, where the correctness and completeness of the dispatch is managed through external means. This is precisely why the ALE language supports all requirements, as it guarantees that the dispatch code in generated REVISITORS is correct.

The REVISITOR interfaces and implementations support modular extension in both syntax and semantics. However, syntax extension requires global knowledge of the complete metamodel. If a new language concept changes the existing inheritance hierarchy of the metamodel, previously defined $-methods cannot be reused as is, since the cascade of runtime type-checks will be out of date. The ALE prototype therefore regenerates the complete dispatch code whenever a metamodel is extended. This does not compromise separate compilation of the actual code; it only presupposes that the metamodel itself is globally known. EMF metamodels, however, are not modular in the first place, so we consider this less of an issue.

Implementing a semantics can also be done using object-oriented pattern matching in order to explore a hierarchy of class from the outside. This usually involves classifying objects by their runtime type, accessing their members, or determining

Approach	Syntax Extension	Semantics Extension	Incremental Compilation	Type Safety	Explicit AST	Opportunistic Reuse	AST Mutability
Interpreter [5]	●	○	◐	◐	●	◐	●
Visitor [5]	○	●	◐	◐	●	○	●
Object Algebras [12]	●	●	●	●	○	●	○
Kermeta [16]	●	●	○	●	●	●	●
Trivially [18]	●	●	●	●	●	●	○
REVISITOR	●	●	●	◐	●	●	●
ALE	●	●	●	●	●	●	●

TABLE I: Comparing REVISITOR and ALE to other implementation strategies.
● = supported, ◐ = partially supported, ○ = not supported.

some other characteristics of a group of objects. Emir et al. compare six different pattern matching techniques [19]: object-oriented decomposition, visitors, type-tests/type-casts, typecase, case classes, and extractors. These techniques are compared according to nine criteria related to conciseness, maintainability/extensibility and performance. The authors discuss case classes and extractors as two new pattern-matching methods and show that their combination works well for all of the established criteria. Pattern matching, however, is closed regarding syntax extension.

B. Reuse and Extension in Language Engineering

Modular and extensible language definition has also received a lot of attention in the software language engineering community (e.g., [20], [21], [22], [23]). Below, we review related work that addresses these problems in the context of MDE.

Heim et al. [24] introduce an approach for deriving Visitor infrastructures from context-free grammars to allow language engineers to work with ASTs for model analysis, transformation, and code generation. This approach separates AST traversal from operations that hook into the traversal. In order to define new operations on ASTs, it provides language engineers with generated, statically type-safe Visitor interfaces, which foster reuse during language composition and allow for traversal adaptation. This approach supports reuse of generic traversal behavior and customization of provided generic visitor implementations. However, since the approach is based on traditional Visitor infrastructure, it requires anticipation in the AST classes.

Finally, there is a longstanding research area on composition operators for language specifications (e.g., [25], [26], [23], [13]). Our work complements these works by enabling separate compilation for specific composition operators, such as extension and customization. Whether the REVISITOR pattern can be applied for realizing other operators as well, is an important direction for further research.

VIII. CONCLUSION

In this paper we propose the REVISITOR language implementation pattern, which brings modular extensibility and customization to the definition of executable domain-specific modeling languages (xDSMLs). The pattern can be seen as a variant of Object Algebras [12], 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 definition. The specification level eases the definition of xDSMLs in a uniform object-oriented way, and offers advanced type checking to ensure safe definition and manipulation of xDSMLs.

We show how REVISITORS facilitate modular extension of both syntax and semantics when applied directly in Java. Furthermore, we show how REVISITORS are applied in the context of the new *Action Language for Ecore* (ALE), a high-level language for defining execution semantics in the context of the Eclipse Modeling Framework (EMF). ALE seamlessly extends the compilation chain offered by EMF to generate the execution semantics according to our pattern. This offers an extensible and incremental alternative to the Switch class provided by EMF for implementing Visitors.

We use the ALE prototype to modularly develop the execution semantics of fUML as proposed in the Tool Transformation Context 2015 (TTC'15, [17]). We show that our implementation fulfills our requirements: independent extensibility, incremental compilation, and opportunistic reuse. Furthermore, we use the TTC'15 benchmarks to investigate the performance of ALE and REVISITORS. Results show that the performance overhead introduced by the REVISITOR pattern is acceptable with regard to EMF Switch, standard Visitors, and ordinary interpreter definitions.

This work is a first attempt to apply Object Algebras in the design and implementation of MOF-based modeling languages. It opens a line of research to increase the reusability and customization of such modeling languages. As future work, we will investigate how the extensibility provided by the REVISITOR pattern could be combined with the substitutability facilities provided by model typing [14]. Finally, it would be of great interest to investigate the application of the REVISITOR pattern for translational semantics (i.e., compilers), and to study how it would support modular and composable compilation (resp. transformation) chains.

ACKNOWLEDGMENT

This work is partially supported by the GEMOC initiative, the associate team ALE (http://gemoc.org/ale/), and the EU's Horizon 2020 Project No. 732223 CROSSMINER.

REFERENCES

[1] D. C. Schmidt, "Model-driven engineering," *IEEE Computer*, vol. 39, no. 2, pp. 25–31, 2006.

[2] J. E. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen, "Empirical assessment of MDE in industry," in *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*, 2011, pp. 471–480.

[3] J. Whittle, J. E. Hutchinson, and M. Rouncefield, "The state of practice in model-driven engineering," *IEEE Software*, vol. 31, no. 3, pp. 79–85, 2014.

[4] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[6] M. Hills, P. Klint, T. van der Storm, and J. J. Vinju, "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.

[7] D. Méndez-Acuña, J. A. Galindo, T. Degueule, B. Combemale, and B. Baudry, "Leveraging software product lines engineering in the development of external dsls: A systematic literature review," *Computer Languages, Systems & Structures*, vol. 46, pp. 206–235, 2016.

[8] M. L. Crane and J. Dingel, "UML vs. classical vs. rhapsody statecharts: not all models are created equal," *Software & Systems Modeling*, vol. 6, no. 4, pp. 415–435, 2007.

[9] S. Efftinge, M. Eysholdt, J. Köhnlein, S. Zarnekow, R. von Massow, W. Hasselbring, and M. Hanus, "Xbase: implementing domain-specific languages for java," in *Proceedings of the 11th International Conference on Generative Programming and Component Engineering (GPCE'12)*, 2012, pp. 112–121.

[10] P. Wadler, "The Expression Problem," http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt, Nov. 1998, Java Genericity mailing list.

[11] 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.

[12] B. C. d. S. Oliveira and W. R. Cook, "Extensibility for the masses - practical extensibility with object algebras," in *Proceedings of the 26th European Conference on Object-Oriented Programming (ECOOP'12)*, 2012, pp. 2–27.

[13] T. Degueule, B. Combemale, A. Blouin, O. Barais, and J.-M. Jézéquel, "Melange: A Meta-language for Modular and Reusable Development of DSLs," in *Proceedings of the 8th International Conference on Software Language Engineering (SLE'15)*. ACM, 2015, pp. 25–36.

[14] T. Degueule, B. Combemale, A. Blouin, O. Barais, and J.-M. Jézéquel, "Safe model polymorphism for flexible modeling," *Computer Languages, Systems & Structures*, vol. 49, pp. 176–195, 2017.

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

[16] J. Jézéquel, B. Combemale, O. Barais, M. Monperrus, and F. Fouquet, "Mashup of metalanguages and its implementation in the kermeta language workbench," *Software and Systems Modeling*, vol. 14, no. 2, pp. 905–920, 2015.

[17] T. Mayerhofer and M. Wimmer, "The TTC 2015 model execution case," in *Proceedings of the 8th Transformation Tool Contest (TTC'15)*, ser. CEUR Workshop Proceedings, vol. 1524, 2015, pp. 2–18.

[18] Y. Wang and B. C. d. S. Oliveira, "The expression problem, trivially!" in *Proceedings of the 15th International Conference on Modularity (Modularity'16)*. ACM, 2016, pp. 37–41.

[19] B. Emir, M. Odersky, and J. Williams, "Matching objects with patterns," in *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP'07)*. Springer-Verlag, 2007, pp. 273–298.

[20] M. Gouseti, C. Peters, and T. v. d. Storm, "Extensible language implementation with Object Algebras (short paper)," in *Proceedings of the 13th International Conference on Generative Programming: Concepts and Experiences (GPCE'14)*. ACM, 2014, pp. 25–28.

[21] B. Basten, J. van den Bos, M. Hills, P. Klint, A. Lankamp, B. Lisser, A. van der Ploeg, T. van der Storm, and J. Vinju, "Modular language implementation in Rascal – experience report," *Science of Computer Programming*, vol. 114, pp. 7–19, 2015.

[22] T. Ekman and G. Hedin, "The JastAdd system – modular extensible compiler construction," *Science of Computer Programming*, vol. 69, no. 1–3, pp. 14 – 26, 2007, special issue on Experimental Software and Toolkits.

[23] M. Mernik, "An object-oriented approach to language compositions for software language engineering," *Journal of Systems and Software*, vol. 86, no. 9, pp. 2451–2464, 2013.

[24] R. Heim, P. Mir Seyed Nazari, B. Rumpe, and A. Wortmann, "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.

[25] H. Krahn, B. Rumpe, and S. Völkel, "Monticore: a framework for compositional development of domain specific languages," *International Journal on Software Tools for Technology Transfer*, vol. 12, no. 5, pp. 353–372, 2010.

[26] S. Erdweg, P. G. Giarrusso, and T. Rendel, "Language composition untangled," in *Proceedings of the 12th International Workshop on Language Descriptions, Tools, and Applications (LDTA'12)*, 2012.