Modular Language Composition for the Masses

Manuel Leduc
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
manuel.leduc@irisa.fr

Thomas Degueule
CWI
Amsterdam, Netherlands
thomas.degueule@cwi.nl

Benoit Combemale
University of Toulouse
Toulouse, France
benoit.combemale@irit.fr

Abstract

The goal of modular language development is to enable the definition of new languages as assemblies of preexisting ones. Recent approaches in this area are plentiful but usually suffer from two main problems: either they do not support modular language composition both at the specification and implementation levels, or they require advanced knowledge of specific paradigm that hampers wide adoption in the industry. In this paper, we introduce a *lightweight* approach to *modular* development of *language concerns* with well-defined interfaces that can be composed modularly at the specification and implementation levels. We present an implementation of our approach atop the Eclipse Modeling Framework, namely Alex—an object-oriented meta-language for semantics definition and language composition. We evaluate Alex in the development of a new DSL for IoT systems modeling resulting from the composition of three independently defined languages (UML activity diagrams, Lua, and the CORBA Interface Description Language). We evaluate the effort required to implement and compose these languages using Alex with regards to similar approaches of the literature.

CCS Concepts • **Software and its engineering → Domain specific languages**; *Reusability*;

Keywords language concerns, modular language development, language interfaces

ACM Reference Format:
Manuel Leduc, Thomas Degueule, and Benoit Combemale. 2018. Modular Language Composition for the Masses. In *Proceedings of ACM Conference on Software Language Engineering (SLE'18).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

SLE'18, October 2018, Boston, United States

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

As recently demonstrated in the context of programming languages [4] and modeling languages [24], many software languages, including Domain-Specific Languages (DSLs), have a lot to share, e.g., recurrent constructs and paradigms. As "software languages are software too" [12], such recurrent pieces of software language specification and implementation would benefit from being developed separately to be eventually reused in new contexts.

The promise of modular language development is to liberate language designers from the burden of developing every new language from scratch, and enable them to reuse, assemble, and customize existing *language concerns* to ease the definition of new ones [7]. Recent approaches in the area of modular language development are plentiful. Dedicated paradigms and underlying implementations have been explored to bring specific properties in language specification, e.g., formal composability [4], or off-the-shelf specification in Spoofax [15], Monticore [16] and Neverlang [26]. However, the specific knowledge required to manipulate the corresponding paradigms hampers their wide adoption in industry (i.e., *for the masses*). Other approaches, such as Lisa [21] or Melange [9], provide language reuse capabilities within frameworks and associated ecosystems based on mainstream language engineering technologies (e.g., object-oriented frameworks such as EMF [25]). However, these approaches currently fail to support modularity at the language implementation level which prevents opportunistic reuse of existing languages.

In this paper, we present a *lightweight* approach to *modular* language development that (i) can easily be integrated in mainstream (object-oriented) language engineering technologies, and (ii) is fully modular at the specification and implementation levels of language concerns.

At the specification level, language concerns expose clear interfaces that foster abstraction and information hiding [10]. The interface of a language concern expresses its requirements towards other concerns, and encapsulates the internals of its implementation. Interfaces do not make any assumption on the internals of the syntax and semantics of required concepts, only on the signature of their semantics.

At the implementation level, we provide an object-oriented pattern supporting the composition of language concerns. It supports separate type-checking and compilation, and can be automatically generated from language specifications. This pattern leverages common practices in language engineering

(e.g., [25]) and previous results on the application of object algebras [8] to modular language extensibility [18].

Concern composition most often requires glue, for instance to express how the evaluation contexts of two independent interpreters interact. Leveraging semantic interfaces, this glue can be written *in terms of the interfaces only*. Besides, the glue between two concerns is expressed as a language concern itself, meaning that the knowledge required to compose two concerns is exactly the same as that required to create a concern. Finally, two concerns can be substituted one another provided that they match the same interface.

We provide an implementation of our approach on top of EMF, using Ecore as the meta-language for defining the abstract syntax, and ALE as the meta-language for modularly defining the operational semantics over Ecore meta-models (e.g., in the form of an interpreter) [18]. We extend ALE into Alex to include composition operators and an associated compiler seamlessly integrated within EMF that generate modular concern implementations conforming to the pattern we propose. We use Alex to re-implement a DSL for IoT systems modeling and simulation that was used to evaluate Melange in earlier work [9]. We show that Alex supports a wide range of language composition scenarios, is intuitive to use, and supports modularity at the implementation level.

The remainder of this paper is organized as follows. Section 2 provides some background on modular language development and introduces a motivating example from which we derive a list of requirements. Section 3 gives an overview of our approach to modular language development and Section 4 presents the underlying implementation pattern we propose. In Section 5, we present Alex, a prototype implementation of our approach within the *Eclipse Modeling Framework* ecosystem, and evaluate it on a use case consisting in the definition of a new DSL for IoT systems modeling. We discuss in Section 6 the methodological concerns related to our approach. Finally, Section 7 discusses related work and Section 8 concludes and draws some perspectives.

2 Motivating Example and Requirements

In this section, we first introduce a motivating example and scenario that we use in the remainder of this paper. From this example, we then derive a list of requirements for modular language development that drive our approach.

2.1 Motivating Example

Let us consider the motivating example given in Figure 1 depicting the metamodel of a simple Finite-State Machine (FSM) language. It consists of a machine that contains a number of states and transitions between these states. States may declare local variables of arbitrary types. Transitions are guarded by a guard expression and may execute an action.

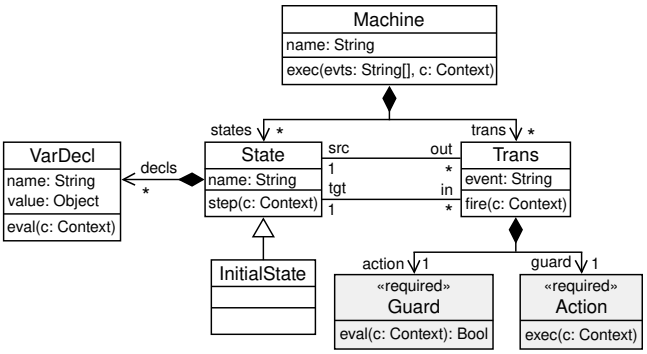

Figure 1. An FSM language concern with explicit interfaces.

For convenience, the evaluation functions defining the operational semantics of domain concepts are depicted as method signatures in the corresponding meta-classes.

From the language designer's point of view, many expression languages would be good candidates for expressing guards, and many action languages would be good candidates for expressing actions.

Rather than defining new guard and action languages from scratch, including their syntax, semantics, and tooling, it would be handy to reuse and plug existing languages that provide these functionalities into the base FSM language—OCL [5] for guards and Xbase [11] for actions, for instance. This would allow modelers to build FSM models by combining the expressiveness of the base FSM language with the expressiveness of dedicated expression and action languages, including their tool support.

The question that naturally arises is: how to express the *required interface* of the FSM language? The notion of language interface in general is the subject of ongoing research [10]. In this paper, we are specifically interested in the interfaces necessary for language composition. From the FSM's standpoint, a guard is merely "a model whose evaluation returns a Boolean;" that is, the signature of its evaluation function is $eval : Ctx \rightarrow Bool$. The internals of the expression language employed, e.g., its syntax (its set of Boolean operators) and semantics (how they are evaluated) do not matter. The key idea here is that most of the language's semantics can be implemented independently from the syntax and semantics of guards and actions. Knowing that actions and guards can be evaluated with their respective evaluation function is sufficient to write the semantics of transitions—thus, only the signatures of their evaluation function is needed. In pseudocode, the semantics of the fire evaluation function of transitions could be written as follows:

```
fire(Trans t, Context ctx) {
  if (t.guard.eval(ctx))
    t.action.eval(ctx);
}
```

An interpreter implementing this semantics can be type-checked and compiled independently; but to be run, it needs concrete implementations of the interpreter of guards and actions—these will be provided later by the external languages themselves.

In Figure 1, the `Guard` and `Action` concepts annotated with `«required»` denote the required interface expected by the base FSM language.

2.2 Language Composition Requirements

From the introduction and motivating example, we derive a list of five requirements that must be addressed for lightweight and modular language development. From now on, following the terminology introduced for concern-oriented language development [7], we refer to languages as languages concerns, regardless of whether they expose an explicit required interface or not.

Concern Encapsulation (R1) Language concerns should be composed without having to inspect their internal implementation. In other words, the information exposed in the interfaces of language concerns should be sufficient to enable safe composition of language concerns.

Explicit Required Interfaces (R2) Required interfaces of language concerns should explicitly state the requirements a concern has towards other concerns. Knowing the interfaces only should be sufficient to state on the validity of the composition of different concerns.

Incremental compilation (R3) Existing language concerns should be type-checked and compiled separately, and should not have to be edited or recompiled to be composed with other language concerns. Furthermore, language concerns should not make any assumption on the way they will be reused and composed, i.e., they should not *anticipate* reuse.

Concern Substitutability (R4) Two language concerns satisfying the same interface should be substitutable one another in the context where this interface is required. From the requiring concern's point of view, the choice of a particular language concern should be transparent. Substitution of a language concern by another should not require any modification of the language concern which depends on it.

Non-intrusivity (R5) The definition of language concerns satisfying the requirements above should not disrupt existing language engineering processes. It should not rely on a new paradigm for the specification of language concerns and should be broadly applicable in mainstream language engineering technologies to foster its adoption.

3 Approach Overview

Figure 2 gives a high-level overview of our approach and is discussed in greater details below. Our approach to modular language development consists in a modular implementation pattern that can safely compose reusable language concerns.

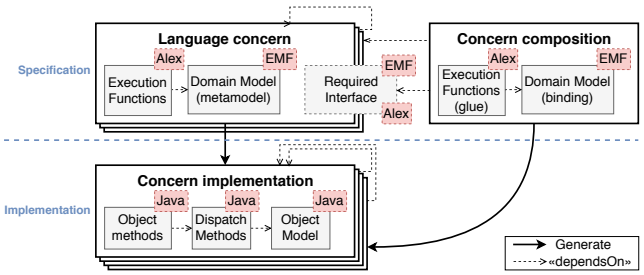

Figure 2. At the specification level, language concerns, their interfaces, and the composition between them are all defined in EMF and Alex. A dedicated compiler generates fully modular Java implementations from them.

At the specification level, a language concern is expressed following a classical metamodeling process. On the one hand, its abstract syntax is specified by an object-oriented metamodel, i.e., a set of meta-classes corresponding to domain concepts including their properties and the relations between them. On the other hand, its operational semantics is defined by a set of execution functions woven on the corresponding concepts of the metamodel [9]. In the case where execution functions would have to manipulate runtime data, these would be specified in a separate dedicated metamodel.

Many formalism could be used to express these two aspects. In our approach, we use Ecore [25] to express metamodels and an extension of ALE [18], named Alex, to express their operational semantics. Following the philosophy of ALE, Alex is a simple action language based on Xbase [11] that enables weaving execution functions in Ecore classes based on static introduction [22].

As mentioned in Section 2, language concerns expose a required interface that materialize their requirements towards other concerns. To express these interfaces, we rely on the built-in annotation mechanism of Ecore to enable language designers to add a Required annotation on concepts of the metamodels that constitute the required interface. The execution functions woven on such concepts consists of signatures only: they express what semantics is expected from the other concerns that will provide such concepts.

The very same meta-languages are used to express how to compose two concerns. To specify that a required concept is realized by an external concept in another concern, we employ a simple delegation pattern between the two concepts: the required meta-class is extended by a new meta-class that holds a reference towards the external concept. A new Ecore metamodel is created to bind all concepts of the required interface of a concern in this way. The glue between the execution function signatures of required concepts and the execution functions of another concern is expressed

in Alex itself. The meta-class that holds the delegate reference implements the required signature in Alex; its body expresses how to glue together the two concerns semantically. Concretely, this means that the skills required to define new concerns are the very same that are required to express how to compose these concerns.

Language concerns are compiled to Java code using two separate compilers: the built-in Java compiler of EMF that compiles Ecore metamodels to a set of Java interfaces and classes, and our own compiler of the Alex meta-language that generates a set of Java interfaces following the pattern described in Section 4. Language concerns can be type-checked and compiled independently of each other. The very same compilation chain is reused to compile the specification of the composition of two language concerns. From a description of the bindings between two concerns in the form of an Ecore metamodel and the glue between their semantics in the form of Alex execution functions, a separate set of Java interfaces that composes the two concerns is generated.

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

4 Modular Language Implementation

From a specification of language concerns as described in Section 3, we derive modular language concern implementations in Java. The modular implementation pattern we propose relies on two main ideas that make it intuitive. First, it leverages two well-known concepts of object-oriented languages: inheritance and the delegation pattern. Second, the same compilation scheme is employed to compile both the language concerns themselves, and the specification of the composition between them.

Our pattern extends the Revisitor pattern that was used in earlier work to support modular and independent extension of the syntax and semantics of DSLs [18]. In this section, we describe how we extend it to account for required interfaces and go beyond strict extension to support composition of language concerns. Our extensions retain the desirable properties of the Revisitor pattern: the syntax and semantics of language concerns can be independently extended modularly and in a type-safe way.

In this section, we present the details of this implementation pattern using Java for illustration. It can however be easily translated in any mainstream 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 trivial to adapt the pattern to Scala using traits or C++ using multiple class inheritance and templates.

4.1 Language concern implementation

In our approach, the abstract syntax of language concerns is defined by an object-oriented metamodel such as the Meta-Object Facility (MOF) [23]. In particular, our implementation uses Ecore metamodels. The compilation chain of EMF automatically generates a set of Java interfaces and corresponding implementation classes for every meta-class in an Ecore metamodel [25].

In order to integrate the Required concept, we extended the Revisitor implementation pattern. Defining a Required class in the metamodel is done by using the EMF annotation mechanism. The class is annotated with the Required value. Required classes are abstract, as they are not instantiable in the current language concern.

The **semantic mapping** intends to specify an abstract mapping from abstract operations to classes of the metamodel. The semantic mapping is represented by a Revisitor interface, which describes how semantics operations can be mapped to the metamodel classes. The Required classes, in addition to explicitly document which concepts are part of the requirement interface, have consequences on the generation of the Revisitor. In a nominal Revisitor interface generation, each class leads to the introduction of (1) a dispatch method, and (2) an abstract factory method. As the Required classes are not meant to be fully implemented in the current language, the generated Revisitor does not include a factory method for the Required classes, but an abstract dispatch method is included. The abstract dispatch methods are expected to be implemented when a concrete class is bound to the Required class (i.e., during languages composition).

Listing 1 depicts an excerpt of the Revisitor interface generated from the Guarded FSM metamodel of Figure 1. The GFSMRev revisitor interface owns a generic type per class in the metamodel. One factory method per class without Required annotation (i.e., machine, trans, etc.) is defined. Consequently, Action does not have a factory method in the revisitor interface. One dispatch method (named $) is defined by class in the metamodel. The Action dispatch method is abstract because Action is Required.

The **semantic interface** is realized by the definition of Java interfaces that represent the operations available on the language. At this point, a set of operations is defined but is not mapped to the classes of the domain model. A *concrete semantic mapping* is realized by inheriting from the Revisitor interface previously defined and binding the operation Java interfaces to the generics.

Listing 2 presents a pretty-printing semantic interface for the Guarded FSM of Figure 1. The IPrint Java interface with a print method is defined as part of the semantic interface. A PrintGFSMRev Java class, inheriting from GFSMRev, is defined. Each of its generics are bound to the IPrint interface. Consequently, the print operation is mapped to all the

```
interface GFSMRev<M,T,A...> {
  M machine(Machine it);
  T trans(Trans it);
  // no Action abstract factory definition
  //  ...

  default M $(Machine it) { return machine(it); }
  default T $(Trans it) { return trans(it); }
  A $(Action it); // abstract dispatch
  // ...
}
```

Listing 1. Revisitor interface for the Guarded FSM language depicted in Figure 1

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

Listing 2. Revisitor implementation of the semantic interface for a pretty printer of the Guarded FSM language depicted in Figure 1

```
interface ImplPrintGFSMRev extends PrintGFSMRev {
  default IPrint machine(Machine it) {
    return () -> "machine " + ...;
  }
  default IPrint trans(Trans it) {
    return () -> it.name;
  }
}
```

Listing 3. Revisitor implementation of the implementation interface for a pretty printer of the Guarded FSM language depicted in Figure 1

concepts of the metamodel. At this point the whole public interface or the language concern is defined, without any concrete implementation of the operations.

The **semantic implementation** is realized by a Java classes that respectively inherits from the Java operations interfaces of the semantic interface. Then, the *implementation mapping* is realized by: inheriting from the interface defined in the semantic mapping, and defining concrete implementations for the factory methods. Each implementation is done by returning instances of the semantic interfaces corresponding to the bindings defined during the semantic mapping.

Listing 3 depicts the implementation layer for the Guarded FSM. A new ImplePrintGFSLRev interface is defined, inheriting from PrintGFSMRev, where anonymous classes declares the concrete semantic implementations returned by the machine and trans factory methods. Each anonymous operation class defines the printing operation from each domain models classes (e.g., a Trans object is printed by returning its name).

4.2 Domain models composition

The first step of the composition of two languages concerns is always the composition of their metamodels. It can be noted that this step can be done regardless of the semantics of the composed language concerns, two language concerns should always have the same syntactic mapping without having to consider the mapped semantics.

The metamodels composition is realized by reusing the syntactic extension mechanism provided by the Revisitor implementation pattern. First a new *Glue* class, inheriting the Required bound class, is introduced. It contains a single field, a reference to the class that realizes the Required class in the composed concern, following the object-oriented delegation pattern. This operation is to be repeated for each Required class of the composed concerns.

Figure 3 shows the composition of the Guarded FSM language concern to an Action Language and an Expression Language concerns. Two metamodel bindings are defined for the Required classes Action and Guard, respectively to the Block and Exp classes. In practice, the binding is implemented by defining a BindAction class that inherits the Required Action class and owns a reference to the Block class of the Action Language concern. The Guard to Exp binding is realized following the same logic. Following the Revisitor implementation pattern, this leads to the generation of a Revisitor interface which inherits from the Revisitor interfaces of the composed language models domain models.

4.3 Semantic interfaces composition

Given a set of language concerns domain models, a semantic mapping and a domain models composition, if some domain model layers are composed of Required concepts, a set of services without semantics implementation can be identified in the form of the semantic interfaces bound to the Required concepts.

For instance, when the Guarded FSM concern is composed, this set contains: *exec* mapped to Action and *eval* mapped to Guard.

The main activity related to semantic interfaces composition is the definition of the missing semantic implementations. At the Revisitor pattern level, the definition of the glue is done by inheriting from the semantic service interface mapped to the Required domain concept and providing an implementation of it with the relevant glue code.

Figure 3's bottom boxes depicts the semantic implementation layer glue code for the Action and Guard concepts.

Listing 4 presents a sample of the corresponding Java implementation following the Revisitor implementation pattern. In this listing, only the Action binding is presented. First the FullFSMRev interface bind the language concerns

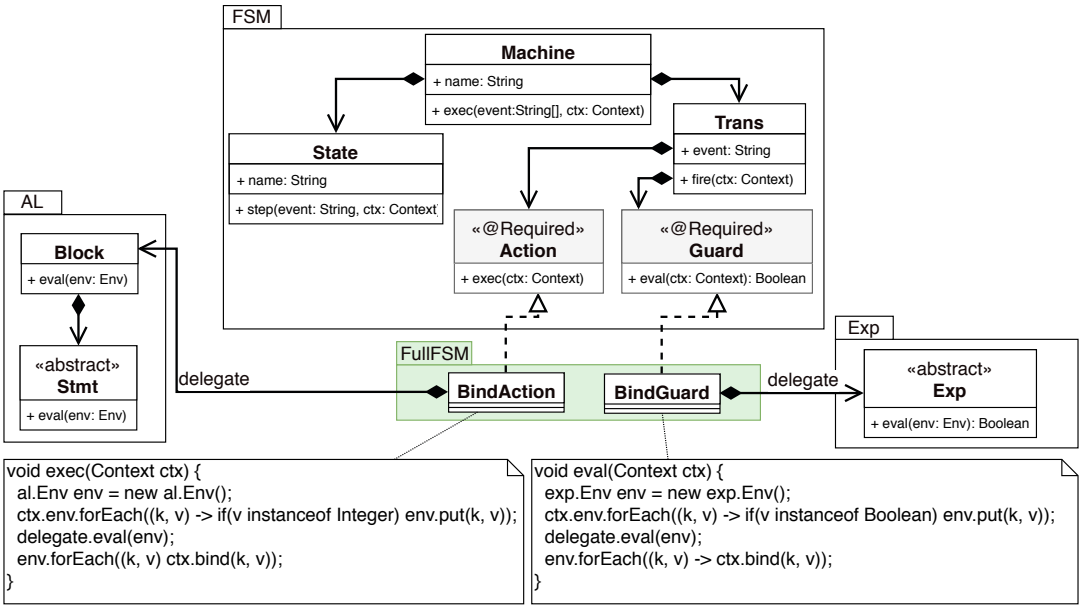

Figure 3. Composing the FSM language concern with an action language concern and an expression language concern

by defining the semantic mapping layer by inheriting from the concerns Revisitor interfaces. It also introduces the mapping to the BindAction glue domain concept. Then, the FullFSMEvalRev defines the implementation mapping and bind the implementation semantics by inheriting from the language concerns respective Revisitor implementations and binding the BindAction implementation which provided the relevant glue code, implemented here as an anonymous class in the body of the bindAction method.

5 Evaluation

This evaluation section is divided in three parts. First, Section 5.1 presents some insights on Alex. Then, Section 5.2 presents the IoT case study used to evaluate our approach. Section 5.3 analyze our implementation in regard to the requirements of Section 2.2. Finally, Section 5.3.2 discusses the impact of our approach on metamodel complexity and runtime performance.

5.1 The Alex Language

Alex[1] (Action Language for Ecore with Xbase) is a meta-language inspired by ALE [18] and Kermeta [14]. It is dedicated to the implementation and composition of operational semantics and tightly integrated with EMF. Following the open class principle [6], it allows to "re-open" meta-classes of a metamodel to weave dynamic behavior, as methods, directly in the meta-classes. Method bodies are written in Xbase [11], a simple yet powerful action language of the Xtext ecosystem that can be easily plugged and reused.

[1] https://github.com/tdegueul/ale-xbase/

```
interface FullFSMRev<...,Action,...,BindActionT extends
      ActionT,...> extends FSMRev<ActionT>, ALRev<...>,
    ExpRev<...> {
  BindActionT bindAction(BindAction it);
  // ...
  default ActionT $(Action it) {
    return bindAction((BindAction) it);
  }
  default BindActionT $(BindAction it) {
    return bindAction(it);
  }
}

interface FullFSMEvalRev extends FullFSMRev<...
    EvalBindAction,...>, FSMEvalRev, ALEvalRev,
    ExpEvalRev {
  default EvalBindAction bindAction(BindAction it) {
    return ctx -> {
      al.Env env = new al.Env();
      ctx.env.forEach((k, v) ->
        if(v instanceof Integer) env.put(k, v));
      delegate.eval(env);
      env.forEach((k, v) ctx.bind(k, v));
    }
  }
}
```

Listing 4. Revisitor semantic implementation layer glue code for the Action to Block binding depicted in Figure 3

Among other benefits, this choice allows us to reuse Xbase tooling, including its built-in Java compiler.

```
1 open class Trans {
2   def void fire(Context ctx) {
3     if (!alg.$(obj.guard).eval(ctx))
4       throw new RuntimeException("Unsatisfied guard");
5     alg.$(obj.action).run(ctx)
6     ctx.current = obj.tgt
7   }
8 }
```

Listing 5. Firing a Transition in Alex

As an illustrating example, Listing 5 depicts the definition, using Alex, of the firing of a transition for the FSM language depicted in Figure 1 corresponding to the pseudocode presented in Section 2. The fire method is defined outside of the transition class itself. Calling other classes semantics is currently done by explicitly calling the $-methods, which takes care of dynamically dispatching to the appropriate method implementation. For instance, on line 5 of Listing 5, the run method is invoked according to the dynamic type of the action attached to the current transition.

The Alex compiler produces a Revisitor implementation according to the semantics definition, this Revisitor implementation inherits from the Revisitor interface derived from the Ecore metamodel imported by the Alex concern. The generated Revisitor factory methods instantiate operation classes that corresponds to the Alex open classes definitions.

5.2 The IoT case study

To illustrate our approach, we re-implemented the case study that was used to evaluate Melange in earlier work [9]. This case study proposes an executable modeling language for the IoT domain. It targets the definition of systems composed of multiple sensors and actuators deployed on resource-constrained micro-controller devices (e.g., Arduino, Raspberry Pi, etc.). This language is built by reusing various existing modeling languages and composing them to form the targeted IoT modeling language. We keep the same list of requirements, reminded below:

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

To fulfill those requirements, Melange's case study selected respectively: OMG's IDL language,[2] a UML Activity Diagram language extracted from the *Transformation Tool Context*,[3] and the scripting language Lua.[4] We reuse the same language concerns for our implementation.

Each language concern is clearly isolated in its own Eclipse plug-in. Dependencies between the concerns are realized by the standard plug-in dependency mechanism offered by Eclipse.

5.3 Case study implementation

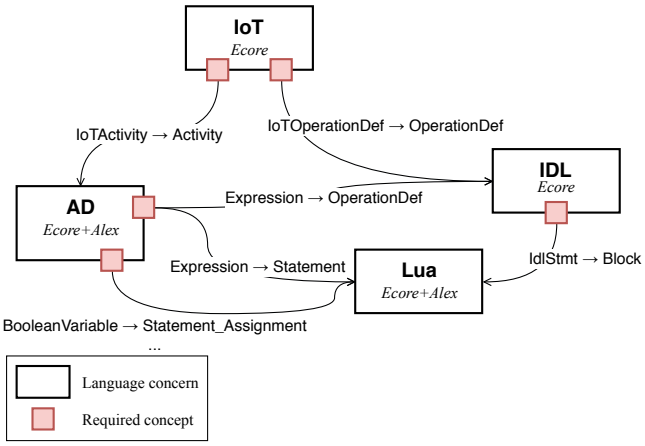

Figure 4. Definition of the case study language by composition of four language concerns

We implement the case study[5] by composing the three concerns detailed in Section 5 with a fourth concern named `IoT` which introduces the vocabulary and concepts of an IoT system.

Figure 4 depicts how those four concerns are composed together. The IoT concern includes two Required concepts in its required interface: IoTActivity and IoTOperationDef. They are respectively bound to the Activity and OperationDef concepts of Activity Diagrams (AD) and IDL. The AD concern exposes three Required concepts: Expression, BooleanVariable and IntegerVariable. The former is bound twice: to OperationDef of IDL, and to Statement of Lua. The two other concepts are bound to the Statement_Assignment concept of Lua. Finally, the IDL concern exposes a Required IdlStmt concept, bound to the Block concept of the Lua concern. This way, every Required concept is bound to a concrete concept and a fully defined language is formed.

From these definitions, we use the built-in EMF compiler and our own custom compiler for Alex to derive a Java implementation of the concerns themselves and their composition following the pattern guidelines presented in Section 4.

Listing 6 presents a sample of the implementation of the IoT language. It can conceptually be represented as semantics

[2] https://www.omg.org/spec/IDL/

[3] http://www.transformation-tool-contest.eu/

[4] https://www.lua.org/

[5] https://github.com/tdegueul/ale-xbase/tree/master/examples/composition/iot

```
1  open class ExpressionBindOperationDef {
2    override void execute(Context c) {
3      // Initialisation of an OperationDef environment
4      val e = new Environment
5      c.inputValues.forEach [iv |
6        e.putVariable(alg.$(iv.variable).name, iv.value)
7      ]
8      (obj.eContainer as OpaqueAction).activity.locals
9        .forEach [l |
10         e.putVariable(
11           alg.$(l).name(),
12           alg.$(l.currentValue).value
13         )
14       ]
15
16     // call to the execution semantics of OperationDef
17     alg.$(obj.delegate.stmt).execute(e)
18
19     // Update of the local context
20     obj.delegate.parameters
21       .filter[p |
22         #[ParameterMode::PARAM_OUT, ParameterMode::
       PARAM_INOUT]
23           .contains(p.direction)
24       ]
25       .forEach [p |
26         c.activity.locals.filter[l |
27           alg.$(l).name() == p.identifier
28         ]
29         .forEach [l |
30           alg.$(l.currentValue)
31             .setValue(e.getVariable(p.identifier))
32         ]
33       ]
34   }
35 }
```

Listing 6. IoT language glue code for the binding of Expression to OperationDef

mapped over the arrows between required concepts and language concerns drawn in Figure 4. In the case depicted in Listing 6, the corresponding arrow is labeled *Expression* → *OperationDef*.

The `ExpressionBindOperationDef` is defined in the metamodel of the IoT language. It inherits AD's `Expression` and has a field named `delegate` which references type `OperationDef` of IDL. The Alex definition of Listing 6 uses the open class mechanism to independently define the semantics of the glue between the Required Expression and the `OperationDef`.

The `execute` method, an abstract method of the semantics of `Expression`, is redefined in the context of the IoT language. Starting line 4, an `Environment` is initialized. It is the parameter of the `execute` method of Lua's `Statement`, defined in the semantics of the IDL language concern. The `Environment` is initialized by copying the current keys and values of the AD's context, which is passed in parameter of `Expression`'s `execute` method.

Then, line 17, the statement of delegated `OperationDef` is used to retrieve its mapped semantics. This operation is done by calling the `$-method` on the instance. The operation is returned, giving us access to the statement semantics, which is called with the initialized Environment in parameter.

Finally, line 20, the variables values possibly updated by Lua's statement evaluation are updated in AD's context. Using ADL's expressiveness we can limit the update to the OUT and IN/OUT parameters.

In the context of this case study, the glue between the language concerns used in the definition of the IoT language stays at this level of complexity and consists mostly in the synchronization of variables values.

5.3.1 Requirement Analysis

In this section we study the requirements of Section 2 and link them to our case study.

Concern Encapsulation (R1) The definition of the glue is fully realized through class inheritance and invocations of the semantic interfaces of language concerns. Hence, in order to compose the four language concerns that are part of our case study, we extend six classes, five methods are overridden for the definition of the glue, and two classes are needed to express how the internal evaluation contexts of different concerns are translated. Language concerns without requirements do not have external dependencies towards other language concerns. Finally, the glue definitions only interact with a small and well-defined part of the reused language concerns. Those observations highlight the isolation capabilities of our approach.

Explicit Required Interface (R2) Each concern requirements are easily identified by looking at the Ecore classes annotated with Required. While most of them are presented in Figure 4, the number of Required classes per language concerns is: none for Lua, one for the IDL, three for the AD, and two for the base IoT concern itself. Each of those Required classes are bound exactly once except for the Exp class of the AD concerns that is bound twice, one time to the IDL concern and another time to the Lua concern. Each bound Required executable semantics declare a single execution function, except for the IoTOperationDef that has no associated semantics. This sums up the information needed for the composition and explicitly exposed in the language concerns interfaces.

Incremental compilation (R3) Each language is clearly isolated in its own Eclipse plug-in containing an Ecore model for its abstract syntax and an Alex file for its semantics. The definition of the IoT language could be done by importing the Eclipse plug-ins of the language concerns from various

places (including remotely, for instance) by using the standard Eclipse update site mechanism, which highlights the modularity of our approach. Each concern produces source code in its own Eclipse plug-in and interacts with other concerns by inheritance and reference to publicly exposed artifact of the other concerns. As long as parts of the languages that are publicly exposed in the artifact are not updated, each language concern can be updated internally without having to recompile other language concerns.

Concern Substitutability (R4) In our use case, as depicted in Figure 4, we bound two different expression language concerns to the AD concern: Lua and IDL can both be used to define expressions of the activity diagrams. From the point of view of the activity diagram concern, the choice of Lua or IDL is transparent. Using the glue between these concerns, Lua and IDL expressions can be used and evaluated indifferently.

Non-intrusivity (R5) The definition of the Required interface is based only on the annotation of classes in the metamodel. This mechanism is built in EMF directly and does not require any intrusive change. The definition of modular languages leverages inheritance and the delegation pattern [13], which are well-known object-oriented concepts. The Revisitor implementation pattern is based on three object-oriented concepts: (i) parametric polymorphism (i.e., generics) with bounded type parameters (ii) multiple class or interface inheritance, and (iii) single dynamic dispatch. Such requirements are easily fulfilled by any mainstream object-oriented language (e.g., Java, C#) and their underlying runtime platforms (e.g., JVM, CLR). In conclusion, every level of our approach is based on existing and well-known object-oriented concepts, leading to a non-intrusive approach, which can be easily ported to compatible technological stacks.

5.3.2 Discussion

In complement to the requirements, we discuss two complementary aspects in this section. What are the consequences of the introduction of intermediate classes when composing languages through the delegation pattern, and what is the impact of modularity on the runtime performance of modular languages.

The application of the delegate pattern, described in Section 4.2, leads to the introduction of new "bind" classes in the metamodel. Those classes are needed for modularly defining and composing languages concerns but are merely technical artifacts that are not related to the domain concepts materialized in metamodels. Still, language designers have to deal with them when adding new tools on top of the language concern.

In order to evaluate the cost of the introduction of those extra classes, we implemented a non-modular Xtext grammar on top the modular language. We observe that managing the extract concepts introduced during the metamodels composition required to introduce additional intermediate production rules in the grammar. These new production rules are only here to deal with the delegation between the production rules of the composed language concerns. Such production rules are simple and do not require advanced grammar engineering knowledge. Compared to a purely monolithic implementation, we estimate the overhead to be of 20 out of 555 lines in the grammar (4%). We claim such cost is largely compensated by the benefits of the modularity and reuse capabilities offered by our approach.

In earlier work, we discussed the impact of the Revisitor pattern at runtime [18]. As explained in Section 4, the definition of modular languages leads, at the implementation level, to multiple inheritance of the composed language concerns Revisitors in the Revisitor of the modular language. We already observed that, due to additional levels of dispatch that cannot be aggressively optimized by the JVM, the Revisitor pattern has a slight impact on performance, albeit reasonable. As in this work, the composition of languages multiplies again the number of dispatch, we expect the performance to be affected in the same way. In addition, the glue between language concerns (for instance, to translate local execution contexts from one concern to the other) may also affect performance negatively. A precise evaluation of this impact remains future work.

6 Discussion

The following paragraph discusses the design of composable language concerns and questions the choice of the boundaries of such concerns. A modular language concern interface should expose the information needed to (i) use and (ii) compose a concern [7]. Using a language is done by first producing a model, hence the structural information, in the form of a metamodel, must be part of the concern interface. Then, semantic operations are called on the model's concepts. To do so the set of provided services and their relationship to the domain concepts must be made available to the concern user. Consequently, the semantic interface layer and the semantic mapping are also part of the concern's interface. In summary a concern interface is composed of the domain models and semantic interface layer and the semantic mapping between them. In contrast, the semantics implementations and the implementation mapping must be encapsulated and it should not be necessary to inspect them in order to use or compose a Modular Language concern.

Now that we have a clear definition of a Modular language concern interface, we have to consider the boundaries of a concern. The boundaries of a language concern are set by the language designer. While we do not enforce strict rules for the definition of language concern boundaries, we suggest following the principles of package cohesion where 1) domain concepts that are used together should be packaged together and 2) a package should not have more than on

reason to change. For instance, when designing the Guarded Finite State Machine concern of Figure 1, the language designer can question the inclusion of that Guard in the language concern. The Guard concept is clearly needed but its realization is subject to discussion. First the implementation of the concepts underlying the notion of Guard are complex and will probably overtake the complexity of the rest of the concern if it would be included. Also, an Expression Language concern is probably already existing and will evolve at a different pace and according to different stakes than the Guarded Finite State Machine. Hence, the Guard domain concept is defined as Required. Consequently, it is outside of the scope of the concern and is expected to be realized later by composition to an Expression Language concern.

7 Related Work

Much work has been done on the definition of reusable languages. Nevertheless, none of them is fully integrated with mainstream (object-oriented) engineering technologies while supporting separate compilation.

Two approaches, Lisa [21] and Melange [9], provide support of language reuse within object-oriented frameworks and technologies. However none of those approaches support separate compilation at the implementation level. Lisa can be distinguished from Melange by the technical space in which it evolves. Lisa is a grammar-first, attribute grammar meta-language while Melange is a model-first meta-language dedicated to the composition of metamodels and object-oriented operational semantics. Being model-first, our approach is influenced by Melange, but supports separate compilation of language concerns.

Other works related to language composition enable separate compilation of languages but introduce new advanced paradigms and do not aim at being integrated in mainstream object-oriented technological stacks. Each of the following work pushed the boundaries of language concern composability in different technical spaces. Monticore [16], Neverlang [2], Rascal [1], and Spoofax [27] are grammar-first language definition frameworks that allow the definition and composition of language; MPS [28] is a language workbench based on projectional editing. MontiCore introduces new advanced concepts in its grammar language (interfaces, aggregation, etc.) to support language composition. Neverlang offers fine-grained granularity for the composition of languages at the price of complex and specific concepts (slices, roles, etc.). Rascal and Spoofax, while not being explicitly dedicated towards composability, have demonstrated their extensibility and composability. Both approaches are based on high-level, domain-specific, operators for the definition of software languages [1, 27]. In constrast, MPS is a projectional language workbench that has positive properties in regard to language composability, mainly due to the absence of a parser. However, despite many recent advances, projectional editing did not percolate in the industry yet. In comparison, our approach does not introduce ad-hoc concepts to allow the safe composition of language concerns and rely only on well known object-oriented concepts (i.e., inheritance, annotations, delegation).

In addition, the question of the modularity of language concerns is strongly connected to the question of Software Product Line (SPL) and feature-oriented language development. Several works study languages from the point of view of software variability. Méndez-Acuña et al., [20] identifies three approaches to language variability management [3, 17, 19]. The Concern-Oriented Language Development (COLD) approach [7] proposes the notion of language concern as the unit of software language reuse. By providing an explicit and modular language interface, our approach can contribute to the improvement of language family and feature-oriented programming solutions.

8 Conclusion and Future Work

In this paper we propose a modular implementation pattern for language concerns composition. Language concerns are equipped with well-defined required interfaces that provide encapsulation and information hiding, and make explicit the requirements a concern has towards other concerns. Language concerns can be composed safely and modularly (i.e., without recompiling any of the concern), and without having to dive into their internals. Our approach is integrated in Alex, an high-level object-oriented language dedicated to the definition of operational semantics on top of Ecore metamodels.

We show that our approach is lightweight and modular through a case study consisting in the definition of a modular language for IoT systems modeling and simulation.

This work is a first step towards a light and efficient definition of software language product lines. It would be of great interest to study how our pattern can help achieve the vision of modular families of software languages.

References

[1] 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. 2015. Modular language implementation in Rascal - experience report. *Sci. Comput. Program.* 114 (2015), 7–19. https://doi.org/10.1016/j.scico.2015.11.003

[2] Walter Cazzola and Edoardo Vacchi. 2013. Neverlang 2 - Componentised Language Development for the JVM. In *Software Composition - 12th International Conference, SC 2013, Budapest, Hungary, June 19, 2013. Proceedings (Lecture Notes in Computer Science)*, Walter Binder, Eric Bodden, and Welf Löwe (Eds.), Vol. 8088. Springer, 17–32. https://doi.org/10.1007/978-3-642-39614-4_2

[3] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. 2009. Variability within Modeling Language Definitions. In *Model Driven Engineering Languages and Systems, 12th International Conference, MODELS 2009, Denver, CO, USA, October 4-9, 2009. Proceedings (Lecture Notes in Computer Science)*, Andy Schürr and Bran Selic (Eds.), Vol. 5795. Springer, 670–684. https://doi.org/10.1007/978-3-642-04425-0_54

[4] Martin Churchill, Peter D Mosses, Neil Sculthorpe, and Paolo Torrini. 2015. Reusable components of semantic specifications. In *Transactions on Aspect-Oriented Software Development XII*. Springer, 132–179.

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

[6] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd D. Millstein. 2000. MultiJava: modular open classes and symmetric multiple dispatch for Java. In *Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA 2000), Minneapolis, Minnesota, USA, October 15-19, 2000.*, Mary Beth Rosson and Doug Lea (Eds.). ACM, 130–145. https://doi.org/10.1145/353171.353181

[7] Benoit Combemale, Jörg Kienzle, Gunter Mussbacher, Olivier Barais, Erwan Bousse, Walter Cazzola, Philippe Collet, Thomas Degueule, Robert Heinrich, Jean-Marc Jézéquel, Manuel Leduc, Tanja Mayerhofer, Sébastien Mosser, Matthias Schöttle, Misha Strittmatter, and Andreas Wortmann. 2018. Concern-Oriented Language Development (COLD): Fostering Reuse in Language Engineering. *Computer Languages, Systems and Structures* (2018), 1–26. https://doi.org/10.1016/j.cl.2018.05.004

[8] Bruno C. d. S. Oliveira and William R. Cook. 2012. Extensibility for the Masses - Practical Extensibility with Object Algebras. In *Proceedings of the 26th European Conference on Object-Oriented Programming (ECOOP'12)*. 2–27.

[9] Thomas Degueule, Benoît Combemale, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. 2015. Melange: a meta-language for modular and reusable development of DSLs. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, SLE 2015, Pittsburgh, PA, USA, October 25-27, 2015*, Richard F. Paige, Davide Di Ruscio, and Markus Völter (Eds.). ACM, 25–36. https://doi.org/10.1145/2814251.2814252

[10] Thomas Degueule, Benoit Combemale, and Jean-Marc Jézéquel. 2017. On language interfaces. In *Present and Ulterior Software Engineering*. Springer, 65–75.

[11] Sven Efftinge, Moritz Eysholdt, Jan Köhnlein, Sebastian Zarnekow, Robert von Massow, Wilhelm Hasselbring, and Michael Hanus. 2012. Xbase: implementing domain-specific languages for Java. In *Generative Programming and Component Engineering, GPCE'12, Dresden, Germany, September 26-28, 2012*, Klaus Ostermann and Walter Binder (Eds.). ACM, 112–121. https://doi.org/10.1145/2371401.2371419

[12] Jean-Marie Favre, Dragan Gasevic, Ralf Lämmel, and Ekaterina Pek. 2010. Empirical language analysis in software linguistics. In *International Conference on Software Language Engineering*. Springer, 316–326.

[13] Erich Gamma. 1995. *Design patterns: elements of reusable object-oriented software*. Pearson Education India.

[14] Jean-Marc Jézéquel, Benoît Combemale, Olivier Barais, Martin Monperrus, and François Fouquet. 2015. Mashup of metalanguages and its implementation in the Kermeta language workbench. *Software and Systems Modeling* 14, 2 (2015), 905–920.

[15] Lennart CL Kats and Eelco Visser. 2010. The spoofax language workbench: rules for declarative specification of languages and IDEs. In *ACM sigplan notices*, Vol. 45. ACM, 444–463.

[16] Holger Krahn, Bernhard Rumpe, and Steven Völkel. 2014. MontiCore: Modular Development of Textual Domain Specific Languages. *CoRR* abs/1409.6633 (2014). arXiv:1409.6633 http://arxiv.org/abs/1409.6633

[17] Thomas Kühn, Walter Cazzola, and Diego Mathias Olivares. 2015. Choosy and picky: configuration of language product lines. In *Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015*, Douglas C. Schmidt (Ed.). ACM, 71–80. https://doi.org/10.1145/2791060.2791092

[18] Manuel Leduc, Thomas Degueule, Benoît Combemale, Tijs van der Storm, and Olivier Barais. 2017. Revisiting Visitors for Modular Extension of Executable DSMLs. In *20th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2017, Austin, TX, USA, September 17-22, 2017*. IEEE Computer Society, 112–122. https://doi.org/10.1109/MODELS.2017.23

[19] Jörg Liebig, Rolf Daniel, and Sven Apel. 2013. Feature-oriented language families: a case study. In *The Seventh International Workshop on Variability Modelling of Software-intensive Systems, VaMoS '13, Pisa , Italy, January 23 - 25, 2013*, Stefania Gnesi, Philippe Collet, and Klaus Schmid (Eds.). ACM, 11:1–11:8. https://doi.org/10.1145/2430502.2430518

[20] David Méndez-Acuña, José A. Galindo, Thomas Degueule, Benoît Combemale, and Benoit Baudry. 2016. Leveraging Software Product Lines Engineering in the development of external DSLs: A systematic literature review. *Computer Languages, Systems & Structures* 46 (2016), 206–235. https://doi.org/10.1016/j.cl.2016.09.004

[21] Marjan Mernik. 2013. An object-oriented approach to language compositions for software language engineering. *Journal of Systems and Software* 86, 9 (2013), 2451–2464. https://doi.org/10.1016/j.jss.2013.04.087

[22] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. 2005. Weaving executability into object-oriented meta-languages. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, 264–278.

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

[24] Ana Pescador, Antonio Garmendia, Esther Guerra, Jesús Sánchez Cuadrado, and Juan de Lara. 2015. Pattern-based development of Domain-Specific Modelling Languages. In *18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MoDELS 2015, Ottawa, ON, Canada, September 30 - October 2, 2015*, Timothy Lethbridge, Jordi Cabot, and Alexander Egyed (Eds.). IEEE Computer Society, 166–175. https://doi.org/10.1109/MODELS.2015.7338247

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

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

[27] Eelco Visser, Guido Wachsmuth, Andrew P. Tolmach, Pierre Neron, Vlad A. Vergu, Augusto Passalaqua, and Gabriël Konat. 2014. A Language Designer's Workbench: A One-Stop-Shop for Implementation and Verification of Language Designs. In *Onward! 2014, Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, part of SPLASH '14, Portland, OR, USA, October 20-24, 2014*, Andrew P. Black, Shriram Krishnamurthi, Bernd Bruegge, and Joseph N. Ruskiewicz (Eds.). ACM, 95–111. https://doi.org/10.1145/2661136.2661149

[28] Markus Voelter. 2011. Language and IDE Modularization and Composition with MPS. In *Generative and Transformational Techniques in Software Engineering IV, International Summer School, GTTSE 2011, Braga, Portugal, July 3-9, 2011. Revised Papers (Lecture Notes in Computer Science)*, Ralf Lämmel, João Saraiva, and Joost Visser (Eds.), Vol. 7680. Springer, 383–430. https://doi.org/10.1007/978-3-642-35992-7_11