This is an author produced version of *The Epsilon Pattern Language*.

White Rose Research Online URL for this paper:
http://eprints.whiterose.ac.uk/123979/

**Proceedings Paper:**

https://doi.org/10.1109/MiSE.2017.8
The Epsilon Pattern Language

Dimitrios S. Kolovos and Richard F. Paige
Department of Computer Science, University of York,
Deramore Lane, York, YO10 5GH, UK.
{dimitris.kolovos, richard.paige}@york.ac.uk

Abstract—We present the Epsilon Pattern Language (EPL), a
textual language that supports expressing and detecting patterns
on models conforming to arbitrary metamodels and captured
using diverse modelling technologies. EPL provides out-of-the-
box integration with existing languages that target a wide range
of related model management activities (such as model validation,
model-to-model and model-to-text transformation), thus enabling
code reuse and seamless runtime interoperability across complex
Model-Driven Engineering workflows. We discuss the syntax and semantics of EPL, its supporting development tools, and
demonstrate how instances of patterns detected using EPL can
be consumed and further processed by other model management
programs.

I. INTRODUCTION

Pattern matching is the activity of discovering sub-structures
of interest within more complex structures. In Model-Driven
Engineering (MDE), pattern matching refers to the process of
identifying sets of model elements that have certain properties
and/or are connected in interesting ways for the model
management task (e.g. model transformation, validation) at
hand. Pattern matching is only one of the steps of a complex
model management process. For example, identified instances
of patterns can be validated, reduced internally to simpler
structures (through in-place transformation), or be used to
guide subsequent model-to-model and model-to-text transfor-
mations. Our review of existing pattern specification languages
for MDE indicates that although such languages often provide
in-place or model-to-model transformation capabilities, they
do not facilitate syntactic and runtime interoperability with
languages targeting model management tasks such as model
validation and model-to-text transformation, and that they are
typically limited to operate on models adhering to a particular
metamodelling architecture, such as the Eclipse Modeling
Framework.

This paper presents the Epsilon Pattern Language (EPL),
a language that supports specifying and detecting structural
patterns in models conforming to diverse metamodels and captured
using a range of modelling technologies. EPL builds
on the Epsilon platform [1] and provides out-of-the-box inte-
gration with existing languages and tools supporting a wide
variety of model management tasks such as model validation,
refactoring, comparison, merging, migration and model-to-
model and model-to-text transformation.

The rest of the paper is structured as follows. In section II
we discuss the limitations of existing MDE pattern matching
languages that have motivated this work. In section III we
discuss the syntax and semantics of EPL and in section IV we
demonstrate how identified patterns can be used in multi-step
MDE workflows. Section V concludes the paper and outlines
directions for future work.

II. BACKGROUND AND MOTIVATION

Several technical solutions have been proposed for the
problem of pattern matching in models. The majority of these
solutions take the form of tailored graphical or textual lan-
guages, through which patterns can be specified at a high level
of abstraction. Accompanying interpreters/compilers can then
match these pattern specifications against concrete models.
Examples of graphical pattern matching languages include
AGG [2] and EMF Tiger [3], while examples of textual
languages include GrGen.NET [4], VIATRA [5] and EMF-
IncQuery [6]. In [7], QVTr has also been used to express and
detect patterns in EMF models.

Pattern matching is often only one of the steps in a se-
quence of model management activities involved in an MDE
workflow. As such, languages for pattern matching should
ideally integrate seamlessly with languages that support other
model management tasks such as model validation, compar-
ison, transformation etc. In our review of previous work, we
have identified that this is not the case; existing languages
for pattern matching typically provide only in-place and/or
model-to-model transformation capabilities, and in order to be
integrated with languages that support other MDE tasks such
as model validation and model-to-text transformation, bespoke
tool adapters need to be developed.

Another limitation of existing pattern matching languages
is that they typically target a specific modelling technology
(e.g. EMF) and/or model representation format. This renders
switching between different technologies or specifying and
detecting patterns that involve elements of heterogeneous mod-
els (e.g. an EMF model and an XML document) particularly
challenging.

The above limitations have motivated us to design and
implement a new pattern matching language, the Epsilon
Pattern Language, which (1) enables seamless runtime inter-
operability and code reuse with languages supporting a
range of MDE model management tasks, and (2) provides
support for specifying patterns that involve elements of models
conforming to different modelling technologies. The following
section provides an overview of the platform on which the
proposed language has been built.
Fig. 1. Overview of the architecture of Epsilon

A. Epsilon

Epsilon [1] is a mature open-source family of interoperable languages for model management that can be used to manage models of diverse metamodels and technologies. At the core of Epsilon is the Epsilon Object Language (EOL) [8], an OCL-based imperative language that provides support for model modification, multiple model access, flow control (loops, branches etc.), user interaction, profiling, and support for transactions. Although EOL can be used as a general-purpose model management language, its primary aim is to be embedded as an expression language in hybrid task-specific languages. Indeed, a number of task-specific languages have been implemented atop EOL, including languages for model transformation (ETL), model comparison (ECL), model merging (EML), model validation (EVL), model refactoring (EWL), model-to-text transformation (EGL) – and now pattern matching (EPL) as illustrated in Figure 1.

B. The Epsilon Model Connectivity Layer

Epsilon takes a broad view on what a model is in order to accommodate a wide range of modelling – and more generally, structured data representation – technologies. To treat models of different technologies in a uniform manner and to shield the languages of the platform (and the developers of model management programs) from the intricacies of underlying technologies, Epsilon provides the Epsilon Model Connectivity (EMC) layer (illustrated at the lower part of Figure 1).

The core abstraction provided by EMC is the IMmodel interface presented in Figure 2, which is a technology-agnostic interface that encapsulates the minimal requirements that a modelling technology needs to support in order to be supported in Epsilon. There are currently several concrete implementations of IMmodel for interacting with EMF and MDR models, XML documents [9], relational databases, spreadsheets and commercial modelling tools such as MetaEdit+ and PTC’s Integrity Modeller. This section briefly discusses how the Epsilon interpreters interact with models through this interface, as this is essential for explaining later on how the results of pattern matching can be consumed by other Epsilon model management programs.

Each Epsilon program (model-to-model/text transformation, set of validation constraints etc.) is executed against a collection of IMmodels through which it can query/modify their underlying concrete models (EMF resources, MDR repositories, XML documents etc.). For example, the EOL program of Figure 3 is executed against an in-memory model repository containing two IMmodels, DB and CD which conform to different metamodels and modelling technologies (EMF and MDR respectively).

In order to evaluate the DB!Table.all expression, the EOL interpreter searches the model repository for a model named DB and when it finds it, it invokes its hasType(type: String) method to check if Table is a valid type for that model. If hasType() returns true then the interpreter invokes the getAllOfKind(type: String) method of the model in order to retrieve all the instances of Table in this model. In the next line, in order to retrieve the value of the name property of t, it iterates through all the models in the model repository to find the one that owns t by calling the models’ owns(element: Object) method. The owning model must then provide a property getter for the element through its getIPropertyGetter() method. The returned IPropertyGetter is then responsible for returning the value of the name property.
III. LANGUAGE SYNTAX AND SEMANTICS

Having introduced the main components of the Epsilon platform that underpins the EPL in section II, this section presents the abstract and concrete syntax of the language as well as its execution semantics. The discussion of the syntax and the semantics of the language revolves around an exemplar pattern which is developed incrementally throughout the section.

The aim of the pattern (which we will call PublicField) is to identify quartets of <ClassDeclaration, FieldDeclaration, MethodDeclaration, MethodDeclaration>, each representing a field of a Java class for which appropriately named accessor/getter (getX/isX) and mutator/setter (setX) methods are defined by the class.

The exemplar pattern is matched against models extracted from Java source code using tooling provided by the MoDisco1 project. MoDisco is an Eclipse project that provides a fine-grained Ecore-based metamodel of the Java language as well as tooling for extracting models that conform to this Java metamodel from Java source code. A simplified view of the relevant part of the MoDisco Java metamodel used in this running example is presented in Figure 4.

![Fig. 3. Example of EMC Runtime Binding](image)

![Fig. 4. Simplified view of the MoDisco Java metamodel](image)

A. Syntax

The syntax of EPL is an extension of the syntax of the EOL language [8], which – as discussed earlier – is the core language of Epsilon. As such, any references to expression and statement block in this section, refer to EOL expressions and blocks of EOL statements respectively.

As illustrated in Figure 5, EPL patterns are organised in modules. Each module contains a number of named patterns and optionally, pre and post statement blocks that are executed before and after the pattern matching process, and helper EOL operations. EPL modules can import other EPL and EOL modules to facilitate reuse and modularity.

![Fig. 5. Abstract Syntax of EPL](image)

In its simplest form a pattern consists of a number of named and typed roles and a match condition. For example, in lines 3-5, the PublicField pattern of Listing 1, defines four roles (class, field, setter and getter). The match condition of the pattern specifies that for a quartet to be a valid match, the field, setter and getter must all belong to the class (lines 8-10), and that the setter and getter methods must be appropriately named2.

```
1 pattern PublicField
2   class : ClassDeclaration,
3   field : FieldDeclaration,
4   setter : MethodDeclaration,
5   getter : MethodDeclaration {
6     match :
7       class.bodyDeclarations.includes(field) and
8       class.bodyDeclarations.includes(setter) and
9       class.bodyDeclarations.includes(getter) and
10      setter.name = "set" + field.getName() and
11      getter.name = "get" + field.getName() or
```

1http://www.eclipse.org/MoDisco/

2To maintain the running example simple and concise, the pattern does not check aspects such as matching/compatible parameter/return types in the field, setter and getter but the reader should easily be able to envision how this would be supported through additional clauses in the match condition.

Fig. 4. Simplified view of the MoDisco Java metamodel
getter.name = "is" + field.getName()
]
15
16@cached
17operation FieldDeclaration getName() {
18    return self.fragments.at(0).name.
19    firstToUpperCase();
20    }

Listing 1. First version of the PublicField pattern

The implementation of the PublicField pattern provided in Listing 1 is functional but not particularly efficient as the match condition needs to be evaluated #ClassDefinition * #FieldDeclaration * #MethodDeclaration times. To enable pattern developers to reduce the search space, each role in an EPL pattern can specify a domain which is an EOL expression that returns a collection of model elements from which the role will draw values.

There are two types of domains in EPL: static domains which are computed once for all applications of the pattern, and which are not dependent on the bindings of other roles of the pattern (denoted using the in keyword in terms of the concrete syntax), and dynamic domains which are recomputed every time the candidate values of the role are iterated, and which are dependent on the bindings of other roles (denoted using the from keyword). Beyond a domain, each role can also specify a guard expression that further prunes unnecessary evaluations of the match condition. Using dynamic domains and guards, the PublicField pattern can be expressed in a more efficient way, as illustrated in Listing 2. To further illustrate the difference between dynamic and static domains, changing from to in in line 4 would trigger a runtime exception as the domain would become static and therefore not able to access bindings of other roles (i.e. class).

1 pattern PublicField
2 class : ClassDeclaration,
3    field : FieldDeclaration
4    from : class.bodyDeclarations,
5    setter : MethodDeclaration
6    from : class.bodyDeclarations
7    guard : (setter.name = "set" + field.getName())
8    getter : MethodDeclaration
9    from : class.bodyDeclarations
10    guard : (getter.name = "get" + field.getName())
11    or getter.name = "is" + field.getName() { }

Listing 2. Second version of the PublicField pattern using domains and guards

The implementation of Listing 2 is significantly more efficient than the previous implementation but can still be improved by further reducing the number of name comparisons of candidate setter and getter methods. To achieve this we can employ memoisation: we create a map (dictionary) of method names and methods once before pattern matching (line 2), and use it to identify candidate setters and getters (lines 9 and 12-14).

1 pre {
2    var methodMap = MethodDeclaration.all.mapBy(m imm.
3        name);
4    }

Listing 4. Demonstration of Active Roles

3) Role Cardinality: The cardinality of a role (lower and upper bound) can be defined in square brackets following the type of the role. Roles that have a cardinality with an upper bound > 1 are bound to the subset of elements from the domain of the role which also satisfy the guard, if the size of that subset is within the bounds of the role’s cardinality. Listing 5 demonstrates the ClassAndPrivateFields pattern that detects instances of classes and all their private fields. If the cardinality of the field role in line 3 was [1..3] instead of ["*"], the pattern would only detect classes that own 1 to 3 private fields.

1 pattern ClassAndPrivateFields

The sections below discuss the remainder of the syntax of EPL.

1) Negative Roles: Pattern roles can be negated using the no keyword. For instance, by adding the no keyword before the setter role in line 8 of Listing 3, the pattern will match fields that have getters but no setters (i.e. read-only fields).

2) Optional and Active Roles: Pattern roles can be designated as optional using the optional keyword. For example, adding optional: true to the setter role would also match all fields that only have a getter. By adding optional: true to the setter role and optional: getter.isDefined() to the getter role, the pattern would match fields that have at least a setter or a getter. Roles can be completely deactivated depending on the bindings of other roles through the active construct. For example, if the pattern developer prefers to specify separate roles for getX and isX getters, with a preference over getX getters, the pattern can be formulated as illustrated in Listing 4 so that if a getX getter is found, no attempt is even made to match an isX getter.
B. Execution Semantics

When an EPL module is executed, all of its pre statement blocks are first executed in order to define and initialise any global variables needed (e.g. the methodMap variable in Listing 3) or to print diagnostic messages to the user. Subsequently, patterns are executed in the order in which they appear. For each pattern, all combinations that conform to the type and constraints of the roles of the pattern are iterated, and the validity of each combination is evaluated in the match statement block of the pattern. In the absence of a match block, every combination that satisfies the constraints of the roles of the pattern is accepted as a valid instance of the pattern.

Immediately after every successful match, the optional onmatch statement block of the pattern is invoked (see lines 8-12 of Listing 5) and after every unsuccessful matching attempt, for combinations which however satisfy the constraints specified by the roles of the pattern, the optional nomatch statement block of the pattern (line 18) is executed. When matching of all patterns is complete, the do part (line 14) of each successful match is executed. In the do part, developers can modify the involved models (e.g. to perform in-place transformation), without the risk of concurrent collection modification errors (which can occur if elements are created/deleted during pattern matching). After pattern matching has been completed, the post statement blocks of the module are executed in order to perform any necessary finalisation actions.

An EPL module can be executed in a one-off or iterative mode. In the one-off mode, patterns are only evaluated once, while in the iterative mode, the process is repeated until no more matches have been found or until the maximum number of iterations (specified by the developer) has been reached. The iterative mode is particularly suitable for patterns that perform reduction of the models they are evaluated against.

IV. IMPLEMENTATION

Having discussed the syntax and semantics of EPL, in this section we briefly discuss the development tools of the language and demonstrate how pattern matching can be seamlessly combined with other MDE tasks, such as model validation and transformation.

A. Development Tools

EPL is supported by Eclipse-based development tools including a syntax-aware editor, a debugger built atop the Eclipse Platform debug framework, and tool-support for fine-grained profiling of the execution of EPL patterns. Figure 6 provides a screenshot of a subset of the EPL development tools which are available as part of the Epsilon distributions (eclipse.org/epsilon/download).

B. Pattern Matching Output

The output of the execution of an EPL module is a collection of matches encapsulated in a PatternMatchModel, as illustrated in Figure 7. PatternMatchModel implements the IModel interface discussed earlier, and as such its instances can be accessed from other programs expressed in languages of the Epsilon family.
• **PublicFieldClass**, instances of which are all the classes in the input model which have been matched to the *class* role in instances of the **PublicField** pattern, and similarly
• **PublicFieldField**,  
• **PublicFieldSetter**,  
• **PublicFieldGetter**

C. Interoperability with Other Model Management Languages

As a **PatternMatchModel** is an instance of **IModel**, after its computation it can be seamlessly queried by other Epsilon programs. For example, Listing 6 demonstrates using the ANT-based Epsilon workflow [10] mechanism to run the EPL module of Listing 3, pass its output to the EVL model validation constraints module of Listing 7 and, if validation is successful, to an ETL model-to-model transformation where it is used to guide the generation of a UML model.

In lines 4-7 of Listing 6, the reverse-engineered Java model is loaded under the local name `Java`. Then, in line 10, the `Java` model is passed on to `publicfield.epl` for pattern matching. The result of pattern matching, which is an instance of the **PatternMatchModel** class (and therefore also an instance of **IModel**) is exported so that it can be used in subsequent tasks under the name `Patterns`. Then, in lines 14, both the `Patterns` and the `Java` models are passed on to the EVL model validation task which validates the identified pattern matches.

Listing 6. ANT workflow calculating and passing a pattern match model to an EVL validation and an ETL transformation module

```xml
<project default="main">
  <target name="main">
    <epsilon.emf.loadModel name="Java"/>
    <epsilon.epl src="publicfield.epl" exportAs="Patterns">
      <model ref="Java"/>
    </epsilon.epl>
    <epsilon.evl src="constraints.evl">
      <model ref="Patterns"/>
      <model ref="Java"/>
    </epsilon.evl>
    <epsilon.etl src="java2uml.etl">
      <model ref="Patterns"/>
      <model ref="Java"/>
    </epsilon.etl>
  </target>
</project>
```

Line 1 of Listing 7 defines a set of constraints that will be applied to instances of the **PublicField** type from the **Patterns** model. As discussed above, these are all matched instances of the **PublicField** pattern. Line 5, specifies the condition that needs to be satisfied by instances of the pattern. Notice the `self.getter` and `self.field` expressions which return the `MethodDeclaration` and `FieldDeclaration` bound to the instance of the pattern. Then, line 6 defines the message that should be produced for instances of **PublicField** that do not meet the defined conditions.
satisfy this constraint.

```java
context Patterns!PublicField {
  guard: self.field.type.isDefined()
  constraint GetterAndFieldSameType {
    check: self.getter.returnType.type = self.field.type.type
    message: "The getter of " + self.class.name + "." + self.field.fragments.at(0).name + 
    " does not have the same type as" +
    " the field itself"
  }
}
```

Listing 7. Fragment of the constraints.evl EVL constraints module

If validation is successful, both the Java and the Patterns model are passed on to an ETL transformation that transforms the Java model to a UML model, a fragment of which is presented in Listing 8. The transformation encodes `<field, setter, getter>` triplets in the Java model as public properties in the UML model. As such, in line 6 of the transformation, the Patterns model is used to check whether field `s` has been matched under the `PublicField` pattern, and if so, the next line ignores the field’s declared visibility and sets the visibility of the respective UML property to `public`.

```java
rule FieldDeclaration2Property
transform s: Java!FieldDeclaration
to t: Uml!Property {
  t.name = s.getName();
  if (s.isTypeOf(Patterns!PublicFieldField)) {
    t.visibility = Uml!VisibilityKind#public;
  } else {
    t.visibility = s.toUmlVisibility();
  }
}
```

Listing 8. Fragment of the java2uml.etl Java to UMLETL transformation

As the Epsilon workflow provides ANT tasks for all its languages, the same technique can be used to pass the result of pattern matching on to model-to-text transformations, to model comparison and model merging programs, and even to subsequent EPL pattern matching programs in order to detect composite patterns.

At this point, it is worth stressing that although EPL has been demonstrated on EMF-based models in this paper in order to avoid duplication, it can be used to define and detect patterns on any other type of models supported by Epsilon (e.g. on XML documents [9])

V. CONCLUSIONS

In this paper we have presented the Epsilon Pattern Language, a textual language for specifying and detecting instances of structural patterns in models. EPL enables the definition of arbitrarily complex patterns by building on a powerful model querying language (EOL). Detected instances of patterns can be further processed (e.g. validated, transformed) using other languages of the Epsilon platform under a uniform and interoperable environment that facilitates code reuse and runtime interoperability. Moreover, EPL can be used to express patterns on models of diverse modelling technologies, and the same patterns can be evaluated on different modelling backends through the layer of indirection provided by the Epsilon Model Connectivity.

On the other hand, EPL is a dynamically typed language and as such, any type-related errors are only reported at runtime. The language run-time does not attempt to optimise the order in which patterns or roles are evaluated based on metamodel/model-level heuristics, as is for example the case in GrGen.NET and EMF IncQuery.

Initial performance evaluation experiments indicate that by using techniques such as memoisation (see Listing 3), the performance of EPL can be very similar to that of other interpreted languages such as GrGen.NET, ATL and EMF-IncQuery/VIATRA. In the case of EMF-IncQuery, we have only considered the two languages in non-incremental mode as EPL does not provide incremental pattern-matching capabilities. In future iterations of this work, we plan to conduct systematic comparative benchmarking that will enable us to accurately assess the performance of EPL against that of existing pattern matching languages.

Acknowledgements. This research was part supported by the EU, through the MONDO FP7 STREP project (#611125).

REFERENCES