Automated Analysis, Validation and Suboptimal Code Detection in Model Management Programs

Ran Wei
University of York
Deramore Lane, Heslington
York, United Kingdom
ran.wei@york.ac.uk

Dimitrios S. Kolovos
University of York
Deramore Lane, Heslington
York, United Kingdom
dimitris.kolovos@york.ac.uk

ABSTRACT

As MDE is increasingly applied to larger and more complex systems, the models that MDE platforms need to manage can grow significantly in size. Additionally, model management programs that interact with such models become larger and more complicated, which introduces further challenges in ensuring their correctness and maintainability. This paper presents an automated static analysis and validation framework for languages of the Epsilon platform. By performing static analysis on model management programs written in the Epsilon languages, this framework aims to improve program correctness and development efficiency in MDE development processes. In addition, by applying analysis on model management programs, sub-optimal performance patterns can be detected early in the development process and feedback can be provided to the developers to enable efficient management of large models.

1. INTRODUCTION

Model Driven Engineering (MDE) aims at raising the level of abstraction at which software is developed, by promoting models into first-class artefacts of the development process. For MDE to be applicable in the large and complex systems, a set of model management tools and languages are needed to enable developers to manage their models in an automated and efficient manner. Typically, model management tasks include validation, transformation, comparison, merging and text generation [7].

The Epsilon platform [7] is a platform which provides a broad range of model management languages built on top of an extensible common imperative model query and modification language called the Epsilon Object Language (EOL). EOL is an interpreted language that supports optional typing of variables, operations, and operation parameters. This provides a high degree of flexibility (for example, it enables duck typing [6]) and eliminates the need for explicit type casts, as typing-related information is currently only considered at runtime. On the other hand, the absence of a static analyser for EOL and the languages that build atop it, implies that a wide range of genuine errors that could be detected at design time are only detected at runtime.

Beyond detecting type-related errors, a static analyser could also be used to support advanced analysis capabilities, such as code detection, to improve the performance of model querying and transformation programs, particularly in the context of processing large models.

The remainder of the paper is organised as follows. In Section 2 we briefly motivate the need for static analysis capabilities in model management languages, in particular the Epsilon platform. In Section 3 we present the architecture of a static analysis framework for the Epsilon platform and a static analyser for the Epsilon Object Language. In Section 4 we present a sub-optimal code detection facility developed atop the static analysis framework. In Section 5 we discuss preliminary evaluation of our work. In Section 6 we discuss related work and in Section 7 we conclude and provide directions for further work.

2. BACKGROUND

MDE allows developers to construct models which abstract away from technical details, using concepts closely related to the domain of interest to reduce accidental complexity. The constructed models are then transformed into part (or all) of the target system under development. Whilst Model Transformation is considered to be the heart and soul of Model Driven Engineering [12], other model management operations are equally important. Typically, such operations include model validation, model merging, model comparison, etc.

2.1 Epsilon

Epsilon is a mature and well-established family of interoperable languages for model management. Languages in Epsilon can be used to manage models of diverse metamodels and technologies. The architecture of the Epsilon platform is depicted in Figure 1. At the core of Epsilon is the Epsilon Object Language (EOL) [9]. EOL is an imperative language which reuses a part of the Object Constraint Language OCL) but provides additional features such as multiple model access, statement sequencing and groupings, uniformity of function invocation, model modification, debugging and error reporting. Although EOL can be used as a general purpose model management language, its primary aim is to be reused in task-specific languages. Thus,

BigMDE '14, July 24, 2014 York, UK

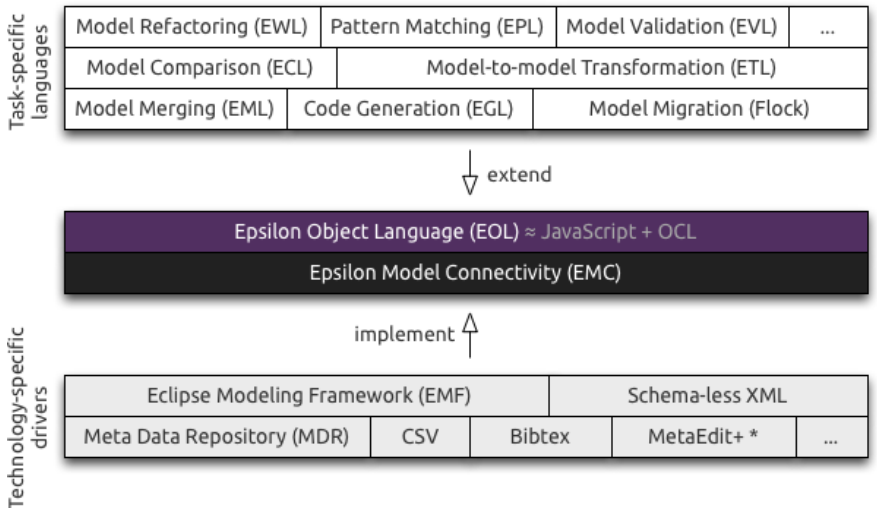

Figure 1: The basic structure of the Epsilon platform.

by extending EOL, a number of task-specific model management languages have been implemented, including those for model transformation (ETL), model comparison (ECL), model merging (EML), model validation (EVL), model refactoring (EWL) and model-to-text transformation (EGL).

Epsilon is metamodeling-technology agnostic [7], models written in different modelling languages can be managed by Epsilon model management languages via the Epsilon Model Connectivity (EMC) layer, which offers a uniform interface for interacting with models of different modelling technologies. Currently, EMC drivers have been implemented to support EMF [13], MDR, Z specifications in LaTeX using CZT and plain XML. Epsilon is an Eclipse project and is widely used both in the academia and the industry[1].

2.2 Epsilon and static analysis

EOL supports optional typing of variables, operations and operation parameters. The listing below demonstrates the flexibility that this delivers using as an example, a program that operates on an Ecore model. In line 1, an operation called *getName()* is defined, its *context type* is Any, which means that the operation can be invoked on objects/model elements of any type. In line 2, the keyword *self* refers to the object on which the operation is called, for example, in the statement *o.getName()*, *self* refers to *o*. Thus in line 2, the operation checks if the object is an instance of ENamedElement, if it is, it will return *self.name*. A typical static analyser would complain in line 3 that "self.name" does not exist because the type of *self* is declared as *Any*, not *Ecore!ENameElement*. However, this will never be a problem at run-time due to the *if* condition in line 2.

```
1  operation Any getName() {
2    if(self.isTypeOf(Ecore!ENamedElement))
3      return self.name;
4    else return "Unnamed";
5  }
```

[1]http://eclipse.org/epsilon/users/

On the other hand, some genuine errors can also remain hidden until runtime without the help of static analysis. The listing below demonstrates an example of a genuine error. In line 1, a variable named *o* is defined to be of type *Ecore!EClass*, and in line 2, *o* is assigned the value of a random EClass in the Ecore model. In line 3, the program tries to print the *value* property of *o* which does not exist according to the Ecore metamodel. As such, an error will be thrown at runtime.

```
1  var o : Ecore!EClass;
2  o = Ecore!EClass.all.first();
3  o.value.println();
```

As the size of such programs grow, locating genuine errors becomes an increasingly difficult task. For example, the EOL program that underpins the widely-used EuGENia tool [10] consists of 1212 lines of code, that query and modify 4 models that conform to 4 different metamodels concurrently. Performing code review on this code for genuine error detection is a time consuming process. Additionally, performing changes is also difficult, as developers have to manually identify and manage dependencies and relations between building blocks (*operations* for example) of such programs. Such tasks require effort and time [15]. For instance, to delete an operation named *F1* it is necessary to know if it is invoked by any other operations or code blocks. Manually performing this task is error-prone.

Since model management programs interact with models, performing static analysis on model management programs can also help identify sub-optimal performance patterns in the process of accessing models. Analysis can also help comprehend the model management programs. For example, coverage analysis can be performed to find out which parts of the metamodel of particular models are accessed, and test cases can be generated based on this comprehension to better test a model management program. With the potential benefits mentioned above, we propose and implement a static analysis framework for Epsilon.

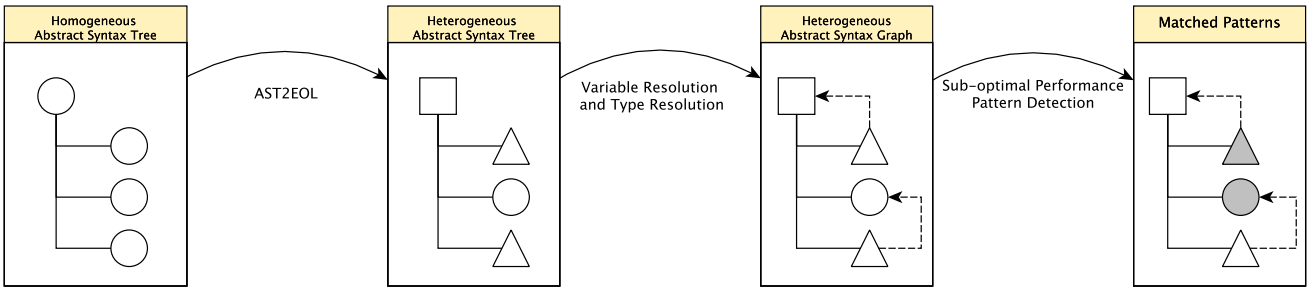

Figure 2: Detecting sub-optimal performance patterns from Abstract Syntax Trees.

3. TOWARDS A STATIC ANALYSIS FRAMEWORK FOR EPSILON

In this section we discuss the proposed static analysis framework in detail. The general idea of our approach can be illustrated by Figure 2. We first transform the Homogeneous Abstract Syntax Tree of an EOL program into a Heterogeneous Abstract Syntax Tree, we then apply resolution algorithms (including variable resolution and type resolution) to derive a Heterogeneous Abstract Syntax Graph, with its elements connected to each other. We then perform pattern detection to detect sub-optimal code. The aim of the framework is to provide a generic static analysis facility for all languages of the Epsilon platform. Since the core language of the Epsilon platform is EOL, we first develop a static analyser for EOL.

3.1 Deriving Heterogeneous Abstract Syntax Trees

Currently, Epsilon provides an ANTLR [7] based parser for EOL. The ANTLR parser produces homogeneous Abstract Syntax Trees (AST) with each node containing a text and an id, which the EOL execution engine consumes. To facilitate static analysis at a high level of abstraction, our first step was to define an EMF-based metamodel for EOL and to transform these homogeneous ASTs to models (heterogeneous trees) that conform to the EOL metamodel.

As EOL is a reasonably complex language, we only introduce the basic and novel parts of the EOL metamodel and the EOL standard library. Figure 3 lists a number of basic building blocks that constitute an EOL program. The fundamental element of the EOL metamodel is the *EolElement* metaclass, as all other metaclasses in the EOL metamodel directly or indirectly extend it, and contains information related to the line and column numbers of the respective text in an EOL program for traceability purposes. A *Program* contains a number of *Import*(s), which are used to import other programs. A *Program* also contains a number of *OperationDefinition*(s) which define additional operations/-functions on existing types. A *Program* contains a *Block*, which contains a number of *Statement*(s). *Expression* is also a fundamental element which is generally contained in *Statement*(s) and other *EolElement*(s). For each of the *Expression*, there is a *Type* associated to it. The type of the *Expression* is generally unknown at the time the source code of a program is parsed into an EOL model, but is resolved later in the type resolution process. In order to run an EOL program that involves processing models, Epsilon currently requires the user to select the required models/metamodels via a user interface at runtime. To facilitate accessing models at design-time for static analysis, we introduce the *ModelDeclarationStatement* to declare references to models in the source code. The syntax of a model declaration statement is as follows.

```
1  model library driver EMF {
2    nsuri = "http://library/1.0"
3  };
```

Like Epsilon, the static analysis framework is also technology-agnostic. As such, beyond the model's local name, a model declaration statement defines the type of the model in question (in this case EMF), as well as a set of model-type-specific key-value parameters (in this case nsuri = *http* : *//library*/1.0) that the framework can use to obtain the model's metamodel. We have currently implemented facilities to support models defined using EMF and plain XML. In the future we plan to extend the static analysis framework with support for other types of models supported by Epsilon, such as models defined in MDR, spreadsheets, etc.

With the metamodel defined, we developed a facility which transforms the ANTLR based ASTs into models that conform to the EOL metamodel. It should be noted that at the stage of AST to EOL model transformation, declared models are not inspected. So at this stage, for the statement

```
1  var book : Book
```

it is only known that variable *book* is of type *ModelElement-Type* whose *elementName* is *Book*. Later in the type resolution process, this information is used against declared models so that the *Book* type can be resolved.

Comparing with our current approach, an alternative approach would have been to redefine EOL's grammar using a toolkit such as Xtext or EMFText which can automate the source-code to model transformation process but we have opted for an intermediate transformation instead in order to reuse Epsilon's robust and proven parsers.

3.2 Deriving Heterogeneous Abstract Syntax Graphs

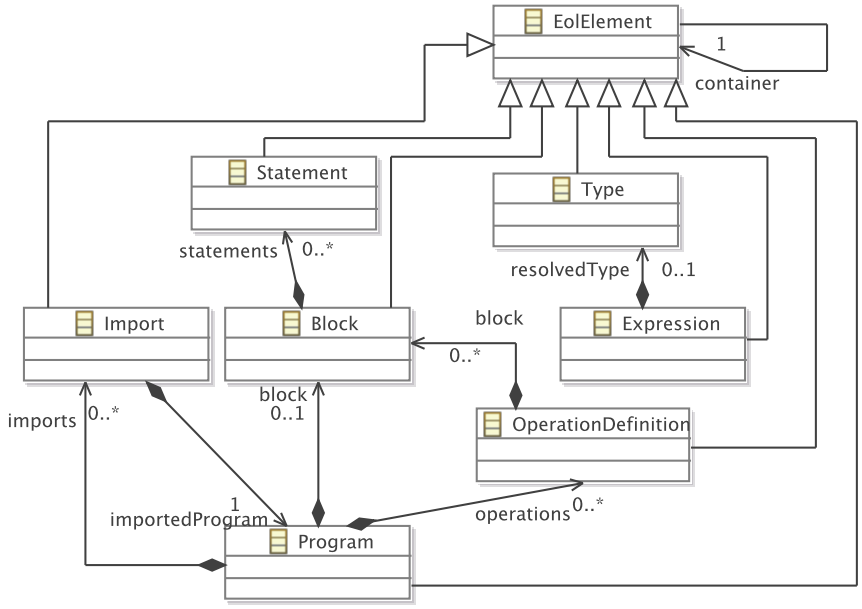

Figure 3: The basic structure the EOL metamodel.

With the EOL metamodel and the AST to EOL transformation defined, the next step of the process involves linking elements of the EOL model (heterogeneous tree) constructed in the previous phase to derive an abstract syntax graph. We have created several facilities to achieve this.

3.2.1 EOL Visitor

To facilitate the traversal of different elements in the EOL model and to support extensibility, we developed a facility which generates Java visitor classes from Ecore metamodels. We then generated visitor classes from the EOL metamodel which provide a general mechanism to traverse all elements in an EOL model. The EOL Visitor serves as the infrastructure of the static analysis framework, as all other facilities in the static analysis framework extend the EOL visitors to implement functionalities. The EOL visitor also provides high extensibility as new analysis mechanisms can be implemented simply by extending the EOL visitor.

3.2.2 Variable Resolver

The first phase of the static analysis on an EOL program involves resolving identifiers to their definitions. Context-free identifiers in EOL can refer to 1) declared variables/operation parameters and 2) undeclared variables provided by the execution engine at run-time. For declared variables, the variable resolver establishes links between the variable declaration and its references. For example, in line 1 of the listing provided below, a variable named a is declared. In line 2, a is referenced. The variable resolver will establish a link between the reference in line 2 and the declaration in line 1.

```
1  var a : Integer;
2  a = 10;
```

Variable resolution also applies to parameters in operation definitions. In the following listing, the variable resolver will establish a link between the reference of the parameter *toPrint* in line 2 and its declaration in line 1.

```
1  operation definedPrint(toPrint: String) : String {
2    toPrint.println();
3  }
```

There are also some implicit variables which are not declared by the developer but are rather provided by the execution engine. For example, the keyword $self$ in an operation definition refers to the object/model element on which the operation is invoked. The following listing demonstrates how $self$ is used. The variable resolver will establish a link between $self$ and the object on which $printSelf$ is invoked.

```
1  operation Any printSelf() {
2    self.println();
3  }
```

It is important to note that at the stage of variable resolution, model element types are not resolved.

3.2.3 Type Resolver

In EOL there are several built-in primitive types (*Boolean*, *Integer*, *Real*, *String*) and collection types (*Set*, *Bag*, *Sequence* and *OrderedSet*). There is also a built-in *Map* type and the *Any* type. These types are all subclasses of *Type* in the EOL metamodel. The resolution of the above types is performed during the heterogeneous abstract syntax tree derivation. There is also a subclass of *Type* in the EOL metamodel called $ModelElementType$ which includes typing information regarding models defined using different technologies. Such typing information should be determined by

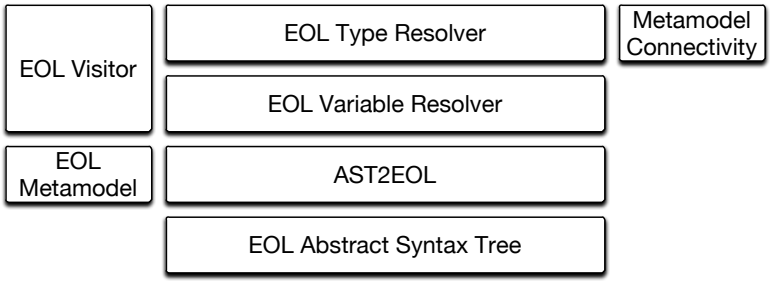

Figure 4: The architecture of the static analysis framework

accessing the corresponding models.

To support accessing metamodels at design-time, we introduced *ModelDeclarationStatement*s, which are flexible to support models defined in different modelling technologies. A model declaration defines the model's name, aliases, the modelling technology (driver) that the model conforms to, and a set of driver-specific parameters. The listing below is an example of *ModelDeclarationStatement*; it declares a model named *library* with alias *l* and its driver to be *EMF*, it then specifies the EMF-specific namespace URI (*nsuri*) of the model so that the analyser knows how to locate and access its metamodel.

```
1  model library alias l driver EMF {nsuri = "http://
       library/1.0"};
```

To facilitate uniform analysis of the structural information of models of diverse modelling technologies, the static analysis framework needs to convert type-related information from different modelling technologies into a common representation. Instead of inventing a new representation, we have decided to use EMF's Ecore. As such, the static analysis framework provides a modular architecture where pluggable components are responsible for transforming different types of model declarations into Ecore EPackages. For different modelling technologies:

- For EMF models, return the registered EPackage by looking up the metamodel nsURI property of the model declaration.
- For plain XML, construct an EPackage by analysing the contents of a sample model file specified by respective the model declaration parameter.

We have developed drivers for EMF models and plain XML files, and a common interface which allows new drivers for different modelling technologies to be integrated with the static analysis framework. By accessing models/metamodels, the type resolver is able to resolve types with regards to models/metamodels.

The variable resolver and type resolver constitute the infrastructure of the static analysis framework for the Epsilon languages. The infrastructure is depicted in Figure 4. The EOL Abstract Syntax Tree layer is provided by the EOL engine, the AST2EOL layer uses the AST and the EOL metamodel to translate the AST to an EOL model. The EOL Variable Resolver and the EOL Type Resolver, both make use of the EOL Visitor and the Metamodel Connectivity layer (which is used to read metamodels) to establish a fully type-resolved EOL model.

The static analysis infrastructure can be easily extended. As proof of concept, we have also implemented all of the aforementioned facilities for the Epsilon Transformation Language. We extended the EOL metamodel to create an ETL metamodel, with the ETL metamodel, we created the ETL visitor facility; we extended the AST2EOL to create a AST2ETL facility; we extended the EOL variable resolver and type resolver to create ETL variable and type resolvers. The EOL and ETL static analysers can be found under the Epsilon Labs open-source project [1].

4. SUBOPTIMAL CODE DETECTION

Rule-based model transformation languages usually rely on query or navigation languages for traversing the source models to feed transformation rules with the required model elements. In [11], the authors suggest that in complex transformation definitions, a significant part of the transformation logic is devoted to model navigation. In the context of large-scale MDE processes, models can contain several millions of elements. Thus, it is important to retrieve desired elements in an efficient way. On top of the static analysis framework, we have built a facility which is able to detect sub-optimal performance patterns when navigating and retrieving model elements. This facility performs pattern matching to detect potential computationally heavy code in EOL (and potentially all Epsilon languages). It does so by matching patterns defined in the Epsilon Pattern Language (EPL) [8] against fully resolved EOL abstract syntax graphs.

The structure of this facility is depicted in Figure 5. The *SubOptimalDetector* has a *EOLModel* as input to perform the detection; it makes use of the *EPLEngine* of the Epsilon platform to derive Abstract Syntax Trees, it has a set of defined *EPLPattern*s (.epl scripts) using EPL, and a logging facility (*LogBook*) to keep the warning messages it generates for pattern matches.

In this section, we present the sub-optimal detection facility. We provide several examples that illustrate potential sub-optimal performance patterns in the context of large scale model manipulation. We then present and explain a sub-

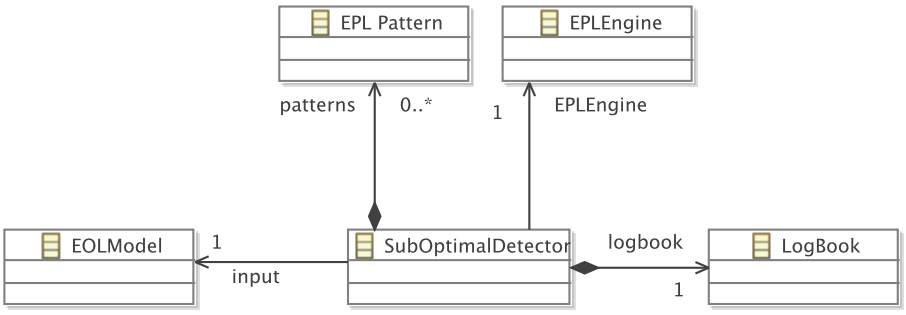

Figure 5: The structure of the sub-optimal performance detection facility

optimal performance pattern defined in EPL. It should be noted that this facility targets EOL programs, however, it can be easily extended to cope with programs written in other Epsilon languages as discussed earlier.

The examples we present are all based on a simple *Library* metamodel illustrated in Figure 6. The Library metamodel contains two simple metaclasses, *Author* and *Book*. An *Author* has a *first_name*, a *surname* and a number of published *Book*s where a *Book* has a *name* and an *Author*. The association between *Author* and *Book* is bidirectional, they are *books* and *authors* respectively.

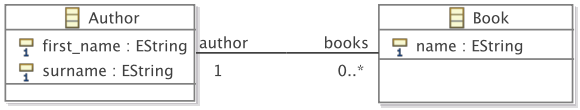

Figure 6: The Library metamodel

4.1 Inverse navigation

A frequent operation in EOL is to retrieve all model elements of a specific type by using the *.all* property call which can be a computationally heavy operation to perform as models grow in size. By analysing the metamodel of the model under question, bidirectional relationships between model elements can be used to avoid such heavy computations.

```
1  var a = Author.all.first;
2  var books = Book.all.select(b|b.author = a);
3  var aBook = Book.all.selectOne(b|b.author = a);
```

The listing above demonstrates a potential performance bottleneck. In line 1, an *Author* is retrieved from the model. In line 2, all instances of type *Book* are retrieved and then a conditional *select* is performed to find the books that are written by *Author* 'a'. However, since the relationship between *Author* and *Book* is bidirectional, this can be replaced by the (more efficient) statement:

```
1  var books = a.books;
```

Thus the complexity of the operation *all* is reduced from n to 1 given that n is the number of *Book*s in the model under question. It is also the case for the *selectOne* operation in line 3, which can be rewritten as:

```
1  var aBook = a.books.first();
```

4.2 Compound select operations

Another computationally-heavy pattern is the presence of compound *select* operations on the same collection.

```
1  var authors = Author.all.select(a|a.first_name =
2    'William').select(a|a.surname = 'Shakespeare');
```

Listing 1: a potential performance overhead using compound select operations

Listing 1 demonstrates such operations. In line 1, all of the *Author*s are retrieved first, then a *select* operation is performed to select all *Author*s whose first_names is *William*, then another *select* operation is performed to select all *Author*s whose surname is *Shakespeare*. The complexity of this operation is n^2 given that n is the number of *Author*s in the model under question. However, the condition of both the *select* operations can be put together to form a single *select* operation. And the statement above can be written as

```
1  var authors = Author.all.select(a|a.first_name =
2    'William' and a.surname = 'Shakespeare');
```

the complexity of this operation is therefore n as the collection of the *Author*s is only traversed once.

4.3 Select operation on unique collections

Performing *select* operations on unique collections (sets) can sometimes be inefficient depending on the *condition* of the *select* operation.

```
1  var authorOne = Author.all.first;
2  var authorTwo = Author.all.last;
3  var bookOne = authorOne.books.first;
4  var bookSet : Set(Book);
5  bookSet.addAll(authorTwo.books);
6  bookSet.select(b|b = bookOne);
```

Listing 2: Select operation on unique collection

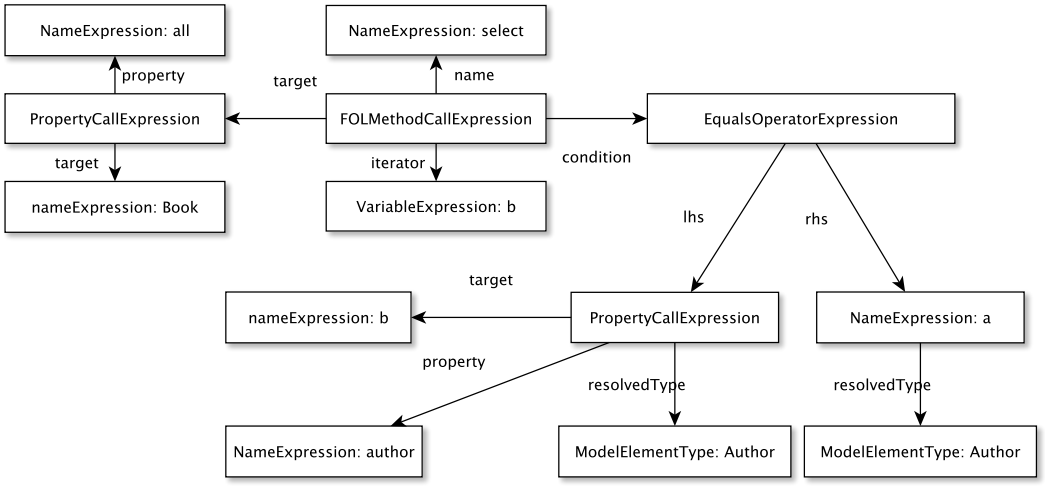

Figure 7: The model representation for $Book.all.select(b|b.name = a)$

Listing 2 demonstrates an inefficient and computationally expensive *select* operation. In Line 1 and 2, two *Author*s are retrieved from the model; in line 3, a *Book* is retrieved from *authorOne*'s publications; in line 4, a *Set* called *bookSet* is created and in line 5, all of the *Book*s that *authorTwo* published are added to *bookSet*. In line 6 the *select* operation iterates through all of the books in the *bookSet* and find the ones that match the *bookOne*. However, the *bookSet* is a unique collection, which means that all of the elements in it only appear once. Therefore, it is not necessary to perform a *select* operation but rather a *selectOne* operation, as the *select* operation would return at most one result eventually. The complexity of the *select* operation is n given that n is the number of books that *authorTwo* published; If the *select* operation is replaced with *selectOne*, the complexity of it would be 1 for the best case scenario and n for the worst case scenario ($n/2$ for the average case).

4.4 Collection element existence

In some cases, checking existence of an element inside a collection can be written in inefficient ways.

```
1  if(Book.all.select(b|b.name = "EpsilonBook")
2    .size() > 0) {
3    "There is a book called EpsilonBook".println();
4  }
```

Listing 3: Collection element existence

Listing 3 demonstrates such a scenario. In line 1, the condition of the if statement retrieves all instances of *Book*, then selects the ones with the name *EpsilonBook*, and calculates the size of it then evaluates if the size is greater than 0. This operation eventually checks for the existence of a book named *EpsilonBook*. Thus, this operation can be more efficiently re-written as:

```
1  Book.all.exists(b|b.name = "EpsilonBook")
```

4.5 Select the first element in a collection

Listing 4 demonstrates another example of sub-optimal EOL code.

```
1  var anEpsilonBook = Book.all.select(b|b.name =
2    "EpsilonBook").first();
```

Listing 4: Select an element in a collection

In line 1, a *select* operation is performed on all of the instances of *Book* to filter out the books with the name 'EpsilonBook', then a $first$ operation is performed to select the first one of the collection returned by *select*. This can be more efficiently re-written as:

```
1  var anEpsilonBook = Book.all.selectOne(b|b.name =
2    "EpsilonBook");
```

to avoid traversing all of the instances of *Book*.

4.6 A sub-optimal performance pattern

In this section we present a sub-optimal performance pattern which is written in the Epsilon Pattern Language (EPL). To understand how this pattern works, it is first important to understand what is contained in an EOL model for a certain EOL program.

4.6.1 Understanding an EOL model

Figure 7 illustrates a fragment of an EOL model which represents the statement below.

```
1  Book.all.select(b|b.author = a);
```

Firstly, invocations of the *select*() operation in the EOL metamodel are represented by the $FOLMethodCallExpression$ (FirstOrderLogic method call) metaclass; it has a *name* (an instance of $NameExpression$) and an *iterator* (an instance of $VariableExpression$). In this case, the *name* is 'select' and the *iterator* is 'b'.

The *select* operation has a *condition*, in this case, it is an instance of *EqualsOperatorExpression*. The lhs (left hand side) of it is an instance of *PropertyCallExpression*, whose *target* (an instance of NameExpression) is 'b' and *property* (an instance of NameExpression) is 'author'; the rhs (right hand side) of it is 'a' (an instance of NameExpression). Both the lhs and rhs of the *EqualsOperatorExpression* have resolvedTypes, in this case, they are both *Author* (instances of *ModelElementTypes*).

The *target* of the *FOLMethodCallExpression* is an instance of *PropertyCallExpression* with its *target* as *Book* (an instance of NameExpression) and its *property* as *all* (an instance of NameExpression). The types of these expressions, altogether with some irrelevant details are omitted for the purpose of the discussion.

4.6.2 The EPL pattern

In Listing 5, we define an EPL pattern to match occurences of the pattern described above. In lines 2-6, a *guard* is defined to look for a *FOLMethodCallExpression* the name of which is either 'select' or 'selectOne'; the type of the condition should be *EqualsOperatorExpression*; its target should be an instance of *PropertyCallExpression*; and the property of the *PropertyCallExpression* should be 'all'.

In lines 8-10, a *guard* is defined to look for an instance of *EqualsOperatorExpression* in the condition of the FOLMethodCallExpression found previously, the *lhs* of which should be an instance of *PropertyCallExpression*.

Lines 12-14 specify that the *resolvedType* of the *lhs* should be an instance of *ModelElementType*. In lines 16-18, it specifies that the *resolvedType* of the *rhs* should be an instance of *ModelElementType*. In lines 20-24, it specifies that the type of the *lhs* and the *rhs* should be the same.

Lines 26-37 perform the match of the pattern. This part firstly fetches the *EReference* from the *lhs* of the condition (in this case, 'b.author', it is an *EReference* because as previously discussed, all metamodels are converted to EMF metamodels for uniformity). The EReference is then inspected; if it is not null and it has an eOpposite reference, the pattern continues to check if the type of the eOpposite of the reference is the type of the *rhs* of the condition (in this case, 'author').

In lines 39-47, a helper method is defined to help look for an *EReference* given an *EClass* and a name; its implementation is straightforward.

```
1  pattern InverseNavigation
2    folcall : FOLMethodCallExpression
3    guard: (folcall.method.name = 'select' or folcall.
         method.name = 'selectOne')
4    and folcall.conditions.isTypeOf(
         EqualsOperatorExpression)
5    and folcall.target.isTypeOf(PropertyCallExpression)
6    and folcall.target.property.name = 'all',
7
8    condition : EqualsOperatorExpression
9    from: folcall.condition
10   guard: condition.lhs.isTypeOf(
         PropertyCallExpression)
11
12   lhs : PropertyCallExpression
13   from: condition.lhs
14   guard: lhs.resolvedType.isTypeOf(ModelElementType),
15
16   rhs : NameExpression
17   from: condition.rhs
18   guard: rhs.resolvedType.isTypeOf(ModelElementType),
19
20   lhsType : ModelElementType
21   from: lhs.resolvedType,
22   rhsType : ModelElementType
23   from: rhs.resolvedType
24   guard: lhsType.ecoreType = rhsType.ecoreType
25   {
26     match {
27       var r = getReference(lhs.target.resolvedType.
            ecoreType, lhs.property.name);
28       if(r.upperBound = 1 and r.eOpposite <> null and
             r <> null)
29       {
30         if(r.eOpposite.eType = lhs.target.resolvedType
               .ecoreType)
31         {
32           return true;
33         }
34       }
35       return false;
36     }
37   }
38
39 operation getReference(class: Any, name:String)
40 {
41   for(r in class.eReferences)
42   {
43     if(r.name = name)
44       return r;
45   }
46   return null;
47 }
```

Listing 5: EPL pattern for inverse navigation

4.6.3 The Java pattern

Our original attempt to construct the sub-optimal performance detection facility was to define patterns using Java, we defined a method in Java to achieve the same function described above. The equivalent Java implementation is 76 lines of code with a long chain of If statements which makes it very difficult to comprehend. With the EPL approach, the patterns are more comprehensible. Developers can contribute to the pattern pool by defining their own EPL patterns and registering them with the framework through appropriate extension points.

5. TOOL SUPPORT

The static analyser proposed in this paper is a pragmatic static analyser. The flexibility of EOL allows its users to write code with optional typings that always work at runtime. Reporting errors on such cases are not desirable for EOL, especially on legacy EOL code that is proven to work with extensive testing. Thus the design decision was to allow such behaviour and delegate the resolution to the EOL execution engine. We applied the static analyser on a large EOL program (Ecore2GMF.eol) that underpins the Eugenia tool [10] for evaluation. We allow optimistic typing - when Any type is encountered in assignments, operation calls and property calls, we provide a warning message to

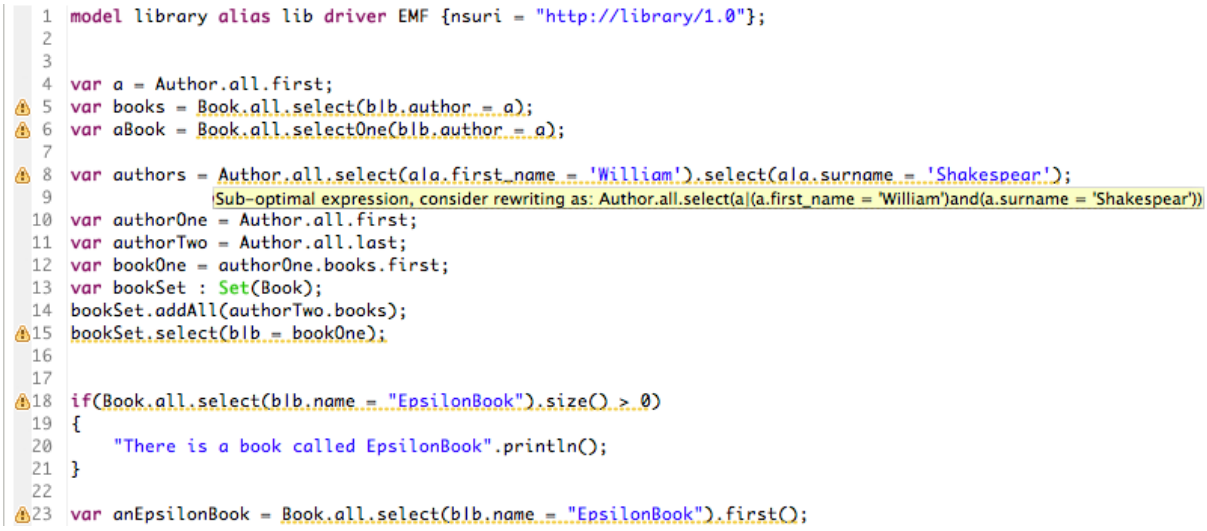

Figure 8: EOL Editor Screen Shot

notify the user that there might be potential type incompatibility issues at run-time. With this configuration, analysis on Ecore2GMF.eol which consists of 1212 lines of code generates 126 warning messages. This result shows that the static analyser supports plausible legacy code. At the same time, it provides reasonable warning messages when optional typing is used.

After evaluating the static analyser, we evaluate the sub-optimal performance detection facility. Figure 8 provides a screenshot of the editor we implemented by extending the existing EOL Eclipse-based editor. The lines of code with warnings represent matches of the patterns discussed above. The implementation of the editor is able to extract and display the warning messages generated by the detection facility. The sub-optimal performance detection facility is not only able to detect patterns that incur performance overheads, but also provide suggestions on how to rewrite the code. An example warning message is shown in line 8, the warning message suggests to rewrite the operation as:

```
1 Author.all.select(a|a.first_name = "William" and a.
      surname = "Shakespeare")
```

The rest of the patterns function as expected.

6. RELATED WORK

There are several automated analysis and validation tools for model management programs. In [14] the authors propose a generic static analysis framework for model transformations specified in VIATRA2 Textual Command Language (VTCL [2]). The latest static analysis framework detects common errors and type related errors regarding models. However, the VIATRA2 framework provides limited support for other metamodelling technologies as it uses its own modelling language (VTML) and store the metamodels in the model space. Additionally, VTCL is not as flexible as EOL; it does not provide optional typing mechanisms as EOL does.

Acceleo [4] provides static analysis mechanisms for syntax highlighting, content assistant, and model related error detection. However, to the best of our knowledge, it does not support modelling technologies other than EMF.

Xtend [3] also provide static analysis facilities which are used to detect syntax and built-in type-related errors, model related type information validations are not included.

In [15], a static analysis tool is proposed to detect errors in model transformations written in the Atlas Transformation Language (ATL), the tool presented is used to convert an ATL program into a model, but no validation algorithms are implemented on this tool to our best knowledge.

The latest release of ATL IDE [5] provides a static analysis facility, it resolves the types of variables including built-in ATL types and types related to metamodels. The ATL IDE also provides code-completion of operation calls and metamodel element navigations. However, the static analysis is not responsible to provide any errors on type incompatibilities as it adopts an optimistic and flexible approach. The ATL platform also provides limited support for multiple modelling technologies other than EMF.

In [11], ways of deriving optimisation patterns from benchmarking OCL operations for model querying and navigation are proposed and several optimisation patterns are identified, including short-circuit boolean expressions, opposite relationship navigation, operations on collections, etc.

7. CONCLUSIONS AND FUTURE WORK

In this paper we have reported on our work on designing and developing a static analysis framework for the core language of Epsilon (EOL). The focus of our report is mostly on the sub-optimal detection facility of the static analysis framework. However, it is to be noted that the static analysis framework is able to detect various type-related errors that may occur using EOL. The static analysis framework follows a pragmatic approach so as not to compromise the flexibility

of the Epsilon languages. As a result, it can generate false negatives (problems that exist but cannot be detected by the static analyser). To minimise the number of false negatives, a more strict coding style is encouraged - to avoid the use of *Any* type as much as possible, so that the static analyser can perform more accurate analysis. This is clearly a trade-off to make; to obtain better error reporting, developers need to write more boilerplate code with explicit type casting, while to obtain better flexibility, developers need to bare with the fact that the analyser may produce false negatives that emerge at run-time.

It should be noted that the sub-optimal performance detection facility is only one application of the static analysis framework for Epsilon. In the future, we plan to look into facilities such as program comprehension, metamodel coverage analysis, impact analysis, etc. We will also look into the possibility of pre-loading models and look for more fine-grained performance patterns for EOL programs.

8. REFERENCES

[1] Epsilon labs. `https://epsilonlabs.googlecode.com/svn/trunk/StaticAnalysis/`.

[2] A. Balogh and D. Varró. Advanced model transformation language constructs in the VIATRA2 framework. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 1280–1287. ACM, 2006.

[3] P. Friese and B. Kolb. Validating Ecore models using oAW Workflow and OCL. *Eclipse Summit Europe*, 2007.

[4] J.-M. Gauthier, F. Bouquet, A. Hammad, F. Peureux, et al. Verification and Validation of Meta-Model Based Transformation from SysML to VHDL-AMS. In *MODELSWARD 2013, 1st Int. Conf. on Model-Driven Engineering and Software Development*, pages 123–128, 2013.

[5] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A model transformation tool. *Science of computer programming*, 72(1):31–39, 2008.

[6] A. Koenig and B. Moo. Templates and duck typing. *Dr. Dobbs, June*, 2005.

[7] D. Kolovos. *An extensible platform for specification of integrated languages for model management.* PhD thesis, University of York, 2008.

[8] D. Kolovos, L. Rose, R. Paige, and A. Garcıa-Domınguez. The Epsilon Book. *Structure*, 178, 2010.

[9] D. S. Kolovos, R. F. Paige, and F. A. Polack. The Epsilon Object Language (EOL). In *Model Driven Architecture–Foundations and Applications*, pages 128–142. Springer, 2006.

[10] L. M. Rose, D. S. Kolovos, and R. F. Paige. Eugenia live: a flexible graphical modelling tool. In *Proceedings of the 2012 Extreme Modeling Workshop*, pages 15–20. ACM, 2012.

[11] J. Sánchez Cuadrado, F. Jouault, J. García-Molina, and J. Bézivin. Deriving ocl optimization patterns from benchmarks. *Electronic Communications of the EASST*, 15, 2008.

[12] S. Sendall and W. Kozaczynski. Model transformation: The heart and soul of model-driven software development. *Software, IEEE*, 20(5):42–45, 2003.

[13] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: eclipse modeling framework.* Pearson Education, 2008.

[14] Z. Ujhelyi, A. Horváth, and D. Varró. A generic static analysis framework for model transformation programs. Technical report, Technical report, Budapest University of Technology and Economics, 2009.

[15] A. Vieira and F. Ramalho. A static analyzer for model transformations. In *3rd International Workshop on Model Transformation with ATL, Zurich, Switzerland*, 2011.