Proceedings of the Federated Conference on
Computer Science and Information Systems pp. 965–975

DOI: 10.15439/2015F179
ACSIS, Vol. 5

Unified Compile-Time and Runtime Java Annotation Processing

Peter Pigula and Milan Nosáľ
Technical University of Košice
Letná 9, 042 00 Košice, Slovakia
Email: peter.pigula@student.tuke.sk, milan.nosal@gmail.com

***Abstract*—Java provides two different options for processing source code annotations. One of them is the annotation processing API used in compile time, and the other is the Reflection API used in runtime. Both options provide different API for accessing program metamodel. In this paper, we examine the differences between those representations and we discuss options on how to unify these models along with advantages and disadvantages of this approach. Based on this proposal, we design a unified Java language model and present a prototype tool which can populate a unified model during both compilation and runtime. The paper includes the designed API of this unified language model. To verify our approach, we have performed experiments to show the usability of the unified metamodel.**

I. Introduction

SINCE the year 2004, when annotations were first introduced to Java, this programming technique is on the rise. More and more programs use annotations during compile time for code generation [1], [2], and in runtime for configuration and reflection [3]. In spite of the fact that usage of annotations in both instances is often very similar, means to access annotations and program model, as well as their representations, are different. If a developer needs to complete the same task in both compile time and runtime, she has to be familiar with both of them, and would have to create two different versions of the source codes. That is because code using annotation processing API cannot be used during runtime, and vice versa.

We came across this problem during our previous work. We designed a tool (*Bridge to Equalia* [4]) that provided an abstraction layer to two most common configuration formats in Java – XML and source code annotations. *Bridge to Equalia* (abbreviated: *BTE*) uses XML to annotations mapping patterns [5] to shorten the configuration interface code in cases when both annotations and XML should be accepted interchangeably. To query the annotations, *BTE* uses Java Reflection API in combination with the Scannotation [6] library. Therefore, it works only in runtime (Reflection API requires compiled classes). As we wanted to use *BTE* also in projects that work during compile time, we faced the problem of working with different APIs for annotation processing during in runtime and in compile time.

This work was supported by VEGA Grant No. 1/0341/13 "Principles and Methods of Automated Abstraction of Computer Languages and Software Development Based on the Semantic Enrichment Caused by Communication".

Algorithm 1 Checking whether a field is public during runtime

```
boolean isFieldPublic (Field field) {
  if (Modifier.isPublic(field.getModifiers())) {
    return true;
  }
  return false;
}
```

Algorithm 2 Checking whether a field is public during compile time

```
boolean isFieldPublic (Element field) {
  for (Modifier modifier : field.getModifiers()) {
    if (modifier == Modifier.PUBLIC) {
      return true;
    }
  }
  return false;
}
```

BTE is a medium size project with a complex implementation, therefore we will not use its source code as a case study for the illustration of the problem. Just as an example of the APIs' diversity we can take a look at the representation of the accessibility modifiers of Java program elements. During runtime, modifiers are encoded to a single integer value. Every bit of this integer represents one modifier. On the other hand, during compile time modifiers are represented as a set of enumeration types `Set<Modifier>`. Each element of this set represents one modifier. This means that for example a method, which checks whether a field is public or not, would be different in both runtime and compile time. Implementation of such a method during runtime is presented in the algorithm 1. Implementation of the same method in compile time is presented in the algorithm 2. We have to note that although both listings use the `Modifier` type, they are both different and belong to different packages. Of course, the *BTE* implementation works not just with modifiers, but many other types of metadata (program elements' names, annotations, annotations' parameters, code tree structures, etc.).

In this work we want to *share our experience with designing and implementing a tool that provides a unified API to Java source code model during runtime and compile time*. We will discuss challenges in the design that we faced, and we

will present the designed API (that was also implemented and can be found at https://github.com/mallynth/UModel). We conclude with a simple experiment that evaluates the tool's performance against standard APIs.

Since there is currently no unified representation in Java, developers who come across this problem are left with only a single option, which is to create two versions of the source code dealing with the same problem in different representations. In small tasks, this option is not that time consuming. But with increasing complexity of the processed annotations it may lead to code difficult to manage, maintain and evolve. In cases like this, unified representation could reduce the needed work by half, and apply the Single point of responsibility principle.

Having two versions of the code accessing annotations goes against *Single point of responsibility* principle. This leads to potential problems during evolution and maintenance (as we faced with the *BTE* tool). Suppose we have two programs that are used for the same purpose, but one of them is used during compile time, and the other during runtime. Degree of difference between these programs would not be very high, because they were developed using the same algorithm, and they just use different APIs. Now let us suppose that we need to implement a change of the base algorithm. That means we need to change both versions of the code separately from one another in order to implement the desired change. This increases the risk of a bug, e.g., when a developer forgets to make a change in one of the versions, or she misses an important difference in APIs. This leads to overhead in testing, because again, both versions need to be tested separately.

Cepa [7], [8] deals with similar problem about the representation of metamodel in programming languages. He recognizes the need for a generalized representation of the program by using a graph of metadata nodes that could be used in compile time, loading time and runtime. He then proposes a Generalized and Attributed Abstract Syntax Tree (GAAST), which is a syntax tree of the language enriched by annotations. The recognition for the need of GAAST support stemmed from his work on his framework MobCon [9], which is a framework used to generate mobile applications in Java. His work in combination with our previous work with *BTE* was the main inspiration for this work.

Contributions of this paper are following:

- we analyse options by which unification of both program model representations in Java can be achieved,
- we discuss and explain the pitfalls that need to be considered during the design and implementation of the unified model, and
- we present the designed API for projecting standard source code models into a unified Java program model during both compile time and runtime.

II. Existing Program Models

Annotations in Java are used as a source code decoration mechanism. It means that they do not directly influence the control or data flow of the program that is annotated. A study [10] performed on the curated collection of programs Qualitas Corpus [11] showed that more than 60% of analyzed programs are using annotations. That includes systems developed before annotations were introduced to Java. This study provides evidence that annotations are an important part of Java.

Algorithm 3 Declaration of annotation with source code retention

```
@Retention(RetentionPolicy.SOURCE)
public @interface SourceAnnotation {}
```

Algorithm 4 Declaration of annotation with runtime retention

```
@Retention(RetentionPolicy.RUNTIME)
public @interface RuntimeAnnotation {}
```

Annotations can be used to express metadata of a program. They can be accessed by the developer in two different ways:

- during **compile time** using annotation processing API, and
- during **runtime** using Reflection API.

Both of these ways offer metadata to the developer in a different representation. Both representations are basically *program models* that include all metadata available at the given time. Program model available during compile time is represented by classes in a package `javax.lang.model.element` [12] and the model available during runtime is represented by classes in a package `java.lang.reflect` [13]. These models are accessed and queried using different APIs. When their difference is not important for the sake of discussion, we will refer to them simply as **basic models**.

Probably, the main reason of their difference is that they are used in a different phase of a program life cycle and different metadata are available during those phases. However, they are still similar since they represent the same program structure.

To illustrate the difference, we can mention annotations marked by the `@Retention` metaannotation. In the code fragment 5, the `SomeClass` class is annotated with two annotations `SourceAnnotation` and `RuntimeAnnotation` which are declared in code fragments 3, and 4 respectively. In this example, if we would access the metamodel during compilation, we would get both annotations. However, if we would access the metamodel during runtime, we would find out that the `SomeClass` class is annotated only with the `RuntimeAnnotation` annotation. That is because `SourceAnnotation` is marked with meta-annotation `@Retention(RetentionPolicy.SOURCE)` to indicate that it should be discarded after annotation processor finishes, and therefore it will not be available during runtime.

Another difference is the option for running methods of the code. In reflection, the developer can invoke methods unknown during compile time using reflection API. However, in compile time the API does not support invoking of processed code, since in that time the classes have not yet been compiled.

Algorithm 5 Usage of annotations

```
@SourceAnnotation
@RuntimeAnnotation
public class SomeClass {
  ...
}
```

The differences do not end with different models. It is very common that the developer only needs some metadata that are present in both models, e.g., names of classes that are marked with a specific annotation. The problem is that the API for acquiring those metadata are different in both models, in spite of the fact that they are the same metadata. In some cases, in addition to the method of acquiring metadata being different, the representation of the metadata is also different (as in case of modifiers discussed in the introduction).

Examples of similar elements in both models can include names of classes, their methods or hierarchy of classes.

A. Annotation Processing Program Model

Annotation processor is executed before compilation of the program and the execution itself is divided into separate rounds. Annotation processor can acquire metadata that are present in source code as well as metadata that were created in previous rounds of this processor execution.

Main class, that is used to represent entities in the program model for this API is the `Element` class. Instances of this class can represent multiple different entities, such as classes, interfaces, methods or fields.

1) Accessing Metadata in Annotation Processor: The main method that every annotation processor must have is the `process` method. As its name suggests, this method is doing the annotation processing. One of the parameters of this method is an environment of current round represented by the `RoundEnvironment` class. From this environment, the developer can acquire metadata that she needs.

One of the methods for acquiring metadata from round environment is by using the method `getRootElements`. This method returns list of all root elements of the program that are represented by `Element` class. The `Element` class has a field `kind` which defines what this element represents (class, interface, method etc.). By querying these elements, it is possible to find metadata that are needed.

Different way would be to use the `getElementsAnnotatedWith` method which returns a list of elements that are marked with a specific annotation. When using this method, there is a useful annotation `@SupportedAnnotationTypes`, which can be used to mark the annotation processor class. It specifies annotation types which annotations are supposed to be retrieved from the source codes during annotation processing.

B. Reflection Program Model

Reflection can be used during runtime and it provides a way for accessing the metadata of the running program.

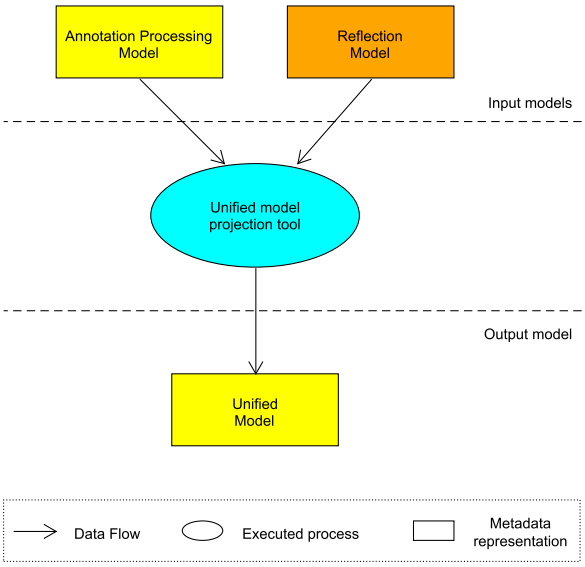

Fig. 1. Concept of the unified API idea

1) Accessing Metadata in Reflection: To gain access to metadata provided by reflection in a specific class, one can use `ClassName.class` parameter of every class which returns a `Class` class that represents a class with the name "ClassName". This class can be then used to retrieve metadata about the class and elements defined inside the class, such as methods, constructors, fields or annotations, or even inner classes. It is possible to retrieve lists of those elements or search for specific ones. For example retrieving the list of annotations by which the class is marked can be done by calling a method `getDeclaredAnnotations`.

By calling methods provided by classes of the reflection model, developer can acquire any metadata existing during runtime. One of the big differences in usability of the two basic models is that in reflection, methods can be executed, which is impossible in annotation processing.

III. Program Model Unification

Due to the similarities in these models, we expected that it should be possible to design a tool that would be able to provide a unified API to data from both basic models. Simple overview of the idea is illustrated in Fig. 1. It is enough to note that input data used in the tool will always come from only one of the input models at the time. Which model it is depends on the time when the tool is used.

A. Unified Model Projection Types

In order to create a unified model, first we need to determine how the unification tool will project basic models into the unified API.

We will illustrate the problem of unified model types on program trees. A node in the tree represents a program element (class, method, etc.) and annotations. The structure models the

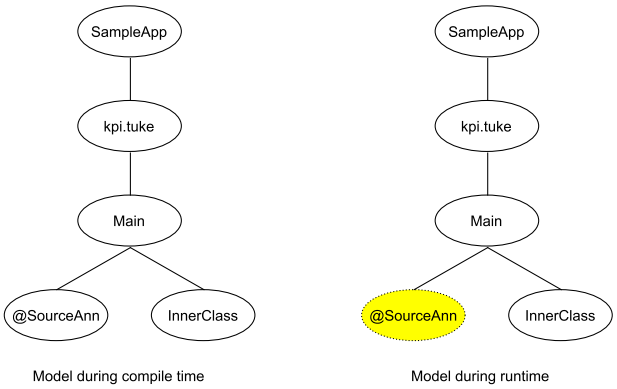

Fig. 2. Model comparison illustration

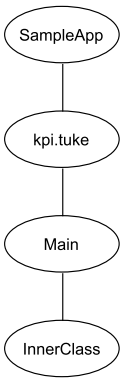

Fig. 3. Equality-based projection

encapsulation relationships of program elements. The tree's root represents Java project (library, application), its children are Java packages, the packages contain classes, or might be annotated by annotations, the nodes representing classes have children nodes representing variables and methods, and again annotations that belong to the classes, and so on. E.g., we might have a root node 'sampleApp' with a single child node 'kpi.tuke' representing package. The 'kpi.tuke' node could have a child node representing the 'Main' class, and this 'Main' node could have two child nodes: a node for the 'InnerClass' class, and a node for the '@SourceAnn' annotation, etc. Program elements and annotations are source code entities that the unified API should expose.

An illustration of both basic models for the discussed example is presented in Fig. 2. Nodes that are **equal** in both models are represented by ellipses with white color and solid border. Equal in this context means that both basic models expose the same metadata about the element in both models.

Node '@SourceAnn', which is represented by an ellipsis with yellow color and dotted border in a model during runtime shows a situation when element **does not exist** in runtime model, but exists in the compile time model. This is a result of using the `@Retention(RetentionPolicy.SOURCE)` meta-annotation that marks the `@SourceAnn` annotation to be discarded by the compiler during the compilation.

Projection to the unified model can be designed in different ways depending on which metadata would be included in it. There are three main approaches to design of a unified API:

1) **equality-based** projection,
2) **combined** model projection, and
3) **problem-specific** model projection.

First, the **equality-based** projection for the discussed example is shown in Fig. 3. In this type, the unified model includes only those elements and metadata, that are available in both basic models. That means node '@SourceAnn', which does not exist in runtime model, will not be projected to the unified model either (regardless whether created during runtime or compile time).

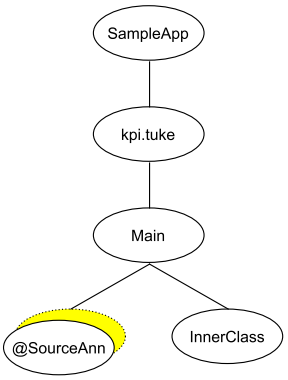

Fig. 4. Combined model projection

Second, the **combined** model projection is shown in Fig. 4. In this type the unified model exposes all the elements and their metadata that were included in the basic model. For example, if the unified model was created during compile time then the unified model would expose all the metadata that are available during compile time, including those that are not available during runtime. This projection type deals only with different APIs, and does not regard the differences of information exposed by basic models (the equality-based projection hides the differences in exposed information about the source code).

Third, the **problem-specific** projection covers cases when the projection is optimized for a given problem (something like a DSL [14] in languages). Basically it represents ad-hoc solutions for a family of problems. E.g., it might be a case of a simplified model, restricted in the tree depth to classes (therefore the only child nodes of the class nodes would be nodes for their annotations). We will not discuss this type of projections further, since its applicability is limited to a restricted set of problems.

B. Conclusion

Both equality-based and combined model projection types can be used in our solution. Both of these types have their specific uses, but we decided to go with a model that is based on a **combined model** projection type. There is also an option of implementing both types and letting the user decide which model he needs at the time, but in current implementation we support only combined model. The reasoning behind the decision to use combined model over equality-based model is that it gives the user more information when using our tool, since the equality-based model hides metadata that are not available in both basic models.

IV. Designing Unified Representation

When designing a unified representation, we faced several challenges. In this section we discuss some of the most important of those questions and problems.

A. Ways to Access Metadata

One of the most important questions is how to access metadata in basic models in order to use them in a unified model. There are two possibilities how to achieve this:

1) **Direct access** to metadata with the use of an API that will unify the access methods in both basic models, and
2) **Model creation** and accessing metadata from the created unified model.

In the following sections, both of these ways are described in detail.

1) Direct Access: The most important thing when using **direct access** approach of accessing metadata is the interface that will be able to unify access to both basic models. In object oriented programming, this interface can be implemented using a adapter design pattern [15] [16].

In terms of accessing metadata, this design pattern can be expressed in such a way that class that needs metadata can access them by requesting them from the adapter, which handles acquiring metadata from basic model and transforms them into the unified model. Simplified example of this can be seen in Fig. 5, where the class `Client` calls a method `getMetadata` on a class `Adapter`. This method collects the requested metadata from one of the basic models which is available at the time, transforms them into the unified model that is expected and returns them to `Client`. The `Model` in this figure can represent either annotation processing model or reflection model. Which of these is currently represented depends on what part of the program life cycle it is currently in. Adapter therefore must be able to tell whether it is called during compile time or during runtime and will then choose how to retrieve metadata accordingly.

Using this method means that the ***basic model is accessed during every single call from the client***, because it has to retrieve the required metadata.

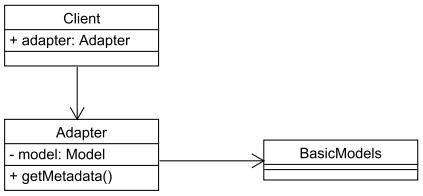

Fig. 5. Adapter Pattern

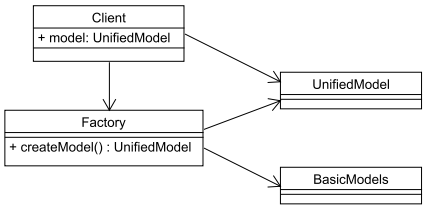

Fig. 6. Model Factory

2) Model Creation: The most important step when using **model creation** approach is the creation of the unified model itself. Unified model can be provided by a method similar to a factory method pattern [15].

In terms of model creation, this method will not be exactly the same as the factory method, because we know which type of class this method will return, but we do not know (and do not care) which basic model was used to create the class. Simplified example can be seen in Fig. 6, where once again the `Client` class needs access to metadata. Unlike the adapter patter, where the `Client` called a method which provided metadata, it now calls a method `createModel` on the `Factory` class. This method creates the unified model from one of the basic models and returns it to the `Client`. Same as with direct access, model that is currently represented by the `Model` class depends on what part of the program life cycle it is currently in. Therefore, just as Adapter had to be able to tell whether it was called during compile time or runtime, the `Factory` has to also be able to tell too. This method can also be compared to reverse engineering [17].

Unlike the direct access method, ***it is not needed to access basic model during every single call from the client***. Basic model needs to be only accessed once, when creating the unified model. Accessing metadata after the creation of unified model is handled by the unified model itself.

B. Additional Costs of Accessing Metadata

In basic models, which are already part of Java, costs to access specific metadata can be easily analysed. During each request to access metadata it is required to search the basic model and find the metadata that were requested. After they are found, they simply are returned in the same format as the basic model defines. If we disregard implementation details of basic models, this process is the same for reflection model as well as annotation processing model.

1) Direct Access: Additional costs when using direct access method is created mainly because during every request to access metadata from basic model, it is needed to search this model, find metadata that are needed and then transform them into the specified unified metadata representation that is expected as an output of the request. In comparison to acquiring metadata from basic model, it has one additional step, which is transforming metadata to the unified model. However, this one additional step is executed during every request, and it increases the time needed to retrieve metadata.

2) Model Creation: Additional cost when using model creation method is the creation of the unified model itself, but this process is only executed once. After the unified model is created, all requests are handled through the unified model itself and results do not have to be transformed into the representation that is expected as an output, because they already are in form of the unified model representation.

Comparison of retrieving metadata from unified model and from basic models depends on the exact implementation of the unified model. In ideal circumstances, the access to metadata through unified model can be faster than accessing them through basic models (in case the unified model is better suited for specific requests). It is more likely, however, that accessing metadata through unified model will be slower than through basic models.

C. Minimizing Additional Costs

In spite of the fact that reducing additional time costs to access metadata when using unified model is nearly impossible and it is not the purpose of this paper, we still have the possibility to employ some methods in order to minimize the performance impact of using unified model.

To help minimize additional costs we can use caching of results in both direct access and model creation methods.

Main idea of this in direct access method is that the results will be stored for use in another request during every request to access metadata. In every request after that, those metadata will already be at hand and it will not be needed to search the basic model for them again. Building on that method, we can make it so that it is very close to model creation method. That can be achieved by storing the cached data in such a way, that they would resemble the unified model created by model creation method. Downside of this is that we introduce a new step into the process of retrieving metadata: a step to check whether the requested metadata are already stored or not.

In model creation method, data would not be stored during requests, because even before the first request, the unified model was already built and put into memory. In this case, data would be prepared for most common requests during the model creation itself.

Because of the additional steps, caching of results runs into problems with small amount of requests and is getting better and better with more requests.

Different approach to minimizing additional costs is **selective model creation**. This method is based on the idea that user knows which metadata he or she will use and he or she can specify which parts of the model are required. User would be able to define which parts of the model will be created and which parts will be ignored. The disadvantage is that the user has to learn how to configure the tool in such a way, that it will provide desired results. If the configuration is too complicated then this disadvantage can be big enough to overshadow the biggest advantage of using a unified model, which is to make it easier for the user to gain access to metadata.

D. Query Methods

As it was mentioned before, both of the basic models have different way of accessing the same metadata. The metadata can be queried via methods that help the user easily navigate through the model. Both basic models and their APIs provide query methods. They are primarily used to conveniently query metadata. Both API query methods and model accessor methods (getters and setter) differ in Reflection API and Annotation Processor API. Example of this difference can be the way how to retrieve list of annotations that class `A` is marked with. To retrieve metadata about these annotations in annotation processing model, developer has to call a method `getAnnotationMirrors` on the instance of the class `Elements` that represents class `A`. On the other hand, in reflection model, developer has to call a method `getDeclaredAnnotations` on the instance of the class `Class` that also represents class `A`. If both of these methods were called in the same program on the same class, their results can be different (disregarding the fact that they are represented in a different way), because some of them can be marked with meta-annotation `@RetentionPolicy.SOURCE` or `@RetentionPolicy.CLASS`, but **they represent the same metadata**, which are annotations of class `A`.

The simplest way how to design query methods is to support both names of the same methods from basic models. For the aforementioned example it would mean that there would be two query methods, one called `getAnnotationMirrors` and second one called `getDeclaredAnnotations`. These methods would always return the same results.

Better way, which is in the spirit of unification, would be to use only one method that would not have to be named exactly the same as the ones in the basic models, but it would be named name clearly enough that there would be no confusion what the purpose of that method is.

There is also a way which mimics the principles of XPath [18] for XML files. It is based on the fact that the unified model is represented in a tree-like structure that can be traversed much like XML[1]. This way is best used as an addition to the method of using one unified method, because it would provide the developer with better control over the search queries.

E. Conclusion

For accessing metadata from basic models we decided to use the **model creation** approach. Main reason for this is that

[1]XML grammar [19] is commonly defined using XML schema [20].

in the direct access approach, searching through basic model during every request could be slow. Using the model creation approach pushes the additional time costs of using the unified API to a single point of the tool initialization.

On the other hand, the method of model creation can be designed in such a way, that when it comes to retrieving metadata from unified model, most common requests could be optimized so that they will be as fast as possible.

V. Prototype Implementation

Based on previous analysis, in this section we present a prototype tool *UModel*[2] that can be used to create a unified model. The name *UModel* comes from the term Unified Model. Because this tool has to be able to create unified model from both basic models, it has to be able to recognize which of the models is available during that time. This is easily done, because both of the basic models have different context in which they are created and these context can be easily recognized from one another.

A. Unified Model Creation

The abstract concept of unifying tool was illustrated in Fig. 1. In that figure, the process of creating unified model was not specified. Fig. 7 illustrates phases that are needed to create a unified model.

Model creation during runtime as well as compile time is divided into four main phases:

1) **Initialization phase**,
2) **Model creation phase**,
3) **Reference resolving phase**, and
4) **Finalisation phase**.

Initialization phase represents the phase in which the tool itself is initialized. That includes the initialization of structures that will be used during model creation as well as initialization of structures needed for result caching. Besides initialization of these structures, this phase also creates the basic structure of the model itself, which will be filled in during the next phase.

In the **model creation phase**, the unified model itself is created. Both runtime and compile time models are created by advancing through the provided basic model from top to bottom. Since metamodel can be represented as a tree structure, the tool starts with the root (topmost) element and gradually advances through its descendants to the leaves (bottommost elements). For every node in the basic model that the algorithm passes through, one node is created in the unified model. If the algorithm finds a reference to another element (e.g., method return type is a class included in the model), then this reference is only saved as a plain text. These references are resolved after the second phase finishes creating the whole model.

The model creation phase also saves the results for the queries encapsulated by the query methods presented in section V-D. These common queries, such as finding all the program elements annotated with a given annotation type, are this way cached and therefore should result in better time efficiency.

[2]https://github.com/mallynth/UModel

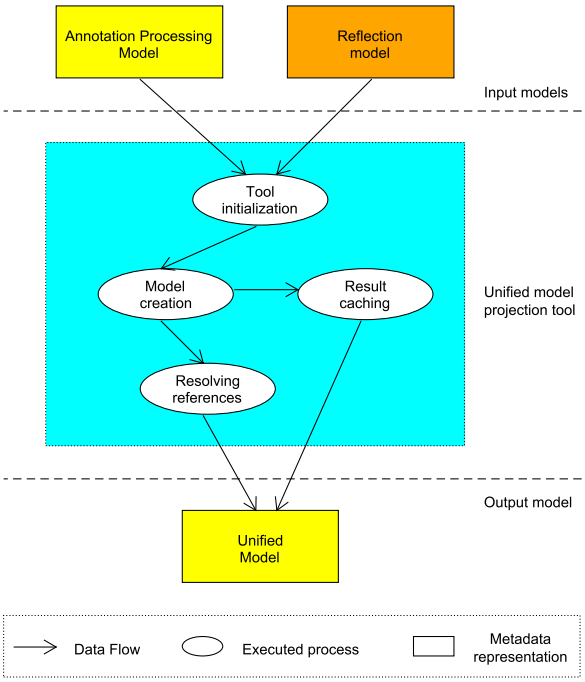

Fig. 7. Tool structure

After the second phase finishes, the **reference resolving phase** begins. During this phase, the tool resolves all of the references between the types that can be resolved (e.g., one class having a field of the type defined by another class). Reference that can be resolved is a reference to an element that is a part of the unified model. This kind of references can be used to easily traverse through the unified model. References that refer an element that is not a part of the unified model are not resolved and are left as a plain text. During this phase, the tool does not use the basic model, it works on the unified model created by phase two.

During the **finalisation phase**, the cache and the unified model (with resolved references from phase three) are joined together. After they are joined, the unified model is complete and can be passed to the tool's client.

B. Unified Model API

This section contains description of the set of most fundamental classes representing the unified model API. All classes start with the letter U which represents their unification from both models.

The `UModelFactory` class is a factory class in a singleton design pattern which is used to instantiate unified model. It only has one factory method, the `createModel` method that returns an object of the `UModel` class.

1) UModel: This class represents the unified model that contains specific packages and classes for a given project (the 'sampleApp' node from Fig. 2). Main purpose of this class is to wrap the components of the unified model into a single root node object and to provide query methods that provide API with common queries. These query methods will be described in section V-D.

2) UPackage: The `UPackage` class represents a package in the source code.

Selected fields:

- `name` - full name of the package.
- `containingClasses` - set of classes implemented in this package.
- `containingEnums` - set of enumeration types implemented in this package.

Packages in the unified model *are not represented with a tree structure*. Instead of using tree structure that organizes packages into a tree with one root (as the file system does), we use a simple structure where every package is represented on its own (the standard Java model).

3) UClass: The `UClass` class represents a class, interface or declaration of new annotation type.

Selected fields:

- `classType` - enumeration type that defines which language element is represented by this class. Possible values are `CLASS` for classes, `INTERFACE` for interfaces and `ANNOTATION_TYPE` for declaration of new annotation type.
- `name` - name of the element.
- `enclosingPackage` - reference to an enclosing package.
- `modifiers` - modifiers represented by Integer.
- `parent` - reference to a parent element.
- `interfaces` - set of interfaces that are implemented by this class. If the `UClass` object represents an interface, then it represents which interfaces are extended and if the `UClass` represents declaration of new annotation type then this set will be empty.
- `annotations` - set of annotations represented by the `UClassAnnotation` class.
- `enums` - set of declared enumeration types.
- `constructors` - set of constructors represented by the `UConstructor` class.
- `fields` - set of fields represented by the `UField` class.
- `methods` - set of methods represented by the `UMethod` class. If `UClass` represents a declaration of new annotation type then this set contains parameters of declared annotation.

4) UField: The `UField` class represents a field (variable, attribute) of the class.

Selected fields:

- `name` - field name.
- `enclosingClass` - reference to a class where this field is declared in.
- `modifiers` - modifiers encoded to `Integer`.
- `annotation` - set of declared annotations represented by the `UFieldAnnotation` class.
- `type` - type of the field. It can be represented either by a string or by a reference.

5) UMethod: The `UMethod` class represents one method of a class. If it is included in annotation type declaration then it represents a parameter of the new annotation.

Fields:

- `name` - method name.
- `enclosingClass` - reference to a class where this method is declared in.
- `modifiers` - modifiers represented by Integer.
- `annotation` - set of declared annotations represented by the `UMethodAnnotation` class.
- `parameterTypes` - ordered list of arguments. They are represented either by a string or a reference.
- `returnType` - type of the return value. It is represented either by a string or a reference.

The `UConstructor` class is very similar to the `UMethod` class. The only difference is that it does not include a name.

6) UAnnotation: The `UAnnotation` class is an abstract class that represents an annotation of any language element. Classes that extend the `UAnnotation` are:

- `UClassAnnotation` which represents a class annotation,
- `UConstructorAnnotation` which represents a constructor annotation,
- `UFieldAnnotation` which represents a field annotation,
- `UMethodAnnotation` which represents a method annotation.

Selected fields:

- `annotationClass` - type of the annotation. It is represented by a full name of the annotation type. If this annotation type is represented in the unified model, then it is also represented by a reference to that annotation type.
- `annotatedElement` - represents an element which is annotated by this annotation. Type of this element depends on the element that is annotated. For example, in `UClassAnnotation` this element has a type of `UClass`.
- `parameters` - set of parameters of this annotation. Each parameter is represented by the `AnnotationParameter` class which includes three `String` fields: its `name`, `type` and `value`.

C. Representation of specific elements

The fact that unified model is created from two existing model means that the representation of some language elements can be problematic, especially when it comes to elements that are represented differently in both basic models. One of them is the problem of representing modifiers, which were explained in the introduction of this paper. Other problem we had to face was how to represent references.

1) Representation of References: References, that are created during the second phase of unified model creation, are represented by the `AbstractMap.SimpleEntry` class. This class represents a key-value pair, which can for example be used in hash tables such as `HashMap`.

Key of this key-value pair is a full name of the element that is referenced. Key of every reference is filled in during the model creation phase. If possible, the value of this key-value pair is filled in during the reference resolving phase. The value is filled in only when the reference references an element included in the unified model. If that is the case, then the value of the key-value pair will be a reference to the element itself. If the reference references element that is not included in the unified model, then the value will be left as `null`. That means that every reference has a key, which represents the full name of the referenced class, but not every reference has a valid value.

2) Representation of Modifiers: As we mentioned before, one of the differences between annotation processing model and reflection model is the representation of modifiers. The difference is that modifiers in annotation processing are represented as a set of enumeration types `Set<Modifier>` and in reflection, they are represented as one integer.

In the unified model, we decided to use the representation identical to the one in reflection, which means all modifiers in unified model are represented by bits of an integer. When the unified model is created from annotation processing model, then the set of modifiers is converted into one integer by using bitwise OR.

D. Query Methods

These methods provide API for convenient metadata queries. Most of these methods are related to annotations, since annotations are the main point of this model.

Query methods included in the class `UModel`:

- `getClasses` - provides all classes that are included in the unified model. Returns a set of `UClass` objects.
- `findClassByFullName` - provides a class specified by its name. Return type is a `UClass` class.
- `findClass` - provides a class specified by a class `Class`. Return type is a `UClass` class.
- `getPackages` - provides all packages included in the unified model. Returns a set of `UPackage` objects.
- `findPackageByName` - provides a package specified by its name. Return type is the `UPackage` class.
- `getEnums` - provides all enumeration types that are declared in the unified model. Return type is a set of `UEnum` classes.
- `getAnnotatedElements` - provides all elements that are annotated by at least one annotation. Return type is a set of classes that implement the interface `AnnotableElement`. This interface is implemented by all elements that can be annotated.
- `getElementsAnnotatedWith` - provides all elements annotated by a specific annotation. Return type is a set of classes that implement the interface `AnnotableElement`.
- `getMethodsAnnotatedWith` - provides all methods annotated by a specific annotation. Return type is a set of `UMetod` classes.
- `getFieldsAnnotatedWith` - provides all fields annotated by a specific annotation. Return type is a set of `UField` classes.
- `getConstructorsAnnotatedWith` - provides all constructors annotated by a specific annotation. Return type is a set of `UConstructor` classes.
- `getClassesAnnotatedWith` - provides all classes annotated by a specific annotation. Return type is a set of `UClass` classes.

Last five methods, which provide elements annotated by a specific annotation, have two alternatives. In one of them, the annotation is specified by a class that extends the `UAnnotation` class. In the other, the annotation is specified by the `Class` object representing the given annotation type. This is a convenience alternative that allows using `Class` objects in order to enjoy type checking.

Beside query methods that are included in the `UModel` class, there are query methods that check whether an element is annotated by an annotation of a specific type. This annotation can either be specified by a class that extends the `UAnnotation` class or by the `Class` class of that annotation.

VI. Experimental Evaluation of Prototype

Although this paper does not deal with creating a faster way to access metadata when compared to basic models, it is still important that the time needed to access metadata by using unified model is not too demanding in comparison to basic models, otherwise the unified model would be unusable.

For the purposes of determining whether the created experimental tool is fast enough, we have conducted two experiments which are explained in this section.

A. Time Needed for Unified Model Creation

In the first experiment, we tested how much time is needed to create the unified model when using the created tool. This metric is very important, since this time can be considered as **additional time** when compared to basic approach.

For the purposes of this experiment we chose two programs, one of them consisted of 122 classes and the other consisted of 1745 classes. Both of these programs have on average approximately 100 lines of code per one class.

We only tested how much time it would take to create the unified model. That means the difference between time of initialization of the tool and the time that unified model was returned.

1) Results: Times that are presented in the table I are average times after 100 runs. It is important to note there were quite big deviations, especially during the compile time.

From the times presented in the table, we can see that the time it takes to create unified model from a small program

TABLE I
TIME NEEDED TO CREATE UNIFIED MODEL

	122 classes	1745 classes
Compile time	42 ms	670 ms
Runtime	387 ms	2043 ms

is not that high, especially during compile time. Time needed to create unified model from a bigger project was naturally higher, because more metadata needed to be processed.

One important thing to note from this experiment is how the tool is scaling when it comes to bigger programs. Second program is approximately 14 times bigger then the first one which means that if the scaling of the tool was linear, it would take 14 times more time to create unified model from bigger program than from smaller program. Creation of unified model from bigger program in compile time took little over 15 times more time than creation of unified model from smaller program. Similarly, creation of unified model from bigger program in runtime took approximately 5 times more time than creation of unified model of smaller program. From these data we can see that scaling of the tool during runtime is much better than scaling during compile time. The fact that the scaling during compile time is worse than linear suggests it may be a good idea to try to optimize implemented tool for the use in compile time.

B. Overall Speed of Prototype

The second experiment was designed to test the speed of the tool when performing a task. For purposes of this experiment, we used the same two programs as we used for the first experiment.

During this experiment we measured four ways in which a given task could be implemented:

- **Annotation processor model** - using metamodel available during compile time.
- **Reflection model** - using metamodel available during runtime. Helper libraries Scannotation [6] and Google Reflection API [21] were used for this measurement.
- **Unified model during compile time** - using the tool to create unified model during compile time and using this unified model to complete the task.
- **Unified model during runtime** - using the tool to create unified model during runtime and using this unified model to complete the task.

The task consisted of two parts (to make the test a little less trivial):

1) Find the names and types of all elements that are marked with the test annotation.
2) Retrieve all fields of a class that is specified by its full name.

For the first part of this experiment a simple test annotation was created and then added to source codes of the programs. In the program that consisted of 122 classes, 8 elements were marked with the test annotation and in the program that consisted of 1745 classes, 137 elements were marked with the test annotation.

TABLE II
OVERALL SPEED OF TASK COMPLETION

	122 classes	1745 classes
Annotation processing model	12 ms	484 ms
Reflection model	1154 ms	2822 ms
Unified model during compile time	47 ms	949 ms
Unified model during runtime	472 ms	2245 ms

1) Results: Results of the experimented are presented in table II.

Times that are presented in this table are the average times after 100 runs. Same as in the first experiment, some of the times that were measured when using the unified model were often very different when compared to the average, especially during the tests with the bigger program. During compile time, the lowest measured value was 477 ms and the highest measured value was 1189 ms. During runtime, the lowest measured value was 1823 ms and the highest measured value was 2480 ms. These fluctuations could have been caused by the fact that we used unordered sets in the unified model and the position of requested metadata is always different.

From the data we can see that the time needed to complete the task in unified model during compile time was more then double when compared to the time needed to complete the same task during compile time without the use of the unified model. This fact reinforces the idea of trying to optimize unified model creation during compile time.

We can also see that during runtime, the tool was actually faster in completing the given task. Main reason for this is that helper libraries Scannotation and Google Reflection API were used to make the task significantly easier in reflection model. These libraries made the task easier at the cost of additional time during execution, which is also the main reason behind the unified model.

VII. RELATED WORK

As it was mentioned before, main inspiration of our solution to the problem we faced with the *BTE* [4] tool was the idea of Generalized and Attributed Abstract Syntax Trees (GAAST) proposed by Cepa [7], [8], [22]. In his work [7] he states that object oriented languages should support an explicit representation of the program as a graph of meta-objects that can be then accessed through a well defined API. He also shows, that usage of meta-model in a GAAST structure helps support model-driven development.

Noguera has a similar approach to Cepa in his tool AVal [23], [24], which is used to validate frameworks that are using annotations. Purpose of this validation is checking, whether the framework uses specific annotations in a correct way. Specification of how annotation is used correctly is designed through meta-annotations, which define constraints on the usage of those annotations. He uses two models for this validation. One of them is *annotation model*, which is a model

derived from annotation types and second one is *code model*. Code model is very similar to an abstract syntax tree.

We used a similar approach to the unification of two models which was discussed in this paper in the tool Bridge To Equalia [4]. This tool is able to create a unified model of annotation-based and XML-based configurations using by extensible metamodel.

There are libraries that help querying metadata during runtime, such as Scannotation [6] and Google Reflections [21]. These libraries provide convenient APIs for annotations processing, however, they do not address the problem of runtime and compile time model unification.

VIII. Conclusion

In this paper we explored differences between compile time and runtime models, explained the problem of their diversity and analyzed the approaches how to unify these models by providing their advantages and disadvantages. The discussion should provide a basis for consideration of providing a standard unified API for accessing metadata in programming languages. We presented the API we designed and implemented in our tool for unified model creation, which was used to support the analysis. The tool was evaluated by experiments to verify the usability of the proposed tool. They and showed that both the creation of unified model and its querying are reasonably fast. The paper can be considered a technical report that should be helpful for developers dealing with the same problem, or developers that are interested in either of the two existing approaches to annotation processing (reflection and annotation processing API).

References

[1] S. Chodarev, D. Lakatoš, J. Porubän, and J. Kollár, "Abstract syntax driven approach for language composition," *Central European Journal of Computer Science*, vol. 4, no. 3, pp. 107–117, 2014. http://dx.doi.org/10.2478/s13537-014-0211-8

[2] D. Lakatoš, J. Porubän, and M. Bačíková, "Declarative specification of references in DSLs," in *2013 Federated Conference on Computer Science and Information Systems*, ser. FedCSIS 2013, Sept 2013, pp. 1527–1534.

[3] Z. Havlice, "Auto-Reflexive Software Architecture with Layer of Knowledge Based on UML Models," *International Review on Computers & Software*, vol. 8, no. 8, 2013.

[4] M. Nosáľ and J. Porubän, "Supporting multiple configuration sources using abstraction," *Central European Journal of Computer Science*, vol. 2, no. 3, pp. 283–299, Oct. 2012. http://dx.doi.org/10.2478/s13537-012-0015-7

[5] M. Nosáľ and J. Porubän, "XML to Annotations Mapping Definition with Patterns," *Computer Science and Information Systems*, vol. 11, no. 4, pp. 1455–1477, 2014. http://dx.doi.org/10.2298/CSIS130920049N

[6] Scannotation, "Scannotation project homepage," 2015. [Online]. Available: http://scannotation.sourceforge.net/

[7] V. Cepa and M. Mezini, "Language support for model-driven software development," *Science of Computer Programming*, vol. 73, no. 1, pp. 13–25, 2008, special Issue on Foundations and Applications of Model Driven Architecture (MDA). http://dx.doi.org/10.1016/j.scico.2008.05.003

[8] V. Cepa, *Attribute enabled software development: illustrated with mobile software applications*. Saarbrücken, Germany: VDM Verlag, 2007.

[9] V. Cepa and M. Mezini, "Mobcon: A generative middleware framework for java mobile applications," in *Proceedings of the 38th Annual Hawaii International Conference on System Sciences, 2005. HICSS '05.*, Jan 2005, pp. 283b–283b. http://dx.doi.org/10.1109/HICSS.2005.431

[10] H. Rocha and M. T. Valente, "How Annotations are Used in Java: An Empirical Study," in *SEKE*, 2011, pp. 426–431.

[11] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies," in *Proceedings of the 2010 Asia Pacific Software Engineering Conference*, ser. APSEC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 336–345. http://dx.doi.org/10.1109/APSEC.2010.46

[12] Oracle, "javax.lang.model.element documentation," 2015. [Online]. Available: http://docs.oracle.com/javase/7/docs/api/javax/lang/model/element/package-summary.html

[13] ——, "java.lang.reflect documentation," 2015. [Online]. Available: http://docs.oracle.com/javase/7/docs/api/java/lang/reflect/package-summary.html

[14] S. Zawoad, M. Mernik, and R. Hasan, "FAL: A forensics aware language for secure logging," in *2013 Federated Conference on Computer Science and Information Systems*, ser. FedCSIS 2013, Sept 2013, pp. 1579–1586.

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

[16] B. Benatallah, M. Dumas, M.-C. Fauvet, F. A. Rabhi, and Q. Z. Sheng, "Overview of Some Patterns for Architecting and Managing Composite Web Services," *SIGecom Exch.*, vol. 3, no. 3, pp. 9–16, Jun. 2002. http://dx.doi.org/10.1145/844339.844346

[17] E. J. Chikofsky and J. H. Cross II, "Reverse Engineering and Design Recovery: A Taxonomy," *IEEE Software*, vol. 7, no. 1, pp. 13–17, Jan. 1990. http://dx.doi.org/10.1109/52.43044

[18] T. Grigalis and A. Čenys, "Using XPaths of inbound links to cluster template-generated web pages," *Computer Science and Information Systems*, vol. 11, no. 1, pp. 111–131, 2014. http://dx.doi.org/10.2298/CSIS130416020G

[19] J. Kollár, I. Halupka, S. Chodarev, and E. Pietriková, "pLERO: Language for grammar refactoring patterns," in *2013 Federated Conference on Computer Science and Information Systems*, ser. FedCSIS 2013, Sept 2013, pp. 1503–1510.

[20] M. Pušnik, M. Heričko, Z. Budimac, and B. Šumak, "XML Schema metrics for quality evaluation," *Computer Science and Information Systems*, vol. 11, no. 4, pp. 1271–1289, 2014. http://dx.doi.org/10.2298/CSIS140815077P

[21] Google, "Google reflection api project homepage," 2015. [Online]. Available: https://github.com/ronmamo/reflections

[22] V. Cepa and M. Mezini, "Declaring and Enforcing Dependencies Between .NET Custom Attributes," in *Generative Programming and Component Engineering*, ser. Lecture Notes in Computer Science, G. Karsai and E. Visser, Eds. Springer Berlin Heidelberg, 2004, vol. 3286, pp. 283–297. http://dx.doi.org/10.1007/978-3-540-30175-2_15

[23] C. Noguera and L. Duchien, "Annotation Framework Validation Using Domain Models," in *Proceedings of the 4th European Conference on Model Driven Architecture: Foundations and Applications*, ser. ECMDA-FA '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 48–62. http://dx.doi.org/10.1007/978-3-540-69100-6_4

[24] C. Noguera and R. Pawlak, "AVal: an extensible attribute-oriented programming validator for Java: Research Articles," *Journal of Software Maintenance and Evolution*, vol. 19, no. 4, pp. 253–275, Jul. 2007. http://dx.doi.org/10.1002/smr.349