DEFINING ANNOTATION CONSTRAINTS IN ATTRIBUTE ORIENTED PROGRAMMING

Štefan RUSKA, Jaroslav PORUBÄN
Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice,
Letná 9, 042 00 Košice, Slovak Republic, tel.: +421 55 602 2565,
e-mail: stefan.ruska@gmail.com,jaroslav.poruban@tuke.sk

ABSTRACT

Attribute oriented programming allows programmers to extend source code elements semantics by adding decorative comments called annotations. Annotations can be then processed by tools and frameworks during compile or run time. With wide usage of such tools and frameworks, annotation creators define many constraints on how these should be used. Constraints are usually expressed informally in source code comments or external documentation. Correct annotation usage is validated lately during deploy or run time. This paper presents a tool that checks annotation constraints at compile time using a Prolog like constraint language. The tool uses an embedded database to store program elements and a simple constraint to SQL compiler. Query results are then asserted against existing annotations to check their validity. The tool can be seamlessly integrated with existing IDEs.

Keywords: *annotations, annotation constraint, attribute oriented programming*

1. INTRODUCTION

Attribute oriented programming allows programmers to extend source code elements semantics by adding custom decorative comments called annotations [1]. With recent support of annotation types in popular programming languages like Java and C# many specialized tools and frameworks have adopted them and are using them during deploy or run time. Annotations allow programmers to express language element semantics explicitly in a declarative fashion. Annotations have gained much popularity in field of source code validation, unit testing, software artifacts generation or object-relational mapping.

1.1. Motivation

Let us first consider following example of a class annotated with Java Persistence API annotations [12].

```
@Entity
@Table(name="vehicle")
public class Car implements Serializable {
   @Id
   @Column(name="carID",nullable=false)
   Long id;
}
```

With @Entity annotation class Car is marked as persistable entity. @Table annotation specifies the Car class will be mapped to a table named vehicle. @Id annotation tells the framework that the id field is going to be tables primary key. @Tables column annotation value signalizes that the ID field is going to be mapped to a column called carID.

As far as above annotations are modeling relational database structures it is obvious they cannot be placed freely. @Column annotation represents a database column which must be present in some table, therefore field's class must be annotated with @Table annotation. Unless we define a compound primary key there must be at most one field having annotation @Id. What is even more, table primary keys cannot have null values what could be violated by setting @Columns nullable parameter to true value.

This is only one of many examples where annotations do require several restrictions to be met to be used correctly. If we take a look at the Java's standard @Target metaanotation, that restricts annotation only to be used on particular element, we see this kind of apparatus is insufficient for defining even simple constraints. Annotations are far away from being standalone entities and there is a real need for a flexible formal apparatus for specifying annotation restrictions. One would argue that these constraints are after all validated by the framework itself but this kind of validation takes place during deploy or even run time. This means incorrect annotation usage is propagated to later software development phases.

Annotation constraints can and should be validated during compile time. These constraints are well known during creation of new annotation types and their authors are able to describe them semi-formally in source code comments or external documentation. Constraints however can be expressed formally and later checked by some tool. Such a generic constraint checking tool would be beneficial for:

- framework creators as they would not need to repeatedly hardcode annotation validation functionality,
- annotation users as the tool would check and recommend correct annotation usage.

This paper presents a flexible, modular and scalable tool for checking correct annotation usage at compile time. Constraints are expressed using a custom domain specific language with Prolog like syntax. The tool uses a simple compiler that transforms custom rules into SQL queries which results are then asserted against existing source code. The tool can be seamlessly integrated with current IDEs.

2. ANNOTATION CONSTRAINT PATTERNS

We empirically noticed several recurring constraint patterns among common annotations. These were also discussed in [3] and can be categorized as follows:

- *Parent-child relation* defines a constraint where annotation can be placed on language element only in a scope of language element annotated with another particular annotation. This constraint is common when annotations represent hierarchical structures.

- *Mutual exclusivity* specifies that two different annotations cannot be present on the same element simultaneously. This constraint is common on annotations having opposite semantics such as @Null and @NotNull or @Stateless and @Stateful annotations. A special kind of mutual exclusivity is JAXB's [13] @Transient annotation that cannot be simultaneously present on one element with any other annotation from package javax.xml.bind.

- *Unique annotation occurrence* represents annotations that can be present at most once in a scope of some language element. This pattern is typical for annotations defining unique attributes or for annotations where presence of multiple annotations would cause ambiguity.

- *Occurrence of multiple annotations* requires multiple annotations are present on one element or in a scope of another element. This type of pattern is rare.

- *Annotation values referencing other elements.* These constraints are especially vulnerable to misuse as references are usually expressed using simple string literals. Therefore if the referenced element's signature changes developer must change all referencing string literals manually (current IDEs do not mange such functionality automatically) what might be time consuming and error prone operation.

Once we can specify annotation constraints we should also think of where an annotation constraint should be placed. For parent-child relation it is sufficient to define constraint on child annotation. Mutual exclusivity is reflexive so it is sufficient to define constraint only on one of the two annotations. Occurrence of multiple annotations constraint might not be symmetric so one need to think about the suitable constraint placement.

All previously observed patterns define structural properties on source code and there are multiple source code structure inspection tools already implemented.

2.1. Source code querying approaches

Source code structure querying tools are usually used for supporting source code comprehension, source code knowledge mining or code metrics measurements. These tools are also well suited for checking source code compliance with company's or general coding standards.

We have recognized three different approaches for source code structure querying.

- Source codes are transformed into XML documents and a XML query mechanism (XPath, XQuery) is used to retrieve structural relations. This approach has been implemented in [4, 5], and it provides great means for representing hierarchical structures. However it is quite memory hungry and repetitive XML parsing is required for every run.

- Source code is persisted into logic program fact base and logic programming language like Prolog is used to retrieve source code structural properties. This has been implemented in [6]. This approach does not scale well as all the facts need to be present in main memory during evaluation.

- Source code is persisted in a relational database and SQL is used to query it [7]. This is memory friendly solution but SQL is very verbose and one needs to be aware of underlying database schema. When defining complex rules SQL queries get unreadable and hard to maintain.

For us the most feasible and interesting approach combines the latter two of mentioned approaches. Similar approach was implemented in [8]. Databases do scale well so the tool should work well even with large projects and prolog syntax is simple but expressive enough to be able to cover majority of known constraints. This approach means a dedicated prolog to SQL compiler needs to be created. We also need to create our own prolog like language.

3. DESIGN OF THE TOOL

3.1. Basic concept

As it was suggested in previous section tool is going to combine a relational database with prolog like constraint language. The tool does following:

1. it persists source code elements into relational database,
2. obtains annotation constraints,
3. compiles rules into SQL queries,
4. runs SQL queries,
5. based on the queries results it checks annotations validity.

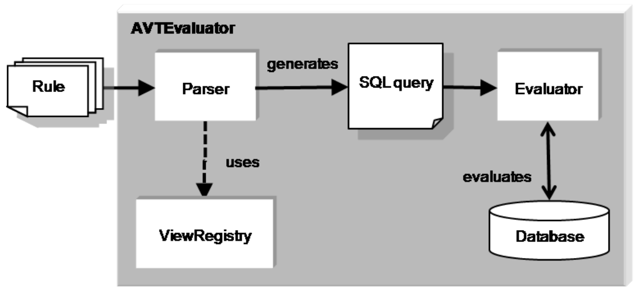

Fig. 1 Rule evaluation process

The evaluation process is showed in Fig. 1. Since relational databases represent rather flat data structures, to map a hierarchical structures like type hierarchy we are using a simple ad-hoc linearization technique where every parent-child relationship is presented as a separate record. Some database systems allow to specify recursive queries using a dedicated language structures but these are not standardized and are therefore not portable.

The tool can be run in two modes. A full mode creates database schema and persists all elements reachable from current project's elements. An incremental mode updates only elements that changed since last compilation and is recommended for a day-to-day use. If the underlying database system is chosen wisely developers would not even recognize there is some validation being done in the background.

Tool uses an expansive algorithm for visiting all reachable elements within source code and is implemented in a standard Visitor pattern fashion. In short this means that e.g. from a sample method element, the algorithm visits methods return type, parameter types, thrown types and annotation types. Since one element can be reached from multiple places, the tool keeps track of already visited elements and assures every relevant element is visited at most once.

3.2. Views

Database views present a virtual tables that are not physically keeping records and they are only declaring a fine grained perspective over underlying data. This allows views to provide higher level of abstraction. Annotation validation tool relies on database views as every rule is mapped to a corresponding database view.

Database views can be effectively used for predicate evaluation. Every view represents a predicate as it defines some condition its records must meet. Let us suppose we have three views on a person table. These are male (selects persons that are male sex), young (selects persons that are under 20) and student (selects persons that are students). If we now want to check if John Smith is a male young student we can craft a query as follows *SELECT 1 FROM male v1,young v2,student v3 WHERE v1.name=v2.name AND v2.name=v3.name AND v1.name='John Smith'*. If the query returns zero rows we can say claim about "John Smith" was false.

The same principle is being used while evaluating if annotation presence on a particular element is valid. We have multiple rules, and therefore multiple views, that declare several predicates about source code elements. Once these are translated into SQL query it is extended with equality condition on element's unique identifier.

As there are unique indexes over the id fields in the database schema such queries use fast unique scans and are evaluated immediately. If the query returns no rows the annotation is not being used correctly. Such approach results in a very short tool run times.

3.3. Constraint language

The tool uses a lightweight version of Prolog like language that does not support recursion and lists. This custom domain specific language was created using YAJCo parser generator [2]. YAJCo uses annotated classes and interfaces to express abstract syntax of the language. Annotations are used for defining concrete syntax of each language concept. A parser is created directly from the model and during parsing it recreates the model back into memory. With this parser generator, language creator does not need to be familiar with compiler creation concepts despite still being very productive.

The language defines two basic types of rules. One is a compound rule known from prolog having form *ruleName(arg1,,argN) -> expression* . An expression can be a conjunct, disjunct or negation. Another rule is also considered as an expression so it can be present on the right side of the compound rule. Second type of rule is target rule in form *bindVar <- expression* that specifies which element an annotation can be placed on. The constraint language defines also different kinds of arguments like integer argument, string argument, binded argument (in form *$<name>*) or free argument (in form _).

Constraint language sentences are compiled into SQL queries using following rules (mentioned also in [9, 10]):

1. Rule is transformed into SELECT clause on particular view.
2. Logical AND operation between rules generates a table join (tool automatically manages views aliases).
3. Binded variables with same name generate equi-join conditions.
4. Negation is transformed into NOT EXISTS clause.
5. Logical OR operation generates UNION clause.
6. Constants are transformed into corresponding SQL literals.

The tool automatically generates an "any" clause (a relation over all elements) when there is only a negation on the right side of the compound rule. This makes sure compiler generates a valid SQL query as the query must contain at least one select clause.

3.4. Defining rules

Annotation constraints can be placed using one of the following forms:

- Constraint is defined in a @TargetRule metaannotation directly placed on required annotation type. This form is recommended for our custom annotation types where constraints have been well tested.
- When using @AppliesTo metaannotation one can define a rule for already existing annotation type he cannot infer. During evaluation tool applies rules from our custom annotation types to specified out-of-the-box annotation types.
- Specifying annotation constraint in an external property file. This is beneficial during debugging when rule can be changed without need for explicit compilation.

Table 1 Tool Runtimes

	Full mode/Project incremental/Class incremental in seconds		
Project Name	HSQL	H2	Firebird
JakartaRegexp	24/1.2/0.6	167/2/2	715/1.5/0.7
JEdit	40/2.7/0.9	387/7.4/2	2040/24/2.1
AspectJ	153/194/4	1760/174/3.2	5412/170/3.4

3.5. Extending rules

The tools provides multiple ways of extending annotation rules to define virtually any kind of constraint.

Defining a new compound rule. This is the preferred way of creating new rules providing the highest level of abstraction. One can use logical operations to compose existing rules into another rules that are once declared again available for further composition. The tool provides a simple polymorphism mechanism where multiple rules having the same name but different number of arguments can be used to specify the same concept at different level of detail.

Defining custom database view. If requested constraint cannot be expressed as a compound rule we can define our own database view in views XML file. These allow us to use the power of SQL language such as outer joins, aggregate functions or even stored procedures. Once a new view is specified it becomes automatically available as a new rule for further composition.

Defining custom element generator can be used if both of above approaches are insufficient. With custom element generator we use a Java language to infer element insertion into database. Full power of general programming language provides unlimited variation of rules. Current version of the tool contains a basic "element_flag" table where one can insert element ids with custom string flags. One can later define a custom rule over the existing element_flag rule to specify a predicate for the particular custom rule.

The tool manages compound rule and XML view files automatically so every change to these files is automatically reflected to underlying database system during next tool run.

4. EXAMPLES AND EXPERIMENTS

This section proves soundness and usability of the tool by providing multiple examples and time measurements. Let us first define constraint for parent-child relation of @Table and @Column annotation types. Before the final rule is defined we define several intermediate rules. Rules like type, method, package etc. are standard.

```
annotation($Id,$Name) ->
   type($Id,$Name,'ANNOTATION_TYPE',_,_,_);
parent($This,$Parent) ->
   field($This,_,_,_,_,$Parent)
   || method($This,_,_,_,_,_,_,$Parent)
   || type($This,_,_,_,_,$Parent)
   || package($This,_,_,$Parent)
   || param($This,_,_,$Parent);

hasAnnotation($Element,$AnnotationName) ->
   annotatedElement($Element,$Annotation,_,_)
   && annotation($Annotation,$AnnotationName);

parentHasAnnotation($Elem,$ParAnnotQualName)->
   parent($Elem,$Parent) &&
   hasAnnotation($Parent,$ParAnnotQualName);
```

For a unique annotation occurrence we can define following rule.

```
sibling($This,$Sibling) ->
   parent($This,$Parent)
   && parent($Sibling,$Parent)
   && $This != $Sibling;

hasUniqueAnnotation($Element,$AnnotName)
-> hasAnnotation($Element,$AnnotName)
   &&
   !(sibling($Element,$Sibling)
     &&
     hasAnnotation($Sibling,$AnnotName));
```

Annotation @PostConstruct must be according to documentation [11] used as follows. *"Annotation PostConstruct must be placed on a method that has no parameters except of parameter of type InvocationContext. The return type of the method must be void and method must not throw a checked exception. The method must not be static."* This relatively complex rule can be easily expressed as below

```
$PostConst <-
    methodSig($PostConst,
      ['* void *('+$Param+')'])
    && ($Param = ''
          || isOfQualType($Param,
             '*.InvocationContext'))
    && !static($PostConst)
```

To demonstrate the tool's usability in terms of performance the tool has been tested on three different sized open source projects namely JakartaRegexp, JEdit and AspectJ in three different modes:

- Full mode when all elements reachable from current project get persisted into database. This mode is very insert intensive.
- Incremental mode of whole project. This mode is much faster as full mode as it persists only elements that changes since last compilation.

- Incremental mode on a single class is the fastest mode. Only necessary changes of single class get persisted.

Results of these runs are presented in Table 1. From the results we can see that tool scales well even for larger projects and that single class incremental mode can be used on regular basis.

5. CONCLUSION

This paper presented a flexible and extensible tool for defining annotation constraints in a declarative way. These are automatically validated during compile time in order to restrict propagation of incorrect annotation usage to later project phases. Constraint language can be easily used to define even complex rules. Tool provides means for specifying virtually any kind of complex rule. It can be seamlessly integrated with IDEs like NetBeans and Eclipse and its execution is completely transparent for end-developer.

ACKNOWLEDGEMENT

This work is the result of the project implementation: Center of Information and Communication Technologies for Knowledge Systems (ITMS project code: 26220120020) supported by the Research & Development Operational Program funded by the ERDF.

REFERENCES

[1] WADA, H. – SUZUKI, J.: Modeling Turnpike Frontend System: a Model-Driven Development Framework Leveraging UML Metamodeling and Attribute-Oriented Programming, In Proc. of the 8th ACM/IEEE MoDELS, October 2005.

[2] PORUBÄN, J. – FORGÁČ, M. – SABO, M. – BĚHÁLEK, M.: Annotation Based Parser Generator, In Computer Science and Information Systems, Vol. 7, No. 2, 2010, pp. 291–307, ISSN 1820–0214.

[3] NOGUERA, C. – PAWLAK, R.: AVal: an Extensible Attribute-Oriented Programming Validator for Java, scam, pp.175-183, Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'06), 2006.

[4] EICHBERG, M. – SCHFER, T. – MEZINI, M.: Using Annotations to Check Structural Properties of Classes, In 8th International Conference on Fundamental Approaches to Software Engineering (FASE 2005), Vol. 3442, pp. 237-252, Springer, 2005.

[5] NODLER, J. M.: An XML-based Approach for Software Analysis, Masters Thesis, Gttingen: University of Gttingen, Institute for Computer Science, 2007.

[6] MARKLE, L.: JQuery - A Tool for Combining Query Results and a Framework for Building Code Perspectives, Master Thesis, Ontario: The University of Western Ontario, 2006.

[7] OSENKOV, K.: Static Analysis and Source Code Querying, on Internet: http://kirillosenkov.blogspot.com/2007/09/static-analysis-and-sourcecode.html

[8] HAJIYEV, E. – VERBAERE, M. – DE MOOR, O.: CodeQuest: Querying Source Code with Datalog, In Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (San Diego, CA, USA, October 16-20, 2005).

[9] DRAXLER, Ch.: A Powerful PROLOG to SQL Compiler, Technical report, CIS Centre for Information and Speech Processing, Ludwig-Maximilians-University, Munich, 1993.

[10] MOGENSEN, L.: Manipulation of Logic Programs and Translation to SQL, In Advanced Topics in Databases, April 2004.

[11] Sun Microsystems, Inc.: Java Platform - Annotation Type PostConstruct, 2008, on Internet: http://java.sun.com/javase/6/docs/api/javax/annotation/PostConstruct.html

[12] Java Persistence API as part of JSR-000220 Enterprise JavaBeans 3.0, on Internet: http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html

[13] Sun Microsystems, Inc., Java Architecture for XML Binding (JAXB), on Internet: http://java.sun.com/javase/6/docs/technotes/guides/xml/jaxb/index.html

Received August 12, 2010, accepted November 22, 2010

BIOGRAPHIES

Štefan Ruska received his MSc. in 2010 at the Department of Computers and Informatics, Technical University of Košice, Slovakia. He defended his master thesis in the field of attribute oriented programming. His interests include design and implementation of domain specific languages and database programming. He currently holds consultant position at Ariba, Inc.

Jaroslav Porubän is Associate professor at Department of Computers and Informatics, Technical university of Košice, Slovakia. He received his MSc. in Computer Science in 2000 and his PhD. in Computer Science in 2004. Since 2003 he is the member of the Department of Computers and Informatics at Technical University of Košice. He was involved in the research of profiling tools for process functional programming language. Currently the main subject of his research is the computer language engineering concentrating on design and implementation of domain-specific languages and computer language composition and evolution.