Electronic Communications of the EASST
Volume 15 (2008)

Proceedings of the 8th International Workshop on OCL Concepts and Tools (OCL 2008) at MoDELS 2008

Static Source Code Analysis using OCL

Mirko Seifert and Roland Samlaus

15 pages

Guest Editors: Jordi Cabot, Martin Gogolla, Pieter Van Gorp
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Static Source Code Analysis using OCL

Mirko Seifert[1] **and Roland Samlaus**[2]

[1] mirko.seifert@inf.tu-dresden.de,
[2] roland.samlaus@inf.tu-dresden.de,
Computer Science Department
Technische Universität Dresden, Germany

Abstract: The majority of artifacts created during software development are representations of programs in textual syntax. Although graphical descriptions are becoming more widespread, source code is still indispensable. To obtain programs that behave correctly and adhere to given coding conventions, source code must be analyzed — preferably using automated tools.

Building source code analyzers has a long tradition and various mature tools exist to check code written in conventional languages, such as Java or C. As new languages emerge (e.g., Domain Specific Languages) these tools can not be applied and building a tool for each language does not seem feasible either.

This paper investigates how meta models for textual languages and the Object Constraint Language can enable generic static source code analysis for arbitrary languages. The presented approach is evaluated using three languages (Java, SQL and a DSL for state machines).

Keywords: Static Analysis, Object Constraint Language, Domain Specific Languages

1 Introduction

Since the early days of software development, the main artifact within all processes was source code. Carefully crafted by developers the textual representation of programs was considered most important to build software systems. To ease the creation of programs, many programming languages were designed and evolved over the decades. Driven by the incredibly fast growing complexity of real world applications, language designers struggled to keep up with this progress. New language constructs or even paradigms were introduced to support higher levels of abstraction, easier readability, extensibility, and maintenance.

Despite the various approaches to reduce complexity, current applications can usually not be understood as a whole by a single human. Finding bugs or design flaws in huge amounts of program code by hand is not feasible and tools are needed to perform analysis and point out problems to the developers automatically. Some of these tools [Bur, HP04, Vol06, HP00] use very sophisticated techniques and do an excellent job analyzing programs written in a particular language. They can find potential runtime errors, dead code, duplicate code, violations of coding conventions or style guidelines, and bad design smells. Some tools even allow to extend the set of available analyses using a custom pattern definition language.

Parallel to the advancements in code analysis, the rapidly growing complexity forced practitioners and researchers to find novel approaches that would allow systems to be more easily created, understood, and maintained. Model Driven Development (MDD) and Domain Specific Languages (DSLs) are currently considered to be promising approaches to tackle the complexity problem. MDD uses models to introduce higher levels of abstraction and DSLs can be used by domain experts to design a system's structure and behavior in their preferred way.

Within this context, source code analysis gains new relevance. Foremost, textual DSLs can and must be subject to code analysis. Checking coding conventions and detecting potential errors can be as beneficial as in the context of traditional programming languages. Furthermore, as models and DSLs are usually transformed into some conventional programming language, source code analysis can be applied to find new disallowed patterns. For example checking that hand-written extensions to generated code do not violate the semantics of the underlying model is one such novel application scenario.

Altogether, it can be observed that future source code analysis must handle many new languages and unanticipated application scenarios. Tools that are generic (applicable to multiple languages) and extensible (usable in arbitrary contexts) will be needed to support modern development processes. This paper describes such an approach for generic static source code analysis based on the Object Constraint Language (OCL) [The06] and evaluates the options and limitations of the presented approach. A prototypical implementation based on the Eclipse Modeling Framework (EMF) and the Eclipse OCL implementation was used to gain practical experience and to find limitations of the theoretic concept.

The remainder of this paper is organized as follows: Section 2 introduces meta modeling of textual languages, which forms a basis for the presented work. Section 3 discusses how the analysis is carried out conceptually, followed by an explanation of our prototypical implementation in Sect. 4. The theoretical concepts are evaluated in Sect. 5. Related work is presented in Sect. 6, before we conclude and elaborate on future work in Sect. 7.

2 Meta Models for Textual Languages

The basic idea of the analysis method we propose, is to use the OCL to query source code. As OCL can only be applied to models, textual artifacts must be transformed in order to evaluate OCL expressions on them. More precisely, a model must be derived from the textual representation. Naturally, these models must conform to a meta model, which specifies the logical structure of the language to analyze. This section points out how we derive such meta models automatically and explains the procedure needed to instantiate models for concrete documents.

We choose to create one meta model for each language that is subject to an analysis. Alternatively one could use a single meta model for all languages and create bindings for the abstract syntax of concrete languages to this meta model. However, using a single meta model complicates query specification, because no custom language elements can be used. Furthermore, additional effort for the specification of the binding is needed. Therefore this option was discarded.

To create a meta model for a particular language we utilize Reuseware. Reuseware is a framework for Invasive Software Composition and contains a tool called EMFTextEdit [Reu], which

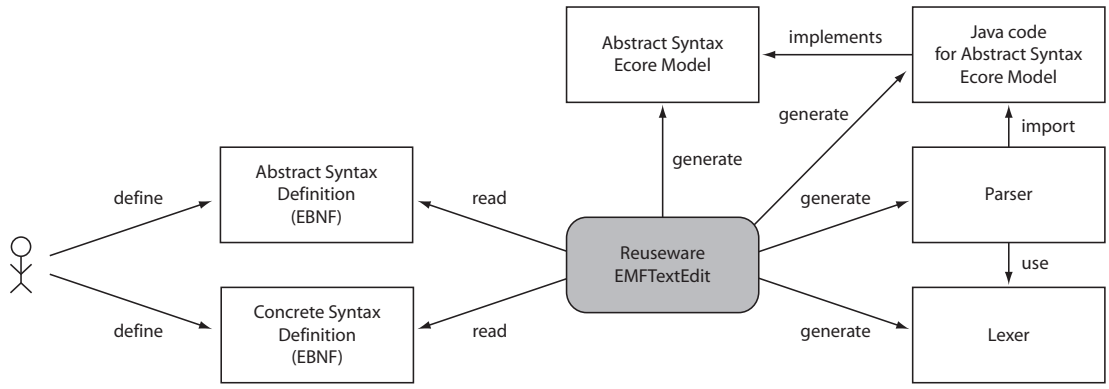

Figure 1: Deriving lexer, parser and meta model from syntax definitions

can translate syntax definitions of a language to a meta model. To perform this task for a particular language two input components are needed. First, an abstract syntax has to be defined that represents the structure of the language. Second, a concrete syntax definition is needed which defines the exact representation in source code. The relation between EMFTextEdit, both syntax definitions and the generated artifacts is shown in Fig. 1. The details of the generation will be explained shortly.

Both the concrete and the abstract syntax are defined using a notation similar to the Extended Backus-Naur Form (EBNF). This notation allows the user to define the structure of a programming language and the terminal symbols. An example for an abstract syntax definition can be found in Listing 1.

Listing 1: Abstract syntax for Java class definitions.

```
1 Class = modifier:Modifier, name:Identifier,
2         extends:QualifiedName?,
3         implements:QualifiedName*,
4         classBody:MemberDeclaration*;
```

While the abstract syntax solely specifies the logical structure of the language, the concrete syntax defines key words and symbols that divide the logical elements. For example, Listing 1 (line 2) states that a class definition can contain an optional extends declaration and Listing 2 (line 2) defines that the key word **`extends`** will be used to mark the beginning of such a declaration. The two syntaxes are connected by using the same identifiers (e.g., **`extends`**).

Listing 2: Concrete syntax for Java class definitions.

```
1 Class ::= modifier "class" name
2           ("extends" extends)?
3           ("implements" implements+)?
4           "{" classBody* "}";
```

Based on the concrete syntax the source code can be divided into tokens, which are the smallest components of a programming language. This task is performed by a lexer that is generated by

EMFTextEdit. According to the concrete syntax definition the lexer uses the symbols to find the elements of a language. The generation of the lexer and parser is carried out by ANTLR [PQ95].

The tokens found by the lexer are analyzed by a parser, which is also generated by EMFTextEdit. The parser structures the linear stream of tokens according to the abstract syntax definition and generates a tree notation of the program called Abstract Syntax Tree (AST). This tree provides detailed information about the source code. Furthermore, the abstract syntax definition is not only used to generate a parser, but also to derive a Ecore model [BBM03]. This is the meta model we will use for the language under consideration. Alternatively the definition of the abstract syntax can be directly established using an Ecore model. This choice is up to the user, but does not affect the analysis.

To allow the parser to instantiate models, concrete Java classes are generated from the Ecore model. Objects of this classes are used to represent the abstract syntax of source code fragments as an AST. The generation of these Java classes is carried out by the EMF.

In summary, EMFTextEdit allows to automatically derive an Ecore model, both a lexer and parser from the abstract and concrete syntax definition of a language. Given a source file of this language we can now easily obtain a model instance of the source code fragment at hand. Such model instances will serve as the starting point for any of our analyses. Additionally, EMFTextEdit generates an Editor plug-in for the defined language that supports text editing features like syntax highlighting.

3 Static Analysis with OCL

Static analysis is based on a suitable representation of a program's source code. This representation can be quite different depending on the type of analysis to be performed and the underlying formalism. Abstract interpretation, for example, uses lattice structures to embody a program's state [CC77]. Other approaches, that are based on logic translate source code into logical formulae and perform analysis based on deduction [RSW04]. Opposed to this complex program representations, we use a more basic representation of programs for our analysis, namely the AST. This implies several consequences, such as a more restricted analysis power, which will be discussed more detailed later on.

Following the procedure presented in Sect. 2, we are able to obtain an AST for source code written in arbitrary languages. This AST is the basis of our analysis. Using a standard OCL implementation user-defined OCL queries can be evaluated on the AST instance. The whole procedure is recapitulated in Fig. 2.

Users (e.g., developers or quality assurance engineers) can now define OCL query expressions that select invalid AST elements. For example names that break coding guidelines can be found, illegal method calls to internal classes can be detected or classes with too many members can be mined from the source code. Selecting invalid AST elements instead of valid ones has usability reasons. In practice, the major part of an AST is expected to be correct and only few elements violate coding conventions. Therefore, defining invalid AST elements is easier than defining all valid AST's. Because the OCL queries narrow the set of legal documents, we use the term *restrictions* to refer to the queries.

Restrictions specified in OCL can be specified organization-wide and reused for multiple

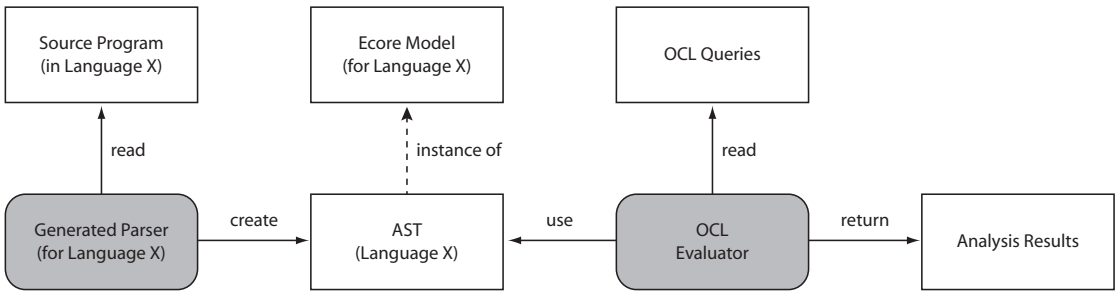

Figure 2: Analysis Process

projects (e.g., coding guidelines). They can also be generated (e.g., within model driven processes). Depending on the defined queries both very specific and broadly applicable restrictions can be formulated. Disallowing access to a particular method is an example for the former category, while limiting the number of members for classes belongs to the latter one. Whatever information is present in the AST and which can be extracted by an OCL query, may be incorporated into an analysis. Note that this is not limited to object-oriented languages as the previous examples may suggest. Sections 5.2 and 5.3 provide examples for quite different types of languages.

The actual OCL expressions to be used, depend on the concrete language that is subject to the analysis, the degree of abstraction determined by the syntax definition and the purpose of the analysis. Section 5 contains some possible application scenarios. These include, but are not restricted to:

- Checking coding guidelines (e.g., naming conventions, documentation rules)
- Detecting security vulnerabilities
- Enforcing extended visibility concepts
- Calculating code metrics to find design flaws

Depending on the concrete application scenario the user must specify OCL expressions that traverse the AST, find the elements that must be incorporated into the analysis and check their properties or accumulate them. For example, to encourage meaningful names for interfaces defined in a set of Java sources, the user can traverse the AST to collect all elements that are of type `Interface` and check that the names have a minimal length. A respective OCL query might be phrased as shown in Listing 1.

Listing 1: OCL expression selecting interfaces with short names.

```
1 self.typeDeclarations->select(
2     i | i.oclIsKindOf(Interface) and
3     i.oclAsType(Interface).name.size() < 10)
```

In line 1 the type declarations of the compilation unit are selected. Line 2 constrains the set of declarations to contain interfaces only and the expression in line 3 checks whether the name

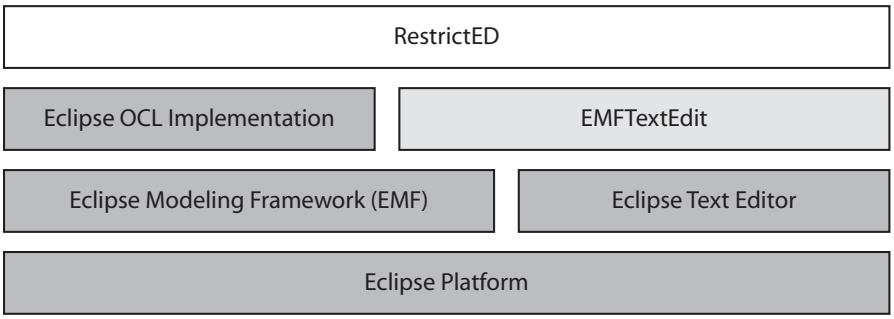

Figure 3: Conceptual design of RestrictED.

of the interfaces is shorter than 10 characters. More OCL queries are presented and discussed in Sect. 5.

4 Implementation

To evaluate the capabilities of a static analysis tool based on OCL, we implemented a prototype called RestrictED[1] based on Eclipse [Ecl]. We used EMF [BBM03] and the Eclipse OCL implementation, which is part of the Model Development Tools (MDT) project, to evaluate queries on model instances. Figure 3 depicts how RestrictED integrates with EMF, the Eclipse OCL implementation and other components.

RestrictED relies on EMFTextEdit, which is part of the Reuseware framework. EMFTextEdit creates an AST for each open document. Since this AST is based on Ecore, it can be easily used for the evaluation of the OCL queries. This work is carried out by the MDT, that were originally designed for use with graphical models, but can examine any Ecore-based structure. If a file is opened with EMFTextEdit, RestrictED looks for restrictions that apply to this document.

By convention two different types of restrictions can apply to a file. Global restrictions affect all files in a project and local restrictions refer to one particular document only. For example, global restrictions are useful to declare naming conventions, while local ones might be used to specify internal access constraints for a particular document (see Sect. 5.1). Global restrictions must be placed in a file called “restrictions.ocl” located in the project root directory. Local restrictions need to be named after the file they refer to and extended by the suffix “.restrictions.ocl”.

Both global and local restriction documents contain OCL expressions accompanied with an error description. The latter shortly describes what kind of analysis is performed by the OCL query. RestrictED reads both the queries and the error descriptions. Now that both the AST of the open document and the respective OCL queries are available, the root node of the AST is given to the Eclipse OCL implementation and all queries are evaluated. If queries return results that are not empty, RestrictED notes violations in the problems view, as depicted in Fig. 4.

The evaluation of the OCL queries starts at the root node of the AST. We consider this a natural context for the evaluation. Accordingly, all queries must take the structure of the AST

[1] Available from http://www.inf.tu-dresden.de/~ms72/RestrictED/

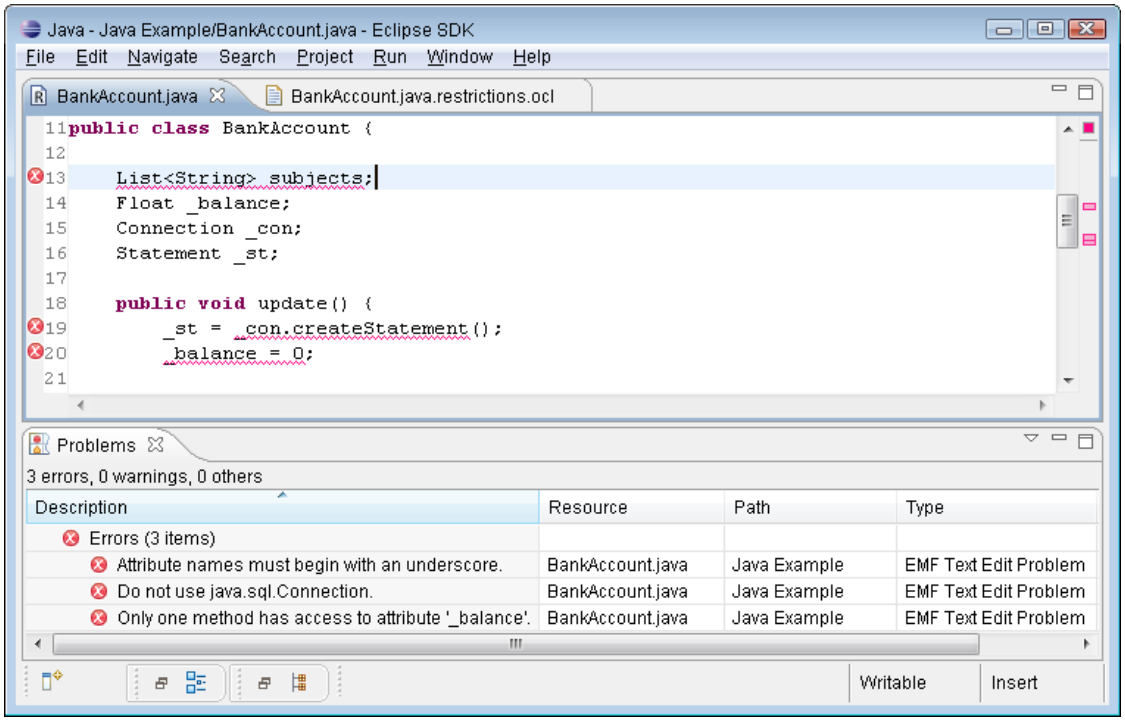

Figure 4: Notifications about violated restrictions.

into account to select appropriate elements. To create queries, one must have knowledge of the underlying meta model. For example, OCL conditions on class declarations, method calls, and attribute declarations (defined in a meta model for Java) can be used to analyze Java programs.

5 Evaluation

To verify the practical applicability of our approach, this section evaluates the proposed concepts by means of three different languages. Even though this can not be considered an exhaustive validation it will reveal the advantages and drawbacks of our method. The use case scenario, which frames our evaluation, is a notional bank application.

Our example application is composed of three different parts. A Java part examines the execution of transactions and encapsulates the respective business logic. All account information is saved in a database, which is accessed using Java Database Connectivity (JDBC). All database queries are externalized from Java source files. Accordingly, the second component of our example is the set of SQL queries, which is used to read and write persistent account information. Finally, we used a small DSL for state machines to incorporate existing business process rules, such as the life cycle of accounts.

Within this example application different scenarios for source code analysis can be found. For example, Java code which effects the balance of an account should have very restricted access. Furthermore, it should use only prepared statements to avoid security vulnerabilities,

such as SQL injection. SQL statements in turn should be limited in the set of data they can modify. Under no circumstances should they alter the database scheme. The state machines must be checked for correctness criteria (e.g., deadlock-freeness). And, as a matter of course, the complete code should follow our style guidelines.

Note that even though this particular scenario can be addressed by existing concepts (e.g., visibilities in Java and database access rights), our approach is more flexible, because no language specific tool implementation is needed and problems are detected at the implementation time. This is superior to detecting bugs at runtime or using test cases to ensure correct behavior.

5.1 Analyzing Java Code

Accurate and applicable coding rules can increase the reliability and reduce the number of faults in software [BM08]. This applies to all programming languages in general and to Java in particular. A very common coding rule is adhering to naming conventions. For example, using the prefixes `get` and `set` for the names of getter and setter methods can be considered almost an unwritten law in the Java community. Another frequently recommended guideline is to name attributes different from method parameters or local variables.

To show that we can check the adherence to such naming conventions, we will exemplarily enforce attribute names that begin with an underscore. An OCL query that specifies this rule is shown in Listing 1.

Listing 1: OCL query to enforce attribute names with a prefix.

```
1 self.typeDeclarations->select(class | class.oclIsKindOf(Class)).
    oclAsType(Class).members->select(v | v.oclIsKindOf(Variable))
    .oclAsType(Variable)->select(var| var.name.substring(1,1)<>'_
    ')
```

The query searches for class declarations, examines all members that are attributes and compares the first character of the respective names with the underscore character. Note that our meta model uses the type `Variable` both for local variables and attributes.

One can already guess from this basic example that queries for more sophisticated rules will be very complex. In particular checking types with `oclIsKindOf` and converting types with `oclAsType` disrupts the readability of the queries.

A second common example for coding rules is to enforce bounds for certain metrics. For example, restricting the maximum number of methods per class can be beneficial. This will encourage programmers to split functionality across classes, which will in turn enable reuse, modularity and maintainability. An OCL query to check such a rule is shown in Listing 2.

Listing 2: Enforce upper limit for number of methods per class.

```
1 if self.typeDeclarations->select(class | class.oclIsKindOf(Class)
    ).oclAsType(Class).members->select(method | method.
    oclIsKindOf(Method))->size() > 15 then self else null endif
```

Again, the query searches for class declarations, but this time members of type `method` are considered. If the number of methods is less or equal to 15 **`null`** is returned, which indicates that

there is no rule violation. Otherwise the result of the query is `self`, which denotes a problem in this compilation unit.

The previous examples have shown that we can check basic properties of Java source code. To take the evaluation one step further, we specified more complex analyses. First of all, we realized advanced access control. Access modifiers in Java (e.g., **`public`** and **`private`**) do allow only a very limited set of access restrictions. However, our bank application requires strict separation of business logic and database queries. Developers should not be allowed to bypass a designated database access layer and create or execute SQL statements. For this purpose using objects of type `java.sql.Connection` must be prohibited. This can not be achieved using Java built-in access modifiers. Listing 3 contains the OCL query, which enforces this rule.

Listing 3: OCL query for method calls to java.sql.Connection.

```
1 self.typeDeclarations->select(class | class.oclIsKindOf(Class)).
    oclAsType(Class).members->select(method | method.oclIsKindOf(
    Method)).oclAsType(Method).body->statements->select(ass| ass.
    oclIsKindOf(Assignment)).oclAsType(Assignment).value->select(
    vr| vr.oclIsKindOf(VariableReference)).oclAsType(
    VariableReference)->select(v|v.variable.type.oclAsType(Class)
    .import = 'java.sql.Connection')
```

The query looks up all assignments in all method bodies and checks the type of the referenced variables. If a variable of type `java.sql.Connection` is used a problem will be reported. Again, the size of the query, one of the major drawbacks of our approach, becomes apparent. Even though an OCL expert can read and interpret the query, most people would consider such a longish specification useless.

Due to space limitations we omit more examples for OCL queries. Our prototype comes with an example workspace that contains all queries mentioned in this paper and some more. Among other things, we created a query that restricts access to attributes within classes. This was in particular useful to define the methods which are allowed to access important data, such as the balance of an account. The goal of this restriction was to reduce the lines of code that modify important data to the minimum. Erroneous modifications, such as thread-unsafe access to an attribute can be detected more easily this way.

5.2 Analyzing SQL Statements

Our example application uses SQL queries to read and write account information to a database. Obviously, erroneous queries can cause great amounts of damage. They can either accidentally modify data which is supposed to be static (e.g., a customer's name) or, even worse, alter the database scheme. Hence, restricting the set of legal queries can prevent bugs at the time of implementation and reduce the number of runtime errors or the amount of required test cases.

A very basic restriction is to ban `drop table` statements. Since all records are lost if this command is executed, its use should be detected and result in an error message. At first glance this seems to be enforceable by using access rules, which are provided by database systems. But, opposed to access rules, which are checked at runtime, we can detect rule violations at development time. To prevent `drop table` statements the OCL query in Listing 4 can be used.

Listing 4: OCL expression disallowing dropping of tables.

```
1 self.sqlStatements->select(dp| dp.oclIsKindOf(Drop))
```

This query basically iterates over all SQL statements and checks their type to be `Drop`. Thus, the query selects `drop table` statements as we consider them to be illegal. In contrast to dropping tables, changing and inserting data must obviously be allowed. However, not all update operations are desired. Consider for example the two queries shown in Listing 5.

Listing 5: Examples of SQL queries.

```
1 update accounts set value=500000 where owner='My Name';
2 update accounts set owner='My Name' where value>100000;
```

While the statement in the first line solely changes the balance of an account, the second one modifies the name of a customer for a particular account. Since our bank has a policy that accounts can not be transferred from one customer to another, we don't want our application to accidentally support this "feature". In terms of SQL queries this implies that updates to column `owner` must be prohibited. The respective OCL query to enforce this rule is shown in Listing 6.

Listing 6: OCL expression disallowing changing data in column "owner".

```
1 self.sqlStatements->select(
2      us| us.oclIsKindOf(Update)).
3      oclAsType(Update).list->select(
4      col| col.columnName.value = 'owner')
```

The query selects all `UPDATE` statements and compares the name of the table, which is subject to the update, to `owner`. Depending on the particular needs of an application other restrictions can be useful too. For example, developers of performance critical applications might not want to use columns in `WHERE` clauses, which do not have a corresponding index. Using a general query language to select illegal code fragments obviously enables very flexible application scenarios. It can also be observed, that detecting SQL commands which undermine design intentions at implementation time is better than using classical rights management provided by database systems. Additionally, it is more efficient to define coding conventions for all languages employed in a development process using one single mechanism instead of several ones. For example, using OCL queries for Java programs and using rights management to enforce database integrity implies learning more concepts and mastering another technology.

5.3 Analyzing a DSL for State Machines

As mentioned in the introduction, the fast-growing application of DSLs within software development processes is a strong motivation for generic source code analysis. To confirm the applicability of our method to this context, we used a textual DSL for state machines. This DSL allows to define states, transitions, triggers and actions. States can be start states, end states or ordinary states.

Our example application uses this DSL to model the life cycle of accounts. Newly opened accounts have the state "regular". After five years, accounts can attain the status of a "gold" account, which will allow customers to benefit from favorable interest rates. If a customer closes

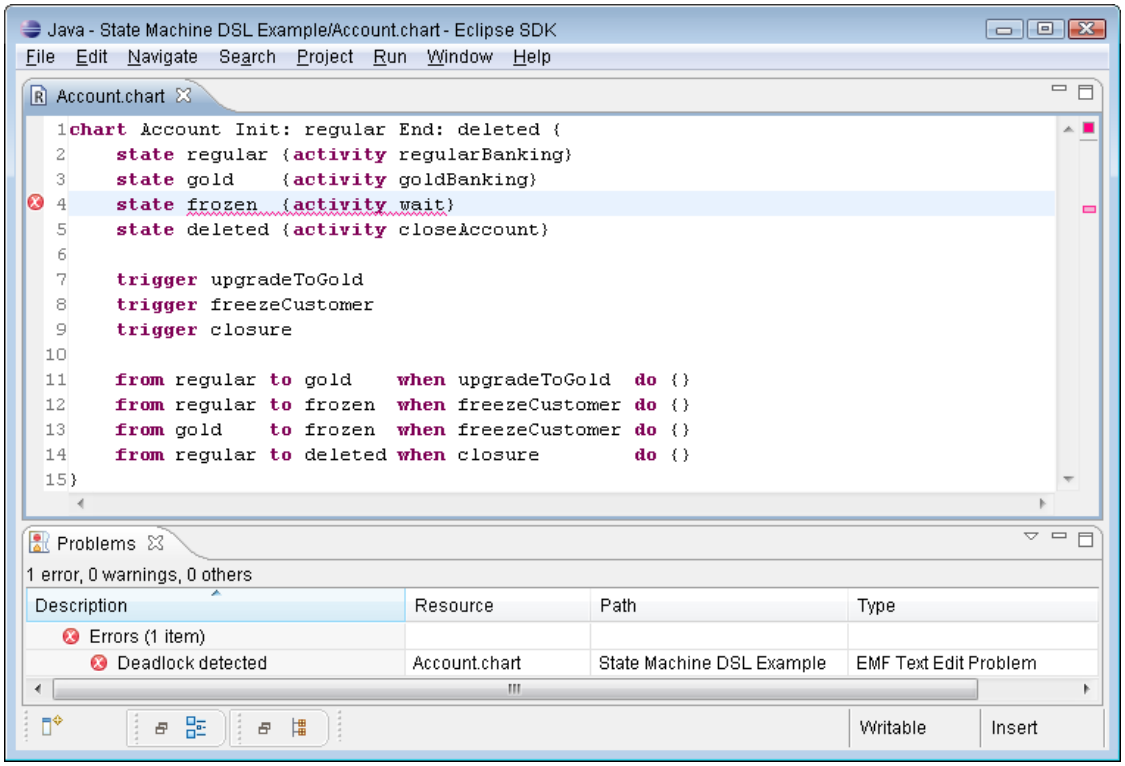

Figure 5: Specification of an account's life cycle in a textual DSL.

his account, its state will change to "deleted". Furthermore, legal regulations require that every account can be frozen. The complete description of this life cycle is shown in Fig. 5.

The states mentioned above can be found in lines 2 to 5 and the respective transitions are defined in lines 11 to 14. To check that the life cycle follows predefined rules we utilized OCL queries. For example, our application requires that the state of an account can change except that it reached an end state. This is similar to checking models for deadlocks. To avoid that an account's state can not change anymore, the OCL query shown in Listing 7 was defined.

Listing 7: OCL expression checking for deadlocks.

```
1 (self.elements->select(tr | tr.oclIsKindOf(Transition))
2      .oclAsType(Transition).target)->reject(w| (self
3      .elements->select(tr | tr.oclIsKindOf(Transition))
4      .oclAsType(Transition).source)->includes(w))
5      ->reject(end | (self.end)->includes(end))
```

The query searches for states without outgoing transitions that are not end states. As shown in Fig. 5, our life cycle does not fulfill this requirement since the state `frozen` has no outgoing transition and is not an end state either. To fix this `frozen` must be either added to the list of end states or a transition to `deleted` must be added. Specifying such a consistency rule also guarantees that future modification to the state model do not violate business rules.

We created more OCL queries (e.g., to limit the maximum number of end states) and of course the previously presented checks (e.g., to enforce naming conventions) can be applied here too. The tiny life cycle example confirms that applying source code checks to a textual DSL can be as beneficial as applying them to widespread programming languages, but suffers from the same problems. In particular the specification of the complex OCL queries posts a major restriction on the usability of the approach.

6 Related Work

Analyzing source code has received attention in computer science for quite a long time. The great amount of existing work can be divided into static and dynamic analysis methods. While static methods analyze programs without executing them, the dynamic approaches run code to perform an analysis by observing a programs execution and potentially modifying the program beforehand. Since our approach does not need to execute programs it belongs to the first category.

Static analysis can be used to detect potential or actual coding errors, bad code smells, violations of style guidelines, proof program properties or even determine resource consumption. Depending on the application different methods can be used to analyze programs. For example model checking can be utilized [HP00], abstract interpretation may be used [CC77] or logic-based approaches are applied [RSW04].

Tools used to find coding errors (e.g., FindBugs [HP07] or SPLint [EL02]) are usually language specific because they heavily rely on the formal semantics of the language that is subject to the analysis. Some of these tools are extensible in the sense that one can add new patterns that shall be detected in the source code. To define these patterns all tools use custom specialized languages. Thus, using a new tool involves learning a pattern definition language. We consider this a major drawback compared to our method, where a standard language (OCL) is used for pattern definition.

Other tools (e.g., Checkstyle [Bur]) that check coding guidelines are not inherently bound to a specific language, because they are not tied as much to the underlying semantics. For example checking naming conventions can be performed with very few knowledge about a languages semantics. With the exception of SemmleCode [HVM06] and “Rough Auditing Tool for Security (RATS)” [For], all existing tools we are aware of deal with one particular language only. SemmleCode supports Java and XML, while RATS can handle C, C++, Perl, PHP and Python code. Our approach pursues this idea and shows that basic rules can be checked using a language-independent approach. We can check naming conventions, restrict calls to methods or limit applicable parameter types. Out of our scope is making statements on the results of calculations. Nevertheless, we think that a language-independent approach to solve basic static analysis can complement existing specialized tools.

Using OCL to define problems or style-guide violations is just one way to do so. Usually specialized query languages are used to select program elements from source code [Vol06]. The tailoring process that preceded these languages makes them more easy to use, but also implies the implementation of a custom query processing engine. We eliminate this effort, because we use existing OCL interpreters. Thus, our approach can be applied to new languages without implementing a query processor.

7 Conclusions and Future Work

In this paper we have shown how meta models of textual languages can be used in conjunction with OCL to create a tool that performs static source code analysis. The presented prototype can be reused to analyze arbitrary languages. The only prerequisite is the availability of a concrete and abstract syntax definition.

The benefits of the approach at hand are the ability to reuse both the analysis tool and the analysis language. Using a standard language, namely the OCL, does not only prevent developers from learning new analysis specification languages over and over again, but opens up a new application area for OCL. Furthermore, OCL provides concepts (e.g., navigation) that must be coded by hand when analysis is performed with standard program code.

The evaluation has shown that useful application scenarios are covered by our method, such as checking coding conventions (e.g., naming standards), enforcing security aware coding or implementing extended visibility concepts. Especially coding conventions and visibility constraints can be applied to various languages.

The downsides of the presented analysis method are mostly induced by the choice of representing programs using AST's and using a generic analysis language. As OCL queries are used to specify concrete analyses, the approach is facilitated and restricted by the expressive power of OCL at the same time. Whatever can be checked by an OCL query is supported by our tool, but nothing else. Moreover, the OCL expressions tend to grow with the complexity of the language that is subject to the analyses. This makes the definition of OCL queries sometimes more complex compared to a custom analysis language. Reuse and generation of queries may be solutions to tackle this complexity.

Furthermore, the representation using AST's also limits the power of our analysis. Since the static structure is embodied rather than the programs state, our analysis can hardly find errors that are exposed by the dynamic semantics. This limits our analysis compared with abstract interpretation. Another limitation specific to our implementation is that it can not be used with arbitrary text editors straight away. As stated before, the presented solution is based on the previously mentioned components, notably EMFTextEdit. In principal the RestrictED plug-in can be used with any editor. However, the Eclipse OCL implementation is limited by the fact that it can only analyze structures that are based on Ecore. Therefore, compatible editors must offer an Ecore-based AST.

All in all we can not perform as sophisticated analysis as language specific tools do, but the presented approach covers many useful application scenarios. The depicted method is based on a standard language (OCL) and can be applied for all textual languages that can be defined with EMFTextEdit. This enables static analysis for arbitrary programs using a single tool and a single language.

In the future we plan to evaluate our prototype using new languages. Especially textual DSLs form interesting use cases. We need to determine where OCL reaches its limits when used for source code analysis. A clear understanding is needed about the possible types of analysis. For existing languages the question is to what extent our method can compete with specialized tools [HP07, EL02].

Furthermore, future work will address the complexity of the OCL queries, which we consider a major drawback of our analysis method. Using the concrete syntax of the analyzed language

itself to specify queries seems a promising goal. Since users are much more familiar with the concrete syntax of their languages, defining queries in the same syntax is much easier than using OCL. To achieve this, the concrete syntax must be extended and queries must be translated to OCL. Last but not least, generating the OCL queries (e.g., within MDD environments) can be another opportunity to hide complexity from users.

Acknowledgements: This work has been partially funded by the German Ministry for Education and Research in the SuReal project. We thank Steffen Zschaler for provoking this work, as well as Simone Röttger, Jendrik Johannes, Christian Wende and Florian Heidenreich for their fruitful comments on earlier versions of this paper. We also thank the anonymous reviewers for their constructive criticism that encouraged use to improve and continue our work.

Bibliography

[BBM03] F. Budinsky, S. A. Brodsky, E. Merks. *Eclipse Modeling Framework*. Pearson Education, 2003.

[BM08] C. Boogerd, L. Moonen. Assessing the Value of Coding Standards: An Empirical Study. In *Proceedings of the 24th IEEE International Conference on Software Maintenance, September 28 to October 4, 2008, Beijing, China, To appear*. 2008.

[Bur] O. Burn. Checkstyle Homepage. http://checkstyle.sourceforge.net.

[CC77] P. Cousot, R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Pp. 238–252. ACM Press, New York, NY, Los Angeles, California, 1977.

[Ecl] Eclipse Foundation. Eclipse.org home. http://www.eclipse.org.

[EL02] D. Evans, D. Larochelle. Improving Security Using Extensible Lightweight Static Analysis. *IEEE Software* 19(1):42–51, 2002.

[For] Fortify Software Inc. Rough Auditing Tool for Security (RATS) Website. http://www.fortify.com/security-resources/rats.jsp.

[HP00] K. Havelund, T. Pressburger. Model Checking Java Programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer* 2(4):366–381, March 2000.

[HP04] D. Hovemeyer, W. Pugh. Finding bugs is easy. *SIGPLAN Notices* 39(12):92–106, 2004.

[HP07] D. Hovemeyer, W. Pugh. Finding more null pointer bugs, but not too many. In Das and Grossman (eds.), *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE'07, San Diego, California, USA, June 13-14, 2007*. Pp. 9–14. ACM, 2007.

[HVM06] E. Hajiyev, M. Verbaere, O. de Moor. CodeQuest: Scalable Source Code Queries with Datalog. In Thomas (ed.), *ECOOP*. Lecture Notes in Computer Science 4067, pp. 2–27. Springer, 2006.

[PQ95] T. J. Parr, R. W. Quong. ANTLR: A Predicated-LL(k) Parser Generator. *Software — Practice and Experience* 25(7):789–810, 1995.

[Reu] Reuseware Team. EMFTextEdit Website. http://emftextedit.reuseware.org.

[RSW04] T. W. Reps, S. Sagiv, R. Wilhelm. Static Program Analysis via 3-Valued Logic. In Alur and Peled (eds.), *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*. Lecture Notes in Computer Science 3114, pp. 15–30. Springer, 2004.

[The06] The Object Management Group. OCL 2.0 Specification. Technical report, May 2006.

[Vol06] K. D. Volder. JQuery: A Generic Code Browser with a Declarative Configuration Language. In Hentenryck (ed.), *PADL*. Lecture Notes in Computer Science 3819, pp. 88–102. Springer, 2006.