A Flexible Framework for Quality Assurance of Software Artefacts With Applications to Java, UML, and TTCN-3 Test Specifications

Jens Nödler*, Helmut Neukirchen**, Jens Grabowski*

* Software Engineering for Distributed Systems Group, Institute for Computer Science,
Faculty for Mathematics and Computer Science, University of Göttingen,
Goldschmidtstr. 7, 37077 Göttingen, Germany

** Faculty of Industrial Engineering, Mechanical Engineering and Computer Science
University of Iceland, Dunhagi 5, 107 Reykjavík, Iceland

E-mail: jens@noedler.de, helmut@hi.is, grabowski@cs.uni-goettingen.de

Abstract

Manual reviews and inspections of software artefacts are time consuming and thus, automated analysis tools have been developed to support the quality assurance of software artefacts. Usually, software analysis tools are implemented for analysing only one specific language as target and for performing only one class of analyses. Furthermore, most software analysis tools support only common programming languages, but not those domain-specific languages that are used in a test process. As a solution, a framework for software analysis is presented that is based on a flexible, yet high-level facade layer that mediates between analysis rules and the underlying target software artefact; the analysis rules are specified using high-level XQuery expressions. Hence, further rules can be quickly added and new types of software artefacts can be analysed without needing to adapt the existing analysis rules. The applicability of this approach is demonstrated by examples from using this framework to calculate metrics and detect bad smells in Java source code, in UML models, and in test specifications written using the Testing and Test Control Notation *(TTCN-3).*

1. Introduction

A multitude of software artefacts are created during the different activities of software development, e.g. models, source code, or test specifications. Reviews, walkthroughs, and inspections [9, 12] are used to find flaws in software artefacts even at an early stage; however, manual review is time consuming and thus, automated analysis tools are used to support and complement the quality assurance of software artefacts [36].

When we wanted to assure the quality of test specifications that are specified using the standardised *Testing and Test Control Notation* (TTCN-3) [7, 14], we experienced that the available software analysis tools supported only common programming languages, but not those domain-specific languages that are used in a test process. Hence, we started to implement our own analysis tool for TTCN-3 test specifications [20]. In that tool, the analysis rules were hard-coded in Java involving low-level de-referencing of pointers in a TTCN-3 syntax tree [19]: implementing new analysis rules required considerable efforts and an easy addition of user-defined analysis rules was not possible. Thus, we were looking for an approach that supports a high-level specification of analysis rules that can easily be created by an end-user.

This paper presents our solution to this problem: an XML and XQuery-based framework for the automated analysis of software artefacts. The framework is based on an abstraction layer that decouples analysis rules from syntactical details of the parsed target software artefact. This makes it easy to add analysis rules as they are specified in a high-level style. A welcome side affect is that the abstraction layer allows existing analysis rules to be re-used without modification for different targets or to re-use the abstraction layer for different classes of analyses. Hence, it is easy to support domain-specific languages as analysis targets.

The structure of this paper is as follows: following this introduction, we provide in section 2 foundations of software analysis, on the XML technologies used in our framework, and on the representation of software artefacts. Our main contribution can be found in sections 3 and 4: section 3 explains the architecture of our analysis framework, whereas section 4 describes results from validating the framework by applying it for varying targets and classes of analyses and by integrating it into an existing *Integrated Development Environment* (IDE) for TTCN-3. Section 5 discusses related work. Finally, we conclude with a summary and outlook in section 6.

2. Foundations

The term “software artefact” relates to all items that are created as part of a software development process, e.g. documents containing requirements, models, source code, scripts, or test specifications.

In this paper, we assume that a software artefact is machine processable, i.e. it adheres to a well-defined syntax that has an associated semantics. This assumption allows the analysis of software artefacts using automated tools.

2.1. Software analysis

Software artefacts can be analysed in various automated ways to determine their properties: by dynamic analysis, by static analysis, or, for example, by model checking. In this paper, we restrict ourselves to static analysis.

Often, static analysis is related to the evaluation of queries on software artefacts or to the detection of patterns in software artefacts [6]. Thus, support for specifying patterns and queries in software artefacts is a vital requisite for software analysis. Other typical applications of automated static analysis in the context of software quality assessment are the calculation of product metrics [10] or the detection of “bad smells in code”, i.e. certain structures in code (such as duplicated code) that are indicators of bad internal quality and should thus be resolved by refactoring [11].

2.2. XML technology

The *Extensible Markup Language* (XML) [5] is a flexible text format for the exchange of any kind of data. As its name implies, the language addresses the markup of data by adding structural information to it. Essentially, XML provides *elements* that can be hierarchically nested resulting in a tree-like structure, where the nodes are the XML elements. Each element may optionally contain further *attributes* with *values*. Between the opening and closing tags of an element, some textual *content* may be stored. To illustrate the usage of XML, an example of a new defined language is given in listing 1 that structures a few seasons, episodes, and titles of the TV sitcom “The Simpsons” as XML.

XQuery [28] is a standardised language for querying XML documents. It is a functional and typed language [17] which is Turing complete [15]. XQuery 1.0 is a superset of the XPath 2.0 language providing more powerful capabilities: while XPath only permits the selection of nodes

```xml
<?xml version="1.0" encoding="UTF-8"?>
<simpsons>
  <season number="1">
    <episode number="1" title="Simpsons Roasting on an
         Open Fire"/>
    <episode number="2" title="Bart the Genius"/>
  </season>
  <season number="19">
    <episode number="401" title="He Loves to Fly and He
         D'oh's"/>
    <episode number="402" title="The Homer of Seville"/>
  </season>
</simpsons>
```

Listing 1. XML example: “The Simpsons”

```
1 for $episode in doc("simpsons.xml")//episode
2 let $title := data($episode/@title)
3 order by $title
4 return element simpsons-episode { $title }
```

Listing 2. Simple XQuery expression

from an XML document, XQuery supports the manipulation, transformation, creation, and sorting of nodes. The syntax of XQuery looks like a mixture of XPath and SQL and is therefore easy to use for anyone who is familiar with these languages. Listing 2 shows a sample XQuery expression which queries some data from the XML document shown in listing 1 and returns new XML elements containing episode titles: The function `doc()` in line 1 of listing 2 loads the XML document tree. Then, path expressions can be used to navigate to nodes of the XML tree and to subsequently select nodes: `/` separates the nodes of a path expression, `//` allows the selection of nodes just by using their element name without needing to provide a full path. Hence, `//episode` selects all `episode` elements of the loaded document. The function `data()` in line 2 retrieves values of nodes: `$episode/@title` refers to the `title` attribute of elements stored in the variable `$episode`. The selected `episode` nodes are ordered by their title (line 3) and returned as a set of `simpsons-episode` XML elements whose tags enclose the title of each episode (line 4).

More complex patterns can be matched by composing queries. XQuery also allows the definition of user-defined functions and modules for grouping related functions. An example function that creates an XML element `hello` containing the content `Hello World!` is shown in listing 3.

2.3. Representation of software artefacts

Software needs to be represented in varying ways to fulfil different requirements: Software architects require a global view of a system. Therefore, they require software representations using abstract artefacts like models. In contrast, source code analysis tools need to investigate all details of a software to find design flaws or potential bugs and require

```
module namespace my-namespace = "foo-namespace";

declare function my-namespace:hello-world() {
  element hello { "Hello World!" }
};
```

Listing 3. XQuery module and function

```
1  public class Person {
2    private String name;
3    public String getName() {
4      return name;
5    }
6  }
```

Listing 4. Java source to be encoded as XML

a representation of the source code artefact that reflects the complete source code. Thus, different representations of software artefacts are used in software development.

A model is an abstraction of the real world. A metamodel defines a language for the specification of models [24]. Different metamodel approaches for software representation can be differentiated by their granularity of the representation of the underlying software. Some, for instance the FAMIX metamodel [32], concentrate on the most important object-oriented entities and their relationships. Others provide a more detailed approach for the representation of Java source code as a metamodel [34]. Metamodels are, however, not limited to the representation of software on a high-level view, but can also be used to represent source code in depth like a metamodel that covers different object-oriented languages [35].

The *Unified Modeling Language* (UML) [24] is a standardised general-purpose modelling language. It is used to create abstract models of systems, called *UML models*. UML has a major impact in the field of software engineering. However, in most cases, UML is used for designing and representing software on an architectural level and not as a representation format for source code.

The *XML Metadata Interchange* (XMI) [23] is a standardised XML language for serialising and exchanging metadata information such as metamodels. XMI is commonly used as an exchange format for UML models and metamodels.

XML is not only used for the exchange of XMI-based models, metamodels, or other structured information but also serves as a language for representation formats. In the following, the usage of XML for the detailed representation of software artefacts, e.g. source code, is presented. Listing 4 contains sample Java source code of a class that will be encoded as XML in the following examples.

Listing 5 shows the encoded Java source code in an XML representation called *JavaML* (Java Markup Language) [4]. It maps all language constructs of Java to XML

```
1  <class name="Person" visibility="public">
2    <superclass name="Object"/>
3    <field name="name" visibility="private">
4      <type name="String"/>
5    </field>
6    <method name="getName" visibility="public">
7      <type name="String"/>
8      <formal-arguments/>
9      <block>
10       <return><var-ref name="name"/></return>
11     </block>
12   </method>
13 </class>
```

Listing 5. JavaML representation of listing 4 (some XML attributes stripped for reasons of clarity)

```
1  <class><specifier>public</specifier> class <name>Person
        </name> <block>{
2    <decl_stmt><decl><type>private <name>String</name>
          </type> <name>name</name></decl>;</decl_stmt>
3    <function><type>public <name>String</name></type>
          <name>getName</name><parameter_list>()
          </parameter_list> <block>{
4      <return>return <expr><name>name</name></expr>;
            </return>
5    }</block></function>
6  }</block></class>
```

Listing 6. srcML representation of listing 4

nodes. For example, the source code `public class Person` (line 1 of listing 4) is mapped to the XML element `class` (line 1 of listing 5) with the attributes `name="Person"` and `visibility="public"`. JavaML is also capable of encoding information about the location of each statement in the original source code (using line, end-line, column, and end-column XML attributes).

Another example for the XML representation of source code is *srcML* (Source Code Markup Language) [16]. Listing 6 contains the result of the annotation of the example Java source code. All constructs and also the formatting of the source code are preserved and XML elements are just put around the Java constructs. In line 1 of listing 6, the start of the Java class is marked-up using a `class` element, the visibility of the class is marked-up with a `specifier` element and the name of the class with a `name` element.

Another approach to encode source code as XML is the mapping of an *Abstract Syntax Tree* (AST) to XML. As an AST is abstract, it represents only semantically relevant parts of the source code (like statements and identifiers). Most ASTs contain the line numbers and offsets for all statements of the underlying programming language, this information can also be mapped to XML attributes.

Sometimes, it is even not necessary to encode a language as XML: domain specific languages that are used in model-driven engineering, testing, or for configuration management often use XML as their native format.

As shown, XML can be used as a universal representation format for many kinds of software artefacts from abstract high-level software models to fully detailed source code, but XML is also used as graph representation format, thus enabling the representation of, e.g., data and control flow graphs. "This makes XML [...] a natural choice to be used as [...] representation format for program representations" [1].

3. The XQuery-based Analysis Framework

For querying and detecting patterns in software artefacts on a high level, we have developed the XQuery-based Analysis Framework (XAF) [22]. It is a highly flexible framework that supports the automated static analysis of arbitrary software artefacts. The framework is independent of any concrete language to be analysed and can easily be adapted to specific languages. Its most important features are:

- generic pattern and query description on a high level of abstraction, thus enabling the easy addition of new analysis rules,
- extensibility regarding new types of patterns and new analysis targets, and
- independency of the patterns and queries from specific analysis targets like concrete programming or domain-specific languages.

These goals are achieved through a layered and extensible framework design. The design makes use of the *facade* design pattern [13] which intermediates between concrete analysis targets and the actual analyses. The facade layer serves as a fixed interface for the analysis layer and provides for each underlying analysis target a corresponding implementation of the facade layer interface. The analysis layer is designed as a plug-in architecture that allows the framework user to easily add new analysis rules or classes of analyses.

As discussed in section 2.3, XML can be used as an universal representation format for all kinds of software artefacts. Hence, XAF uses XML as internal representation of any software artefact and thus, if the software artefact under investigation has not XML as its native format (as it is often the case for domain-specific languages), the software artefact must be converted into an XML representation (for source code, e.g., by making use of one of the representations presented in section 2.3; for models, e.g., by using XMI). This permits the usage of XAF for the analysis of various kinds of software artefacts independent of their original syntax as long as a corresponding XML representation is available.

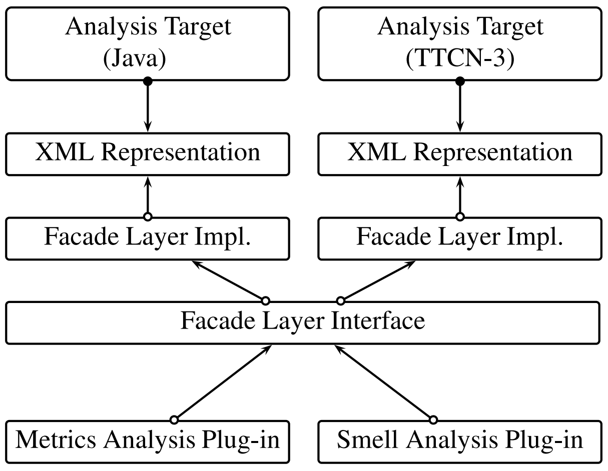


Figure 1. Framework design: plug-ins extend the framework and are independent of analysis targets

The rules for analysing a software artefact are described as XQuery functions that query XML data. Hence, an analysis plug-in for the analysis layer of XAF is composed of XQuery functions that are specific to the kind of analysis to be performed. The XQuery expressions do not operate directly on the concrete XML representation of the specific software artefact—instead they access the software artefact through the XAF facade layer that converts between the possibly varying XML representations of the software artefacts and the fixed interface provided by the XAF facade layer. Hence, an end-user only needs to know the fixed interface instead of the concrete XML representation to write new queries. The layered design of XAF allows the construction of analysis plug-ins independent of any concrete analysis target by having only a dependency to the fixed interface of the facade layer. For example, a metrics analysis plug-in could be used to calculate metrics for Java and for TTCN-3 source code without any modifications—as the facade layer provides a unified access to the different analysis targets.

We designed our own facade layer, because we experienced that existing XML representations of software artefacts (see section 2.3) were either too specific to the particular type of artefact or too abstract to provide relevant detailed information. Instead of creating the union of all required information (or alternatively only the intersection of common concepts), the functionality and kind of abstraction that is currently provided by the XAF facade layer is rather driven by the existing applications of the framework (see section 4). Following the practise of extending a framework interface as part of framework evolution, the interface of the XAF facade layer is intended to be stable, but can be

```
1  module namespace example-facade = "facade-namespace";
2  declare variable $example-facade:input external;
3
4  declare function example-facade:get-functions() {
5    $example-facade:input//FunctionDefinition
6  };
```

Listing 7. Example XQuery module of the facade layer (facade function for TTCN-3 using XML representation of an AST)

```
module namespace example-plugin = "plugin-namespace";

import module namespace facade =
  "facade-namespace" at "facade.interfaces.xquery";

declare function example-plugin:functions-per-class() {
  for $class in facade:get-classes()
  return count(facade:get-functions($class))
};
```

Listing 8. Example XQuery module of the analysis plug-in layer

extended by additional functionality to broaden the applicability of XAF as required.

Figure 1 illustrates the design of XAF by two exemplary analysis plug-ins (one for the calculation of metrics and one for the detection of bad smells) and two different analysis targets (Java and TTCN-3 source code respectively) having their own XML representation and thus their own corresponding facade layer implementations that provide the common interface of the facade layer.

The XQuery technology is used for the facade and analysis layer of XAF: For each analysis target—for example a domain-specific language—an XQuery module is required which implements the fixed interface of the facade layer and provides the implementation towards the analysis plug-in layer. An exemplary facade layer module is shown in listing 7: The XML input is bound to the module variable `input` in line 2 and one exemplary facade layer function is declared in the lines 4–6. The XQuery function `get-functions()` allows the retrieval of all function definitions. The function's interface abstracts from the concrete XML element name and thus provides an independence of the concrete underlying XML representation (in the example, the function implements access to an XML element named `FunctionDefinition` (line 5) that is used in an XML representation of a TTCN-3 function definition). However, in those cases where the facade layer does not provide a functionality required for a certain analysis, it is still possible for the analysis plug-ins to sacrifice the target independence by directly accessing the XML representation of a software artefact.

The XQuery modules that serve as plug-ins of the analysis layer access the analysis target via the facade layer interface. They are hence completely independent of the concrete analysis target and also of the XML representation of the analysis target. For example, a simple metrics calculation XQuery module which contains an XQuery function for counting the number of functions per class in a software artefact is shown in listing 8. The module imports the facade layer interface module to make use of its functions like `get-classes()` and `get-functions()`. The logic itself is completely encapsulated in the metrics module and can be re-used without modifications for any kind of analysis targets for which a reasonable definition of the `get-classes()` and `get-functions()` facade layer XQuery functions is possible.

The actual analysis is performed by executing the XQuery expressions using an XQuery processor like Saxon [29]. The result of an analysis consists of matches which are instances of patterns detected in the XML representation of the software artefact. These results should be presented by the embedding application in the analysis target and not in its XML representation format. Hence, representation formats need to fulfil some requirements to allow the mapping of matches from the representation format back to the original analysis target. The framework takes care of this and returns matches according to an XML schema definition (`matches.xsd`) in order to allow the embedding application like an IDE to display the matches in the underlying analysis target.

The framework also provides some additional features, which ease the integration into an embedding application like an IDE: An XML schema definition (`hierarchy.xsd`) is provided for a further XML document that allows to enumerate and structure the available analysis plug-ins that contain the XQuery analysis functions. Hence, XAF can be extended with new analysis plug-ins and rules in a completely declarative way. Furthermore, XQuery library modules for commonly used tasks are offered by XAF, for example to access symbol table or cross-reference information, if the embedding application offers them. This library includes also an infrastructure for the detection of duplicated elements, e.g. duplicated code. This functionality is implemented by matching XML sub-trees with each other, mainly by using the XPath function `deep-equals()` which determines whether two XML nodes and their children are equal or not.

4. Applications of the framework

To evaluate the applicability of our XQuery-based Analysis Framework, we have instantiated it in several experiments. The goal of these experiments were

1. to validate the feasibility of the concepts provided by XAF by demonstrating that the framework can be used for realising the functionality of existing tools that are specific for a certain artefact and class of analysis,

2. to substantiate the ease of exchangeability of the underlying analysis target or classes of analysis,

3. to study the integration of XAF into existing IDEs,

4. to get an impression of the reduction of efforts for specifying analysis rules using the high-level XQuery style instead of implementing them on a low-level using a general purpose programming language.

In our experiments, we used XAF to analyse three different kinds of software artefacts and applied two different classes of analyses to them. The analysed targets were: programming language source code (Java), UML models (given in XMI format), and test specifications (TTCN-3 textual core notation). The analysis classes were: detection of bad smells [11] and calculation of product metrics [10].

Currently, the most extensive analysis plug-in for XAF consists of XQuery modules that are able to detect various bad smells in software artefacts: we provide a module that supports the detection of generic bad smells that relate to code duplication, long statement blocks, long parameter lists, magic numbers, and nested conditionals. In addition, we have developed a module that is able to detect instances of 14 bad smells specific to TTCN-3 [22].

In the following, XQuery functions for the detection of the bad smells *Magic Number*, *Long Parameter List*, and *Duplicate Code in Conditionals* are presented in detail. The bad smell *Magic Number* refers to the usage of hard-coded numbers instead of symbolic constants. *Long Parameter List* refers to confusingly long parameter lists and *Duplicate Code in Conditionals* relates to conditional statements containing the same branches more than once.

The bad smell *Magic Number* is detected by iterating over all numeric values (listing 9, line 3). The following `where` clause (lines 5 and 6) skips numeric values in constant definitions and those values of an ignore list, which was passed to the function as a parameter in line 1. Finally, the found magic value is returned in line 7.

The second smell detection function (listing 10) searches for instances of a *Long Parameter List* and makes use of an XQuery `for` loop to iterate over all parameterisable constructs like functions or classes (lines 3 and 4). The parameters of such constructs are bound to an XQuery variable in lines 5 and 6. The number of parameters required to qualify a parameter list as too long is passed in as parameter in line 1. It is compared to the number of parameters of the current construct which is computed in line 7. If the comparison evaluates to true, the construct having a long parameter list is finally returned.

```
1  declare function smell:magic-number($ignore)
2  {
3    for $value in facade:get-numeric-values(())
4    (: skip constants and values in the ignore list :)
5    where not(facade:is-const-value($value)) and
6      not(functx:is-value-in-sequence($value, $ignore))
7    return $value
8  };
```

Listing 9. XAF smell detection function for the smell *Magic Number*

```
1  declare function smell:long-parameter-list($floor)
2  {
3    for $parametrizable-construct in
4      facade:get-parametrizable-constructs(())
5    let $parameters :=
6      facade:get-parameters($parametrizable-construct)
7    where count($parameters) > $floor
8    return $parametrizable-construct
9  };
```

Listing 10. XAF smell detection function for the smell *Long Parameter List*

Listing 11 shows a simplified version of the smell detection function to find *Duplicate Code in Conditionals*. Each duplicate search starts with the definition of the scope for the comparison—in this case all blocks (line 3). For each block, all `then` and `else` branches are selected (lines 4–8) to be checked for duplicate code. The actual comparison is performed by the function `find-duplicates()` that is provided by an XAF library module as described at the end of section 3. In line 9 of listing 11, this function is called with the XML nodes that have been retrieved in lines 3–8. Hence, the `find-duplicates()` function line 9 will return all `then` and `else` branches that are exact duplicates in the XML representation of the analysed software artefact.

A second class of analyses that has been realised using XAF is the calculation of metrics. Listing 12 shows an example that calculates the metric *Lines of Code* which counts all non-empty and non-comment lines of a source code file. Using XQuery and the facade layer interface, this is done by iterating over all files (line 3) and returning for each file an XML element `lines-of-code` (line 4). Each of these elements contains the according filename as attribute (line 5) and the number of lines of code as content. The calculation of the lines of code is based on the location information contained in the XML document: the line information where each entity of the source code starts and ends are joined in one sequence (lines 7 and 8). The total number of the distinct values of this sequence corresponds to the lines of code (calculated using the XQuery functions `count()` and `distinct-values()` in line 6).

```
declare function smell:duplicate-code-in-conditionals()
{
  for $scope in facade:get-blocks(())
  let $to-compare := (
    let $if := facade:get-if-constructs($scope)
    return (facade:get-if-branch($if) |
        facade:get-else-branch($if))
  )
  return lib:find-duplicates($to-compare)
};
```

Listing 11. XAF smell detection function for the smell *Duplicate Code in Conditionals*

```
declare function metric:lines-of-code()
{
  for $file in facade:get-files(())
  return element lines-of-code {
    facade:get-filename($file),
    count(distinct-values(
      (facade:get-lines($file),
      facade:get-line-ends($file))
    ))
  }
}
```

Listing 12. XAF metric calculation function for the metric *Lines of Code*

```
1 declare function get-parametrizable-constructs($n) {
2   facade:get-functions($n)
3 };
4 declare function facade:get-functions($n) {
5   facade:get-node($n)//function
6 };
7 declare function facade:get-parameters($n) {
8   facade:get-node($n)//parameter_list/param
9 };
```

Listing 13. Facade functions for Java in srcML flavour

```
1 declare namespace UML = "org.omg.xmi.namespace.UML";
2 declare function facade:get-functions($n) {
3   facade:get-node($n)//UML:Operation
4 };
5 declare function facade:get-parameters($n) {
6   facade:get-node($n)//UML:BehavioralFeature.parameter/
        UML:Parameter
7 };
```

Listing 14. Facade functions for UML in XMI flavour

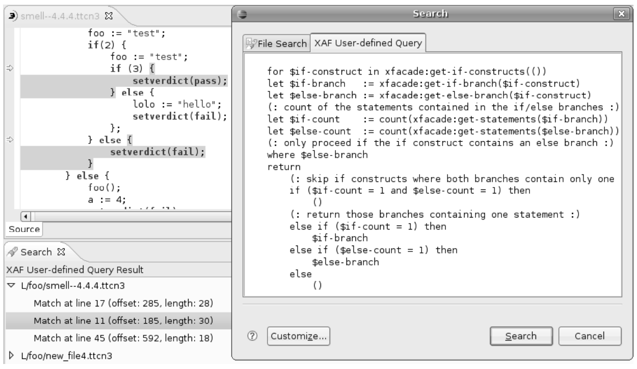
Some analysis functions may make assumptions that are not valid for all analysis targets: for example, a smell detection function that relates to the existence of goto statements is not reasonable applicable for UML as the notion of goto is not part of UML; it depends on the developer of the facade layer how these cases are handled: a facade layer function that just returns an empty result may be implemented or—by not implementing a facade layer function—an error may be thrown at run-time indicating that the corresponding analysis is not applicable.

To support the detection of bad smells on different targets, we have developed corresponding facade layer modules for Java, UML, and TTCN-3. By providing the different facade layer modules, it was possible to re-use the previously described analysis functions without changes.

The developed facade layer for Java artefacts expects as XML representation the srcML format. This format can automatically be generated from Java source code using the `src2srcml` tool from the srcML toolkit [16]. The XQuery functions of the Java facade layer that are required for supporting the XQuery analysis function that detects instances of the bad smell *Long Parameter List* (see listing 10) are shown in listing 13. For reasons of clarity, only functions are treated as parameterisable constructs for the following facade implementations. Therefore, the function in lines 1–3 of listing 13 just delegates calls to `get-functions()`. Java functions correspond to `function` elements in the srcML document (line 5 of listing 13). The associated parameters can be found in the `param` sub-nodes of `parameter_list` elements (line 8).

The facade layer for UML expects that UML models are represented using the XMI format that is provided by modern UML tools. Listing 14 shows again the facade functions that are required by the smell detection function for *Long Parameter List*. The only kind of considered parameterisable constructs are UML operations (as, e.g., defined in class diagrams) that are matched by the XQuery expression in line 3 of listing 14. Line 6 navigates to the associated parameters of the parameter list in the XMI representation. The facade layer function `get-parametrizable-constructs()` is the same for the srcML and UML facade and therefore not repeated in listing 14.

The support for targeting TTCN-3 test specifications has been realised by integrating XAF into the Eclipse-based open-source TTCN-3 tool TRex [20, 21, 33]. TRex provides advanced IDE functionality for the standardised TTCN-3 language [7]. TTCN-3 has a textual C-like syntax and TRex uses internally a TTCN-3 parser that creates an AST and a symbol table from TTCN-3 files. The XAF integration into the TRex IDE includes the automatic conversion of the TTCN-3 AST to XML. Listing 7 that has been described in section 3 shows an excerpt of the TTCN-3 facade layer that operates on the XML representation of the TRex TTCN-3 AST.



```
        foo := "test";
        if(2) {
            foo := "test";
            if (3) {
                setverdict(pass);
            } else {
                lolo := "hello";
                setverdict(fail);
            };
        } else {
            setverdict(fail);
        }
    } else {
        foo();
        a := 4;
```

```
XAF User-defined Query Result
L/foo/smell--4.4.4.ttcn3
    Match at line 17 (offset: 285, length: 28)
    Match at line 11 (offset: 185, length: 30)
    Match at line 45 (offset: 592, length: 18)
L/foo/new_file4.ttcn3
```

```
for $if-construct in xfacade:get-if-constructs(())
let $if-branch   := xfacade:get-if-branch($if-construct)
let $else-branch := xfacade:get-else-branch($if-construct)
(: count of the statements contained in the if/else branches :)
let $if-count    := count(xfacade:get-statements($if-branch))
let $else-count  := count(xfacade:get-statements($else-branch))
(: only proceed if the if construct contains an else branch :)
where $else-branch
return
    (: skip if constructs where both branches contain only one
    if ($if-count = 1 and $else-count = 1) then
        ()
    (: return those branches containing one statement :)
    else if ($if-count = 1) then
        $if-branch
    else if ($else-count = 1) then
        $else-branch
    else
        ()
```

Figure 2. Integration of XAF into the TRex IDE

The user interface that has been created as part of the integration of XAF into the TRex IDE supports the execution of analysis runs which consist of arbitrary numbers of analysis functions from any analysis plug-in modules available, the structured display of analysis results, and the addition of user-defined XQuery analysis rules to extend the set of 20 predefined smell detection functions. Figure 2 gives an impression how the dialogue for the definition of user-defined queries looks like and how search results are presented.

An evaluation of the XAF integration into the TRex IDE revealed that the framework implementation satisfies the most important requirements: the analysis results are correct, new user-defined rules can easily be added, and the performance is satisfactory.

The instantiation of XAF for the detection of bad smells in TTCN-3 test suites allowed the comparison with our previous TTCN-3 smell detection tool that was implemented purely in Java [19]: The XAF-based approach detected the same instances of bad smells in TTCN-3 test suites as the Java-based tool that we used as reference did.

In terms of efforts, using XAF and high-level XQuery expressions made it possible to implement more smell detection functions in less time than using the low-level Java approach. However, in terms of run-time, the Java implementation performed better than the XAF approach: an analysis run that included the detection of duplicated code in 15,485 lines of TTCN-3 code took 2.4 seconds with the Java-based approach while the same analysis with XAF took 22.3 seconds. This performance loss seems to be the price that needs to be paid for the high-level approach. However, there is potential for increasing the performance: The XML representation of the TRex TTCN-3 AST is quite elaborate and thus, the resulting XML files are relatively large (about 1.6 megabyte per thousand lines of TTCN-3 code). This leads to longer analysis time than necessary in particular due to the creation of the XML file by TRex and the subsequent import of the XML file by the XQuery processor. Other XML representations similar to srcML would be more appropriate as they are more compact and help to save resources.

These experiments successfully demonstrate that the goals of a high-level description of queries and patterns, the target-independence of the analyses, and the extensibility towards new classes and targets of analyses have been reached.

5. Related work

Several approaches are related to XAF. The work that is discussed in the following has the outstanding property in common to allow a software analysis (at least mostly) independent of the target software artefact.

Some of the XML-based representation formats, such as *scrML* [16] or *OOML* [18], allow a unified representation of several programming languages and can be used together with XQuery for analysing source code. However, they support only a set of closely related programming languages and are thus not as independent of the target software artefact as XAF.

TAWK [3] makes use of the core concepts of regular expressions. It is based on the AWK language and uses a language-independent pattern syntax which combines the lexical power of AWK with matching support for abstract syntax trees. "Retargeting to a new language requires [...] no special effort [...] to the pattern matcher itself, since it is language independent" [3]. While TAWK's patterns itself are language-independent, the queries are restricted to ASTs. This allows the analysis of a wide range of programming languages, but is still limited compared to XAF which supports also the analysis of software artefacts that are not necessarily represented as ASTs, but as XML.

The *Generic Transformation Language* (GenTL) [31] is an approach for software analysis based on logic-meta programming. Although its name states that GenTL is a transformation language, it also provides analysis capabilities. GenTL extends the logic-based programming by so-called *Concrete Syntax Patterns* (CSP). CSPs are snippets of the programming language under investigation containing meta-variables. Those meta-variable are logic predicates and act as placeholders for expressions of the underlying programming language. An example GenTL pattern matching expression that finds all classes and binds them to the user-defined meta-variable `?allClasses` is as follows: `?allClasses is [[ class ?classname ??class_members ]]`. "Since CSPs match at the AST level, matching is not restricted to lexical structures" [2]. Hence, the above example also matches abstract classes even though this keyword is not contained in the pattern. GenTL is adapted for Java but its core concepts are language-independent [2]. However, the usage of concrete syntax patterns complicates the adaptation towards new analysis targets, in particular compared to approaches like TAWK and XAF that avoid the usage of snippets from the analysis target language for its own analysis rules.

The *Source Code Algebra* (SCA) [25, 26] follows the idea of modelling source code using an algebra. Hence, SCA plays the same role for source code as the relational algebra plays for relational databases. The algebra is developed as a theoretical foundation for a powerful source code query system. The benefits of using SCA include the integration of structural and flow information and the ability to process high-level source code queries using SCA expressions. The SCA query language itself is domain-independent which is a valuable feature and means that an implementation of an SCA query processor works unchanged across different SCA domain models [25]. But as SCA domain models vary from one programming language to another, each model must be redesigned and implemented to reflect the specifics of a given programming language. Furthermore, while XAF is a generic analysis framework which allows the analysis of source code as well as other software artefacts, SCA is strictly bound to the analysis of source code.

PMD [27] is a tool for scanning Java source code for potential problems. Even though this approach is *not* language independent at all, it is presented here as it makes use of XPath to analyse software artefacts and therefore, it is related to XAF. However, PMD is limited compared to XAF as it is restricted to Java. Additionally, PMD makes use of XPath 1.0 while XAF benefits from XQuery 1.0 that includes XPath 2.0. A potential advantage of PMD over XAF is its application programming interface which allows also the addition of new powerful analysis rules in a procedural way—compared to the pure XQuery-based approach of XAF.

6. Summary and outlook

We presented our XQuery-based Analysis Framework (XAF) for a flexible quality assurance of software artefacts: analysis rules can be easily specified with a high-level approach based on queries and pattern matching expressions. In contrast to comparable approaches, no proprietary rule language needs to be learnt for XAF—instead the standardised XQuery language is used. The power of XQuery removes the burden of implementing the actual low-level matching of patterns.

Due to its layered design, XAF allows the creation of analysis rules that are independent of the target software artefact. Thus, it is easy to support domain-specific languages that are frequently used in software testing. A facade layer is used for the adaptation of the abstract XQuery analysis rules to individual XML representations of the target software artefacts. Thus, facade layer implementations and analysis rules can be independently re-used for analysing different kinds of software artefacts or for different classes of analyses respectively. XAF is available as open-source software as part of the TRex TTCN-3 Refactoring and Metrics tool project [33].

Several instantiations of the framework for varying targets and analyses were used to validate the applicability of XAF. Experiences with developing analyses for detecting bad smells in TTCN-3 test specifications showed that it is easier and faster to create analysis rules using high-level XQuery expressions and the functionality provided by XAF than implementing the same rules on a low-level in Java.

As future work, we would like to use XAF for further classes of analyses, e.g. for the recovery of test patterns in TTCN-3 test specifications and for supporting traceability between test purposes and test cases by identifying scenarios specified with the semi-formal test purpose language TPLan [8, 30] in TTCN-3 test case specifications, i.e. analyses across the boundary of one language.

References

[1] R. Al-Ekram and K. Kontogiannis. An XML-Based Framework for Language Neutral Program Representation and Generic Analysis. In *CSMR '05: Proceedings of the Ninth European Conference on Software Maintenance and Reengineering*. IEEE Computer Society, 2005.

[2] M. Appeltauer and G. Kniesel. Towards Concrete Syntax Patterns for Logic-Based Transformation Rules. In *Proceedings of the Eighth International Workshop on Rule-Based Programming (RULE'07), Workshop at Federated Conference on Rewriting, Deduction, and Programming (RDP 2007)*, 2007.

[3] D. Atkinson and W. Griswold. Effective Pattern Matching of Source Code Using Abstract Syntax Patterns. *Software – Practice & Experience*, 36(4):413–447, 2006.

[4] G. Badros. JavaML: A Markup Language for Java Source Code. *Computer Networks*, 33(1-6):159–177, 2000.

[5] T. Bray, M. Sperberg-McQueen, F. Yergeau, E. Maler, and J. Paoli. Extensible Markup Language (XML) 1.0 (Fourth Edition). World Wide Web Consortium Recommendation, Aug. 2006. `http://www.w3.org/TR/2006/REC-xml-20060816`.

[6] J. Ebert, B. Kullbach, and A. Winter. Querying as an Enabling Technology in Software Reengineering. In *CSMR '99: Proceedings of the Third European Conference on Software Maintenance and Reengineering*. IEEE Computer Society, 1999.

[7] ETSI. Standard ES 201 873-1 V3.4.1 (2008-09), Part 1: TTCN-3 Core Language. European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France, 2007.

[8] ETSI. Standard ES 202 553 V1.1.1 (2008-02) TPLan: A notation for expressing Test Purposes. European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France, 2008.

[9] M. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.

[10] N. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co. Boston, MA, USA, 1997.

[11] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing, Boston, MA, USA, 1999.

[12] D. Freedman and G. Weinberg. *Handbook of Walkthroughs, Inspections, and Technical Reviews: Evaluating Programs, Projects, and Products*. Dorset House Publishing Company, New York, USA, 1990.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston, MA, USA, 1995.

[14] J. Grabowski, D. Hogrefe, G. Réthy, I. Schieferdecker, A. Wiles, and C. Willcock. An Introduction into the Testing and Test Control Notation (TTCN-3). *Computer Networks*, 42(3):375–403, June 2003.

[15] S. Kepser. A Simple Proof for the Turing-Completeness of XSLT and XQuery. In *Extreme Markup Languages 2004*. IDEAlliance, 2004.

[16] J. Maletic, M. Collard, and H. Kagdi. Leveraging XML Technologies in Developing Program Analysis Tools. In *ACSE 2004 Proceedings, 4th International Workshop on Adoption-Centric Software Engineering, Workshop at 26th IEEE/ACM International Conference on Software Engineering (ICSE 2004)*, 2004.

[17] A. Malhotra, K. Rose, J. Siméon, M. Fernández, P. Fankhauser, D. Draper, P. Wadler, and M. Rys. XQuery 1.0 and XPath 2.0 Formal Semantics. World Wide Web Consortium Recommendation, Jan. 2007. `http://www.w3.org/TR/2007/REC-xquery-semantics-20070123`.

[18] E. Mamas and K. Kontogiannis. Towards Portable Source Code Representations Using XML. In *WCRE '00: Proceedings of the Seventh Working Conference on Reverse Engineering*. IEEE Computer Society, 2000.

[19] H. Neukirchen and M. Bisanz. Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites. In *Proceedings of the 19th IFIP International Conference on Testing of Communicating Systems and 7th International Workshop on Formal Approaches to Testing of Software (TestCom/FATES 2007)*, volume 4581 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2007.

[20] H. Neukirchen, B. Zeiss, and J. Grabowski. An Approach to Quality Engineering of TTCN-3 Test Specifications. *International Journal on Software Tools for Technology Transfer (STTT)*, 10(4):309–326, 2008.

[21] H. Neukirchen, B. Zeiss, J. Grabowski, P. Baker, and D. Evans. Quality assurance for TTCN-3 test specifications. *Software Testing, Verification and Reliability (STVR)*, 18(2):71–97, 2008.

[22] J. Nödler. An XML-based Approach for Software Analysis – Applied to Detect Bad Smells in TTCN-3 Test Suites. Master's thesis, Center for Computational Sciences, University of Göttingen, Germany, ZFI-BM-2007-36, ISSN 1612-6793, Oct. 2007.

[23] Object Management Group, MOF 2.0/XMI Mapping Specification, v2.1. `http://www.omg.org/technology/documents/formal/xmi.htm`, 2005.

[24] Object Management Group, Unified Modeling Language (UML), version 2.1.2. `http://www.omg.org/technology/documents/formal/uml.htm`, 2007.

[25] S. Paul and A. Prakash. Supporting Queries on Source Code: A Formal Framework. *International Journal of Software Engineering and Knowledge Engineering*, 4(3):325–348, 1994.

[26] S. Paul and A. Prakash. A Query Algebra for Program Databases. *IEEE Transactions on Software Engineering*, 22(3):202–217, 1996.

[27] PMD website. `http://pmd.sourceforge.net`, 2008.

[28] J. Robie, J. Siméon, M. Fernández, D. Chamberlin, D. Florescu, and S. Boag. XQuery 1.0: An XML Query Language. World Wide Web Consortium Recommendation, Jan. 2007. `http://www.w3.org/TR/2007/REC-xquery-20070123`.

[29] Saxon – The XSLT and XQuery Processor website. `http://saxon.sourceforge.net`, 2008.

[30] S. Schulz, A. Wiles, and S. Randall. TPLan-A Notation for Expressing Test Purposes. In *Proceedings of the 19th IFIP International Conference on Testing of Communicating Systems and 7th International Workshop on Formal Approaches to Testing of Software (TestCom/FATES 2007)*, volume 4581 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2007.

[31] D. Speicher, M. Appeltauer, and G. Kniesel. Code Analyses for Refactoring by Source Code Patterns and Logical Queries. In *Proceedings of the 1st Workshop on Refactoring Tools held in conjunction with 21st European Conference on Object-Oriented Programming (ECOOP 2007). Technical Report No 2007-8, ISSN 1436-9915*. Technical University Berlin, 2007.

[32] S. Tichelaar, S. Ducasse, S. Demeyer, and O. Nierstrasz. A Meta-Model for Language-Independent Refactoring. In *Proceedings of the International Workshop on Principles of Software Evolution (IWPSE)*. IEEE Computer Society Press, 2000.

[33] TRex – The TTCN-3 Refactoring and Metrics Tool website. `http://www.trex.informatik.uni-goettingen.de`, 2008.

[34] E. van Emden and L. Moonen. Java Quality Assurance by Detecting Code Smells. In *WCRE '02: Proceedings of the Ninth Working Conference on Reverse Engineering*. IEEE Computer Society, 2002.

[35] P. Van Gorp, N. Van Eetvelde, and D. Janssens. Implementing Refactorings as Graph Rewrite Rules on a Platform Independent Metamodel. In *Proceedings of the 1st International FUJABA Days. Technical Report tr-ri-04-247*. University of Paderborn, 2003.

[36] J. Zheng, L. Williams, W. Snipes, N. Nagappan, J.Hudepohl, and M. Vouk. On the value of static analysis tools for fault detection. *IEEE Transactions on Software Engineering*, 32(4):240–253, 2006.